

Linux Driver Development for Embedded Processors

ALBERTO LIBERAL DE LOS RÍOS

Copyright © 2018, Alberto Liberal de los Ríos. 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 de los Ríos
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.

<http://www.rejoiceblog.com/>

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 a Linux Embedded System	30
Setting up Ethernet Communication	31
Building a Linux Embedded System for the NXP i.MX7D Processor	32
Introduction.....	32
Host Packages	33
Setting up the Repo Utility	33
Yocto Project Setup and Image Building	33
Working Outside of Yocto.....	36
Building the Linux Kernel.....	38
Installing a TFTP Server.....	41
Installing a NFS Server.....	42
Setting the U-Boot Environment Variables	42
Building a Linux Embedded System for the Microchip SAMA5D2 Processor.....	43
Introduction.....	43
Host Packages	44
Yocto Project Setup and Image Building	44
Working Outside of Yocto.....	46
Building the Linux Kernel.....	47
Installing a TFTP Server.....	49
Installing a NFS Server.....	49
Setting the U-Boot Environment Variables	50

Table of Contents

Building a Linux Embedded System for the Broadcom BCM2837 Processor.....	50
Raspbian	50
Building the Linux Kernel.....	51
Copying Files to your Raspberry Pi.....	54
Working with Eclipse.....	55
Eclipse Configuration for Working with Kernel Sources	56
Eclipse Configuration for Developing Linux Drivers	67
Chapter 2: The Linux Device and Driver Model.....	73
Bus Core Drivers	74
Bus Controller Drivers.....	76
Device Drivers	77
Introduction to the Device Tree.....	78
Chapter 3: The Simplest Drivers	83
Licensing.....	84
LAB 3.1: "helloworld" Module	84
Listing 3-1: helloworld_imx.c.....	85
Listing 3-2: Makefile.....	86
helloworld_imx.ko Demonstration	86
LAB 3.2: "helloworld with parameters" Module.....	87
Listing 3-3: helloworld_imx_with_parameters.c.....	88
helloworld_imx_with_parameters.ko Demonstration	88
LAB 3.3: "helloworld timing" Module.....	88
Listing 3-4: helloworld_imx_with_timing.c	89
helloworld_imx_with_timing.ko Demonstration	90
Chapter 4: Character Drivers.....	91
LAB 4.1: "helloworld character" Module	93
Registration and Unregistration of Character Devices	94
Listing 4-1: helloworld_imx_char_driver.c	98
Listing 4-2: Makefile.....	102
Listing 4-3: ioctl_test.c	102
helloworld_imx_char_driver.ko Demonstration	102
Add the Module to the Kernel Build	103

Creating Device Files with devtmpfs.....	104
LAB 4.2: "class character" Module	105
Listing 4-4: helloworld_imx_class_driver.c.....	107
helloworld_imx_class_driver.ko Demonstration	109
Miscellaneous Character Driver.....	109
Registering a Minor Number.....	110
LAB 4.3: "miscellaneous character" Module	111
Listing 4-5: misc_imx_driver.c.....	111
misc_imx_driver.ko Demonstration	113
Chapter 5: Platform Drivers.....	115
LAB 5.1: "platform device" Module	117
Listing 5-1: hellokeys_imx.c.....	120
hellokeys_imx.ko Demonstration	122
Documentation to Interact with the Hardware.....	123
Hardware Naming Convention.....	123
Pin Controller.....	124
Pin Control Subsystem	127
Device Tree Pin Controller Bindings.....	134
GPIO Controller Driver.....	138
GPIO Descriptor Consumer Interface.....	140
Obtaining and Disposing GPIOs.....	141
Using GPIOs.....	141
GPIOs Mapped to IRQs.....	143
GPIOs in Device Tree.....	143
Exchanging Data between Kernel and User Space.....	144
MMIO (Memory-Mapped I/O) Device Access	145
LAB 5.2: "RGB LED platform device" Module	147
LAB 5.2 Hardware Description for the i.MX7D Processor	147
LAB 5.2 Hardware Description for the SAMA5D2 Processor	149
LAB 5.2 Hardware Description for the BCM2837 Processor	151
LAB 5.2 Device Tree for the i.MX7D Processor	153
LAB 5.2 Device Tree for the SAMA5D2 Processor	156
LAB 5.2 Device Tree for the BCM2837 Processor	159
LAB 5.2 Code Description of the "RGB LED platform device" Module.....	161

Table of Contents

Listing 5-2: ledRGB_sam_platform.c	166
ledRGB_sam_platform.ko Demonstration.....	171
Platform Driver Resources.....	172
Linux LED Class	174
LAB 5.3: "RGB LED class" Module.....	176
LAB 5.3 DT for the i.MX7D, SAMA5D2 and BCM2837 Processors	176
LAB 5.3 Code Description of the "RGB LED class" Module	180
Listing 5-3: ledRGB_sam_class_platform.c	184
ledRGB_sam_class_platform.ko Demonstration.....	188
Platform Device Drivers in the User Space	188
User Defined I/O: UIO.....	190
How UIO Works.....	192
Kernel UIO API.....	193
LAB 5.4: "LED UIO platform" Module	195
LAB 5.4 DT for the i.MX7D, SAMA5D2 and BCM2837 Processors	195
LAB 5.4 Code Description of the "LED UIO platform" Module	197
Listing 5-4: led_sam_UIO_platform.c	199
Listing 5-5: UIO_app.c	201
led_sam_UIO_platform.ko with UIO_app Demonstration.....	203
Chapter 6: I2C Client Drivers.....	205
The Linux I2C Subsystem.....	206
Writing I2C Client Drivers	210
I2C Client Driver Registration	210
Declaration of I2C Devices in Device Tree	212
LAB 6.1: "I2C I/O expander device" Module	214
LAB 6.1 Hardware Description for the i.MX7D Processor	214
LAB 6.1 Hardware Description for the SAMA5D2 Processor	214
LAB 6.1 Hardware Description for the BCM2837 Processor.....	216
LAB 6.1 Device Tree for the i.MX7D Processor	216
LAB 6.1 Device Tree for the SAMA5D2 Processor	218
LAB 6.1 Device Tree for the BCM2837 Processor	220
LAB 6.1 Code Description of the "I2C I/O expander device" Module	221
Listing 6-1: io_imx_expander.c	224
io_imx_expander.ko Demonstration.....	228

The Sysfs Filesystem.....	229
The Kobject Infrastructure	230
LAB 6.2: "I2C multidisplay LED" Module	233
LAB 6.2 Hardware Description for the i.MX7D Processor	235
LAB 6.2 Hardware Description for the SAMA5D2 Processor	235
LAB 6.2 Hardware Description for the BCM2837 Processor.....	235
LAB 6.2 Device Tree for the i.MX7D Processor	235
LAB 6.2 Device Tree for the SAMA5D2 Processor	237
LAB 6.2 Device Tree for the BCM2837 Processor	239
Unified Device Properties Interface for ACPI and Device Tree	240
LAB 6.2 Code Description of the "I2C multidisplay LED" Module	242
Listing 6-2: ltc3206_imx_led_class.c.....	246
ltc3206_imx_led_class.ko Demonstration.....	254
Chapter 7: Handling Interrupts in Device Drivers	257
Linux Kernel IRQ Domain for GPIO Controllers	260
Device Tree Interrupt Handling.....	268
Requesting Interrupts in Linux Device Drivers.....	272
LAB 7.1: "button interrupt device" Module	274
LAB 7.1 Hardware Description for the i.MX7D Processor	274
LAB 7.1 Hardware Description for the SAMA5D2 Processor	274
LAB 7.1 Hardware Description for the BCM2837 Processor.....	274
LAB 7.1 Device Tree for the i.MX7D Processor	274
LAB 7.1 Device Tree for the SAMA5D2 Processor	276
LAB 7.1 Device Tree for the BCM2837 Processor	277
LAB 7.1 Code Description of the "button interrupt device" Module	278
Listing 7-1: int_imx_key.c.....	280
int_imx_key.ko Demonstration	282
Deferred Work	283
Softirqs	284
Tasklets	286
Timers.....	286
Threaded Interrupts	290
Workqueues	292

Table of Contents

Locking in the Kernel	296
Locks and Uniprocessor Kernels.....	297
Sharing Spinlocks between Interrupt and Process Context	297
Locking in User Context	298
Sleeping in the Kernel.....	298
LAB 7.2: "sleeping device" Module.....	300
LAB 7.2 Device Tree for the i.MX7D Processor	301
LAB 7.2 Device Tree for the SAMA5D2 Processor	302
LAB 7.2 Device Tree for the BCM2837 Processor	303
LAB 7.2 Code Description of the "sleeping device" Module	305
Listing 7-2: int_imx_key_wait.c.....	308
int_imx_key_wait.ko Demonstration.....	312
Kernel Threads	312
LAB 7.3: "keyled class" Module	313
LAB 7.3 Hardware Description for the i.MX7D Processor	314
LAB 7.3 Hardware Description for the SAMA5D2 Processor	314
LAB 7.3 Hardware Description for the BCM2837 Processor.....	315
LAB 7.3 Device Tree for the i.MX7D Processor	316
LAB 7.3 Device Tree for the SAMA5D2 Processor	319
LAB 7.3 Device Tree for the BCM2837 Processor	322
LAB 7.3 Code Description of the "keyled class" Module	324
Listing 7-3: keyled_imx_class.c.....	331
keyled_imx_class.ko Demonstration.....	343
Chapter 8: Allocating Memory with Linux Drivers	345
Walking ARM MMU Translation Tables.....	346
Linux Address Types	353
User Process Virtual to Physical Memory Mapping.....	354
Kernel Virtual to Physical Memory Mapping	355
Kernel Memory Allocators.....	357
PAGE Allocator.....	357
Page Allocator API	358
SLAB Allocator	358
SLAB Allocator API.....	361
Kmalloc Allocator.....	362

LAB 8.1: "linked list memory allocation" Module	363
Listing 8-1: linkedlist_imx_platform.c	366
linkedlist_imx_platform.ko Demonstration	371
Chapter 9: Linux DMA in Device Drivers	373
Cache Coherency	373
Linux DMA Engine API	375
Types of DMA Mappings	378
LAB 9.1: "streaming DMA" Module	382
Listing 9-1: sdma_imx_m2m.c	390
sdma_imx_m2m.ko Demonstration	395
DMA Scatter/Gather Mappings	396
LAB 9.2: "scatter/gather DMA device" Module	397
Listing 9-2: sdma_imx_sg_m2m.c	400
sdma_imx_sg_m2m.ko Demonstration	407
DMA from User Space	407
LAB 9.3: "DMA from user space" Module	409
Listing 9-3: sdma_imx_mmap.c	411
Listing 9-4: sdma.c	416
sdma_imx_mmap.ko Demonstration	417
Chapter 10: Input Subsystem Framework for Device Drivers	419
Input Subsystem Drivers	420
LAB 10.1: "input subsystem accelerometer" Module	423
Device Tree	425
Input Framework as an I2C Interaction	426
Input Framework as an Input Device	428
Listing 10-1: i2c_imx_accel.c	431
i2c_imx_accel.ko Demonstration	433
Using SPI with Linux	435
The Linux SPI Subsystem	437
Writing SPI Client Drivers	440
SPI Client Driver Registration	440
Declaration of SPI Devices in Device Tree	442

Table of Contents

LAB 10.2: "SPI accel input device" Module	446
LAB 10.2 Hardware Description for the i.MX7D Processor	446
LAB 10.2 Hardware Description for the SAMA5D2 Processor	447
LAB 10.2 Hardware Description for the BCM2837 Processor	447
LAB 10.2 Device Tree for the i.MX7D Processor	447
LAB 10.2 Device Tree for the SAMA5D2 Processor	449
LAB 10.2 Device Tree for the BCM2837 Processor	451
LAB 10.2 Code Description of the "SPI accel input device" Module	451
Listing 10-2: adxl345_imx.c	462
adxl345_imx.ko Demonstration.....	476
Chapter 11: Industrial I/O Subsystem for Device Drivers	479
IIO Device Sysfs Interface	481
IIO Device Channels.....	481
The iio_info Structure.....	484
Buffers	485
IIO Buffer Sysfs Interface	485
IIO Buffer Setup.....	485
Triggers	487
Triggered Buffers.....	487
Industrial I/O Events	489
Delivering IIO Events to User Space	492
IIO Utils	494
LAB 11.1: "IIO subsystem DAC" Module	494
Device Tree	495
Industrial Framework as an I2C Interaction.....	497
Industrial Framework as an IIO Device	498
Listing 11-1: ltc2607_imx_dual_device.c.....	504
LAB 11.2: "IIO subsystem DAC" Module with "SPIDEV dual ADC user"	
Application	508
LAB 11.2 Hardware Description for the i.MX7D Processor	510
LAB 11.2 Hardware Description for the SAMA5D2 Processor	510
LAB 11.2 Hardware Description for the BCM2837 Processor	510
LAB 11.2 Device Tree for the i.MX7D Processor	511
LAB 11.2 Device Tree for the SAMA5D2 Processor	511
LAB 11.2 Device Tree for the BCM2837 Processor	512

Listing 11-2: LTC2422_spidev.c	512
ltc2607_imx_dual_device.ko with LTC2422_spidev Demonstration	516
LAB 11.3: "IIO subsystem ADC" Module	518
Device Tree	518
Industrial Framework as a SPI Interaction.....	520
Industrial Framework as an IIO Device	521
Listing 11-3: ltc2422_imx_dual.c.....	524
LTC2422_app User Space Application.....	526
Listing 11-4: ltc2422_app.c	526
ltc2422_imx_dual.ko with ltc2422_app Demonstration	529
LAB 11.4: "IIO subsystem ADC with hardware triggering" Module	529
LAB 11.4 DT for the i.MX7D, SAMA5D2 and BCM2837 Processors	530
Sleep and Wake up in the Driver.....	534
Interrupt Management	535
Listing 11-5: ltc2422_imx_trigger.c	536
ltc2422_imx_trigger.ko with LTC2422_app Demonstration.....	540
Chapter 12: Using the Regmap API in Linux Device Drivers	541
Implementing Regmap.....	543
LAB 12.1: "SPI regmap IIO device" Module	546
Listing 12-1: adxl345_imx_iio.c	557
adxl345_imx_iio.ko Demonstration	569
References	573
Index	575

<http://www.rejoiceblog.com/>

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. Today's processors are made from silicon, which itself is fashioned from one of the most abundant materials on earth: sand. Processor technology has moved from 90nm fabrication in the year 2000 to 14nm today, and it is expected to shrink to 7nm or 5nm by 2021.

Today's embedded processors range from multicore 64-bit CPUs, manufactured in advanced 14nm processes, with extensive heterogeneous computing capabilities including powerful GPUs and DSPs that are engineered to allow the running of trained neural networks and enable next generation of virtual reality applications, to single or dual core embedded processors for power efficient, cost optimized applications designed for the growing IoT (Internet of Things) and industrial markets. Nowdays, it is possible to have an embedded linux system running in a few dollars processor and new processors are coming, shrinking this cost.

The flexibility of Linux embedded, the availability of powerful, energy efficient processors designed for embedded computing and the low cost of new processors are encouraging many industrial companies to come up with new developments based on embedded processors. Current engineers have in their hands powerful tools for developing applications previously unimagined, but they need to understand the countless features that Linux offers today.

Linux embedded firmware developers need to understand the lower level hardware function controls, to be able to write interfaces to multiple peripherals - meaning peripherals: GPIOs, serial buses, timers, DMA, CAN, USB, and LCD.

The following could be a real example of low level hardware control: A Linux embedded firmware developer is designing a Linux application that needs to talk with three different UARTs. A Linux SBC (Single board computer) with three available UARTs is being used, but when the application is being tested, it looks like there are only two functional UARTs. The reason is that processors have pins multiplexed to different functions, and the same pin can be an UART pin, an I2C pin, a SPI pin, a GPIO, etc. To activate the third UART, the firmware developer first must look for Device Tree (DT) source files within the kernel that describe the hardware of the SBC being used. Second, it must be checked if the missing UART device is created and enabled in these DT files. If the UART device node is not included, it can be created using other created UART nodes as a reference. After that, the new UART pads must be multiplexed for UART functionality, making sure they are not conflicting with other devices in the DT that use the same pads for an alternate function.

Preface

In Device Tree Linux systems the device drivers are loaded by the kernel when the device is declared in the Device Tree. The driver retrieves the configuration data from the DT node (e.g, the physical addresses allocated to the device, which interrupt(s) the device triggers, and also device specific information). Throughout this book, you will see the important role of the Device Tree for the development of your Linux device drivers and it will be consequently explained in detail.

This book will teach you how to develop device drivers for Device Tree Linux embedded systems. You will learn how to write different types of Linux drivers as well as the appropriate APIs (Application Program Interfaces) and methods to interface with kernel and user spaces. This is a book is meant to be practical, but also provides an important theoretical base. More than twenty drivers are written and ported to three different processors. You can choose between NXP i.MX7D, Microchip SAMA5D2 and Broadcom BCM2837 processors to develop your drivers. You can use any of these processor boards while it has available some GPIOs, a SPI controller and an I2C controller. I encourage you to acquire one of these boards before you start with the reading, as the content of this book is eminently practical and reproducing the labs with the board will help you to fix the theoretical content exposed throughout the book.

You will learn how to develop drivers, from the simplest ones that do not interact with any external hardware, to drivers that manage different kind of devices: accelerometers, DACs, ADCs, RGB LEDs, Multidisplay LED controllers, I/O expanders, and Buttons. You will also develop DMA drivers, drivers that manage interrupts, and drivers that write/read on the internal registers of the processor to control external devices. To easy the development of some of these drivers, you will use different types of Frameworks: Misc framework, LED framework, UIO framework, Input framework and the IIO industrial one.

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.

Who is this Book for?

This book is ideal for Linux embedded application developers that want to know how to develop drivers from scratch. It is also indicated for embedded software developers that have developed drivers for non-Device Tree kernels and want to learn how to create new Device Tree-based ones. And finally, it is for students and hobbyists that want to learn how to deal with the low level hardware of embedded platforms using Linux. Before reading this book, having prior, basic knowledge of C language, Linux embedded and Yocto Project tools would be helpful, though not mandatory.

How this Book is Structured

Chapter 1, Building the System, starts by describing the main parts of an Embedded Linux system and the different ways to build it, explaining why the Yocto Project and Debian were chosen as your build options. Next, it describes in detail how to build a Linux embedded image using Yocto and Debian and how to compile Linux kernel outside Yocto. The generated Linux image will be used for the development of drivers and applications throughout the book. Finally, this chapter describes how to configure the free Eclipse IDE to develop the drivers.

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

Chapter 3, The Simplest Drivers, covers several simple drivers that are not yet interacting with user applications through "system calls". You will use the Eclipse IDE to create, compile and deploy the drivers in your target board. 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 using system calls and how to exchange data between the kernel and user spaces. It also explains how to identify and create each Linux device. Several drivers will be written that exchange information with the user space using different methods for the creation of the device nodes. The first driver developed will use the traditionally static device creation method using the "mknod" command, the second one will show how to create device files with "devtmpfs" and the last one will create the device files 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 your target processor can be multiplexed to different functions and how to select the required muxing option within the Device Tree. This chapter also describes the Pinctrl Subsystem and the new GPIO Descriptor Consumer Interface. You will develop drivers that control external devices mapping of peripheral addresses from physical to virtual and writing/reading to/from these virtual addresses within kernel space. You will also learn to write drivers that control LEDs using the Linux LED subsystem. Finally, this chapter explains how to develop an user space driver using the UIO framework.

Chapter 6, I2C Client Drivers, describes the Linux I2C subsystem, which is based in the Linux device model. You will learn to declare Device Tree I2C devices and will develop several I2C client drivers throughout this chapter. You will also see how to add "sysfs" support to a platform driver to control the hardware via sysfs entries.

Chapter 7, Handling Interrupts in Platform Drivers, introduces interrupt hardware and software operation in Linux embedded processors, explaining how the interrupt controllers and the interrupt-capable peripheral nodes are linked in the Device Tree. You will develop drivers that manage interrupts from external hardware. You will learn about deferred work kernel facilities that allows one to schedule code to be executed at a later time. This scheduled code can run either in process context using "workqueues" or "threaded interrupts", or in interrupt context using "softirqs", "tasklets" and "timers". Finally, this chapter shows how an user application is put to sleep using a "wait queue" and woken up later via an interrupt.

Chapter 8, Memory Management Drivers, explains the MMU (Memory Management Unit), and the different types of addresses used in Linux. Finally, this chapter describes the different kernel memory allocators.

Chapter 8, DMA Drivers, describes the "Linux DMA Engine Subsystem", and the different types of DMA mappings. Several drivers are developed that manage memory to memory transactions without CPU intervention, using DMA scatter/gather mappings and DMA from user space using the `mmap()` system call.

Chapter 10, Input Subsystem Framework for Device Drivers, 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 focus on the Input subsystem framework, that takes care of the input events coming from the human user. This chapter also describes the Linux SPI subsystem, which is based in the Linux device model. You will learn how to declare Device Tree SPI devices and will develop a SPI client driver using the Input framework. Finally, this chapter explains how to interact from user space with the I2C bus using the "i2c-tools" applications.

Chapter 11, Industrial I/O Subsystem for Device Drivers, describes the IIO (Linux Industrial I/O Subsystem). The IIO subsystem provides support for ADCs, DACs, gyroscopes, accelerometers,

magnetometers, pressure and proximity sensors, etc. It will be explained in detail how to set up IIO triggered buffers and Industrial I/O events. Several IIO subsystem drivers are developed that manage I2C DACs and SPI ADCs with hardware triggering interrupts. This chapter also explains how to interact from user space with the SPI bus using the "spidev" driver.

Chapter 12, Using the Regmap API in Linux Device Drivers, provides an overview of the regmap API, and explains how it will take care of calling the relevant calls of the SPI or I2C subsystem replacing these buses specific core APIs. You will transform the SPI Input subsystem driver of the Chapter 10, which uses specific SPI core APIs, into an IIO SPI 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 SPI IIO driver.

Terminology

New terms, important words and input terminal commands are set in **bold**.

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in **bold**.

Downloading the Kernel Modules Labs

The kernel modules developed in this book can be accessed through the GitHub repository at https://github.com/ALIBERA/linux_book_2nd_edition.

What You Need to Develop the Drivers

The drivers have been tested with an Ubuntu Desktop 14.04 LTS 64-bit system. You can download Ubuntu Desktop at <https://www.ubuntu.com/download>.

Eclipse **Neon** IDE for C/C++ developers has been used to write, compile and deploy the drivers. You can download the Eclipse environment at <https://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/neonr>.

Go to the Neon Packages Release and download the Eclipse IDE for C/C++ developers (Linux 32-bit or 64-bit depending on your Linux host system).

The drivers and the applications have been ported to three different processors: NXP i.MX7D, Microchip SAMA5D2 and Broadcom BCM2837, and these are the hardware platforms that were used to develop them:

ATSAMA5D2B-XULT: The SAMA5D2 Xplained Ultra is a fast prototyping and evaluation platform for the SAMA5D2 series of microprocessors (MPUs):

Preface

- <http://www.microchip.com/developmenttools/ProductDetails/PartNO/ATSAMA5D2B-XULT>

MCIMX7SABRE: SABRE Board for Smart Devices based on the i.MX 7Dual Applications Processors:

- <https://www.nxp.com/support/developer-resources/hardware-development-tools/sabre-development-system/sabre-board-for-smart-devices-based-on-the-i.mx-7dual-applications-processors:MCIMX7SABRE>

Raspberry Pi 3 Model B: Broadcom BCM2837 single-board computer with wireless LAN and Bluetooth connectivity:

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

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.

Acknowledgments

I would like to thank to Daniel Amor from RBZ EMBEDDED LOGICS for the ideas, suggestions and great reviewing work for some of the chapters.

Thanks to my parents that supported me during this project and through my life.

Finally, special thanks to my wife for encouraging me to finish this book, for her love, patience and good humor during this project and always.

1

Building the System

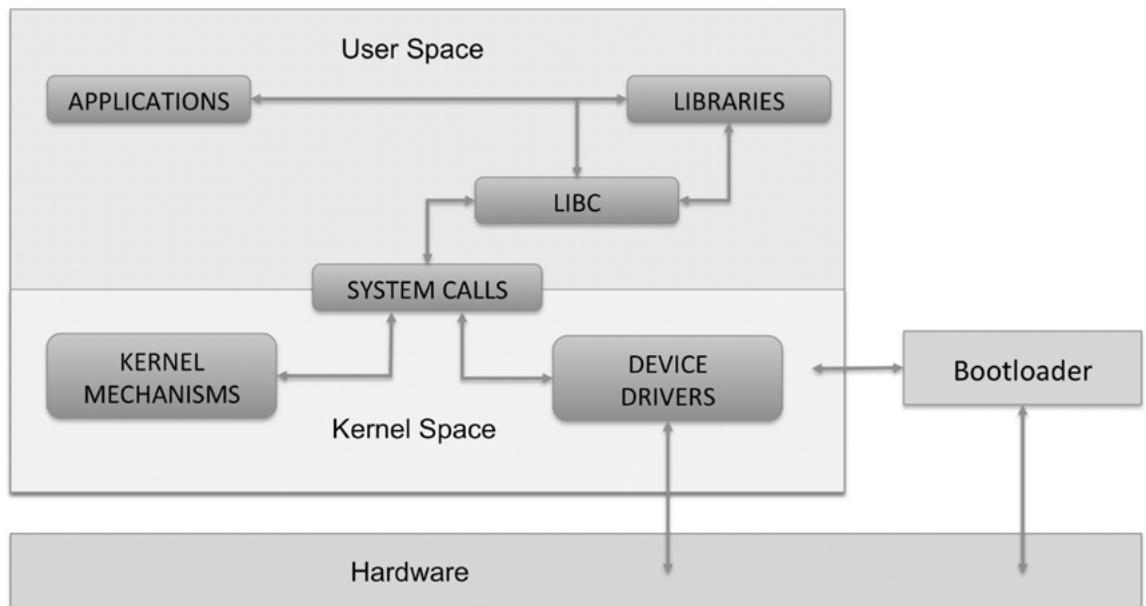
The Linux kernel is one of the largest and most successful open source projects that has ever come about. The huge rate of change and number of individual contributors show that it has a vibrant and active community, constantly stimulating evolution of the kernel. This rate of change continues to increase, 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 from an active community of developers.

These are the main components of a Linux embedded system: **Bootloader**, **Kernel**, **System call interface**, **C-Runtime library**, **System shared libraries** and **Root filesystem**. Each of these components will be described more in detail in the next sections. In the next figure, you can see a high-level Linux embedded architecture:



Bootloader

Linux cannot be started in 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. The U-Boot Mainline is located at <http://git.denx.de/u-boot.git> and there is a dedicated page on U-Boot wiki at <http://www.denx.de/wiki/U-Boot/SourceCode>.

These are some of the main U-Boot features:

1. **Small:** U-Boot is a bootloader, i.e. 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. An 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. For example, RAM initialization and size detection may print a "RAM: " before they start, and "256 MB\n" when done. 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 for 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**: Kernel headers
- **/include/asm-<arch>**: Architecture and machine dependent headers
- **/include/linux**: Linux kernel core headers
- **/init**: Linux initialization (including main.c)
- **/block**: 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.

Longterm release kernels

Version	Maintainer	Released	Projected EOL
4.14	Greg Kroah-Hartman	2017-11-12	Jan, 2020
4.9	Greg Kroah-Hartman	2016-12-11	Jan, 2019
4.4	Greg Kroah-Hartman	2016-01-10	Feb, 2022
4.1	Sasha Levin	2015-06-21	May, 2018
3.16	Ben Hutchings	2014-08-03	Apr, 2020
3.2	Ben Hutchings	2012-01-04	May, 2018

In the following screenshot from www.kernel.org you can see the latest stable kernel, kernels under development (mainline, next), several 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 Stable Kernel:



4.16.13

mainline:	4.17-rc7	2018-05-27	[tarball] [patch] [inc. patch] [view diff] [browse]
stable:	4.16.13	2018-05-30	[tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm:	4.14.45	2018-05-30	[tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm:	4.9.104	2018-05-30	[tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm:	4.4.134	2018-05-30	[tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm:	4.1.52	2018-05-28	[tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm:	3.18.111 [EOL]	2018-05-30	[tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm:	3.16.56	2018-03-19	[tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
longterm:	3.2.101	2018-03-19	[tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]
linux-next:	next-20180529	2018-05-29	[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 kernel.org, you are running a Distribution kernel. To find out the version of your kernel, run `uname -r`:

```
root@imx7dsabresd:~# uname -r
4.9.11
```

You will work with **LTS kernel 4.9.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 an 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 calls 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 an user space application to provide it 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

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.

A Linux embedded 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 is also used for configuration.
- **/proc:** Represent the current state of the kernel.

Linux Boot Process

These are the main stages of a Linux embedded boot process:

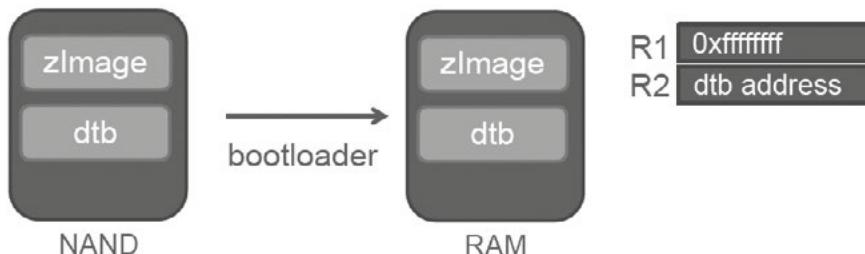
1. The boot process begins at POR (Power On Reset) where the hardware reset logic forces the ARM core to begin execution starting from the on-chip boot ROM. The boot ROM can support several devices (e.g, NOR flash, NAND Flash, SD/eMMC). In the i.MX7D processor the on-chip boot ROM sets up the DDR memory controller. The DDR technology is a potential key difference between different boards. If there is a difference in the DDR technology, the DDR initialization needs to be ported. In the i.MX7D the DDR initialization is coded in the DCD table, inside the boot header of the U-Boot image. The DCD (device configuration data) feature allows boot ROM code to obtain SoC configuration data from an external bootloader residing on the boot device. As an

example, DCD can be used to program the DDR controller for optimal settings improving the boot performance. After setting up the DDR controller the boot ROM loads the U-Boot image to external DDR and runs it.

The Microchip SAMA5D2 processor also embeds a boot ROM code. It is enabled depending on BMS (Boot Mode Select) pin state on reset. The ROM code scans the contents of different media like serial FLASH, NAND FLASH, SD/MMC Card and serial EEPROM. The Romcode will take the AT91Bootstrap from NAND FLASH and put it on internal SRAM. AT91Bootstrap is the 2nd level bootloader for Microchip SAMA5D2 SoC providing a set of algorithms to manage the hardware initialization such as clock speed configuration, PIO settings, and DRAM initialization. The AT91Bootstrap will take the U-Boot from NAND FLASH and put it on DDR RAM.

In another processors the second stage bootloader is known as the SPL.

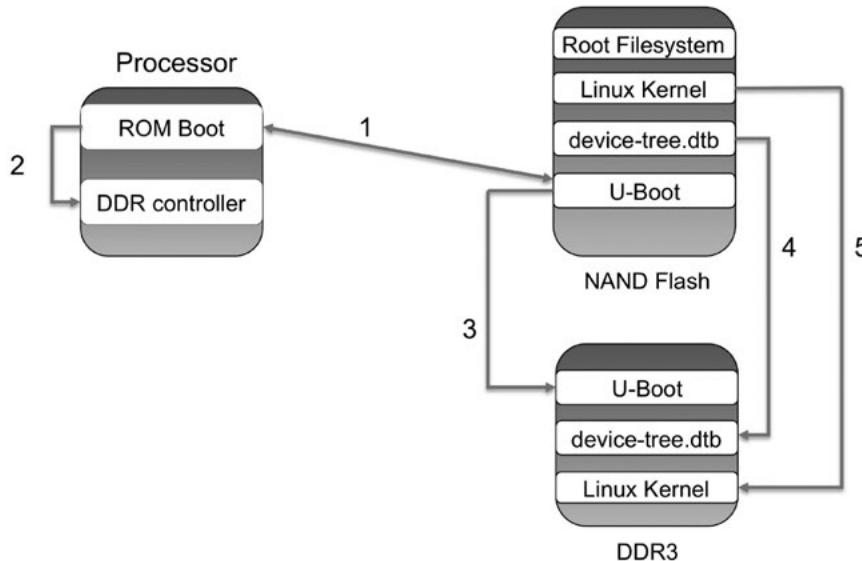
2. The U-Boot loads both the kernel image and the compiled device tree binary into RAM and passes the memory address of the device tree binary into the kernel as part of the launch.



3. The U-Boot jumps to the kernel code.
4. Kernel runs low level kernel initialization, enabling MMU and 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. 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 process init, by default, is /init for initramfs, and /sbin/init for a regular filesystem. The three init programs that you usually find in Linux embedded devices are **BusyBox init**, **System V init**, and **systemd**. If System V is used, then the process init reads its configuration file, /etc/inittab, and executes several scripts that perform final system initialization.

In the following image, you can see the Linux embedded boot process:



Building a Linux Embedded System

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, or 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 Linux embedded 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.

The "Yocto Project" will be your framework of choice to build the images for the Microchip SAMA5D2 and the NXP i.MX7D processors, whereas Debian will be the image used for the Broadcom BCM2837 processor.

Setting up Ethernet Communication

You are going to transfer files from the host to the target using the TFTP protocol:

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, like 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.

Building a Linux Embedded System for the NXP i.MX7D Processor

The i.MX7Dual family of processors features an advanced implementation of the Arm Cortex®-A7 core, which operates at speeds of up to 1.2 GHz, as well as the Arm Cortex-M4 core. The i.MX7Dual family supports multiple memory types including 16/32-bit DDR3L/LPDDR2/LPDDR3-1066, Quad SPI memory, NAND, eMMC, and NOR. Several high-speed connectivity connections include Gigabit Ethernet with AVB, PCIe, and USB. Both parallel and serial Display and Camera interfaces are provided, as well as a way to directly connect to the Electrophoretic Displays (EPD).

You can check all the info related to this family at <https://www.nxp.com/products/processors-and-microcontrollers/applications-processors/i.mx-applications-processors/i.mx-7-processors/i.mx-7dual-processors-heterogeneous-processing-with-dual-arm-cortex-a7-cores-and-cortex-m4-core:i.MX7D>.

For the development of the labs the **MCIMX7SABRE**: SABRE Board for Smart Devices Based on the i.MX7Dual Applications Processors will be used. The documentation of this board can be found at <https://www.nxp.com/support/developer-resources/hardware-development-tools/sabre-development-system/sabre-board-for-smart-devices-based-on-the-i.mx-7dual-applications-processors:MCIMX7SABRE>.

Although the MCIMX7SABRE is the board used to develop the i.MX7D drivers throughout this book, these drivers can be easily ported to another boards as the **ARROW IMX7 96 BOARD**. The documentation of this board can be found at <https://www.96boards.org/product/imx7-96/>.

Introduction

To get the Yocto Project expected behavior in a Linux Host Machine, the packages and utilities described below must be installed. An important consideration is the hard disk space required in the host machine. For example, when building on a machine running Ubuntu, the minimum hard disk space required is about 50 GB for the X11 backend. It is recommended that at least 120 GB is provided, which is enough to compile all backends together.

The next instructions have been tested on an Ubuntu 14.04 64-bit distribution.

Host Packages

A Yocto Project build requires that some packages be installed for the build that are documented under the Yocto Project. Essential Yocto Project host packages are:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
  build-essential chrpath socat libssl1.2-dev
```

Packages for an Ubuntu 14.04 host setup are:

```
$ sudo apt-get install libssl1.2-dev xterm sed cvs subversion coreutils \
  texi2html docbook-utils python-pysqlite2 help2man make gcc g++ \
  desktop-file-utils libgl1-mesa-dev libglu1-mesa-dev mercurial autoconf \
  automake groff curl lzop asciidoc u-boot-tools
```

Setting up the Repo Utility

The repo tool has been developed to make it easier to manage multiple Git repositories. Instead of downloading each repository separately, the repo tool can download all with one instruction. Download and install the tool by following the instructions below:

1. Create a directory for the tool. The example below creates a directory named bin in your home folder:

```
$ mkdir ~/bin
```

2. Download the tool:

```
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
```

3. Make the tool executable:

```
$ chmod a+x ~/bin/repo
```

4. Add the directory to the PATH variable. The line below could be added to your .bashrc file, so the path is available in each started shell/terminal:

```
$ export PATH=~/bin:$PATH
```

Yocto Project Setup and Image Building

The NXP Yocto Project BSP Release directory contains a "sources" directory, which contains the recipes used to build, one or more build directories, and a set of scripts used to set up the environment.

The recipes used to build the project come from both the community and NXP. The Yocto Project layers are downloaded and placed in the sources directory. This sets up the recipes that are used to build the project.

The following example shows how to download the NXP Yocto Project Community BSP recipe layers. For this example, a directory called "fsl-release-bsp" is created for the project:

```
~$ mkdir fsl-release-bsp
~$ cd fsl-release-bsp/
~/fsl-release-bsp$ git config --global user.name "Your Name"
~/fsl-release-bsp$ git config --global user.email "Your Email"
~/fsl-release-bsp$ git config --list
~/fsl-release-bsp$ repo init -u git://git.freescale.com/imx/fsl-arm-yocto-bsp.git \
-b imx-morty -m imx-4.9.11-1.0.0_ga.xml
~/fsl-release-bsp$ repo sync -j4
```

When this process is completed, the source code is checked out into the directory fsl-release-bsp/sources. You can perform periodic repo synchronization with the command repo sync to update to the latest code. If errors occur during repo initialization, try deleting the .repo directory and running the repo initialization command again.

There is a script, fsl-setup-release.sh, that simplifies the setup for i.MX machines. To use the script, the name of the specific machine to be built for needs to be specified as well as the desired graphical backend. The script sets up a directory and the configuration files for the specified machine and backend.

In the meta-fsl-bsp-release layer, i.MX provides new or updated machine configurations that overlay the meta-fsl-arm machine configurations. These files are copied into the meta-fsl-arm/conf/machine directory by the fsl-setup-release.sh script.

Before starting the build it must be initialized. In this step the build directory and local configuration files are created. A **distribution** must be selected when initializing the build. In the setting below the machine imx7dsabresd, the build directory build_imx7d and the fsl-imx-x11 distribution is selected:

```
~/fsl-release-bsp$ DISTRO=fsl-imx-x11 MACHINE=imx7dsabresd source fsl-setup-release.
sh -b build_imx7d
```

After doing this setting you are redirected to the build_imx7d directory:

```
~/fsl-release-bsp/build_imx7d$
```

If you are opening a new terminal, before starting the build, you must source the fsl-setup-release.sh script:

```
~/fsl-release-bsp$ source fsl-setup-release.sh -b build_imx7d/
```

A Yocto Project build can take considerable build resources both in time and disk usage, especially when building in multiple build directories. There are methods to optimize this, for example, using a shared sstate cache (caches the state of the build) and shared download directory (holds

the downloaded packages). These can be set to any location in the local.conf file under fsl-release-bsp/build-x11/conf directory by adding statements such as these:

```
DL_DIR="opt/freescale/yocto/imx/download"
SSTATE_DIR="opt/freescale/yocto/imx/sstate-cache"
```

Take also care of creating these folders in your host:

```
~$ sudo mkdir -p /opt/freescale/yocto/imx/download
~$ sudo mkdir -p /opt/freescale/yocto/imx/sstate-cache
```

These directories need to have appropriate permissions. The shared sstate helps when multiple build directories are set, each of which uses a shared cache to minimize the build time. A shared download directory minimizes the fetch time. Without these settings, Yocto Project defaults to the build directory for the sstate cache and downloads. When you want to start a build from a clean state, you need to remove both the sstate-cache directory and the tmp directory.

```
~$ sudo chmod -R a+rwx /opt/freescale/yocto/imx/download/
~$ sudo chmod -R a+rwx /opt/freescale/yocto/imx/sstate-cache/
```

Build now the Linux image:

```
~/fsl-release-bsp/build_imx7d$ bitbake fsl-image-validation-imx
```

When the build has finished the images will be available in the directory specified below. Please note that this directory will be different if you are using another build directory or machine configuration.

```
~$ ls fsl-release-bsp/build_imx7d/tmp/deploy/images/imx7dsabresd/
```

Finally, you are going to load the image onto a SD card. These are the instructions to program it using a laptop's built-in SD reader:

```
~/fsl-release-bsp/build_imx7d/tmp/deploy/images/imx7dsabresd$ dmesg | tail
~/fsl-release-bsp/build_imx7d/tmp/deploy/images/imx7dsabresd$ sudo umount /dev/mmcblk0p1
~/fsl-release-bsp/build_imx7d/tmp/deploy/images/imx7dsabresd$ sudo dd if=fsl-image-validation-imx-imx7dsabresd.sdcard of=/dev/mmcblk0 bs=1M && sync
```

If you are using an external USB SD reader use the following commands:

```
~/fsl-release-bsp/build_imx7d/tmp/deploy/images/imx7dsabresd$ dmesg | tail
~/fsl-release-bsp/build_imx7d/tmp/deploy/images/imx7dsabresd$ sudo umount /dev/sdX
~/fsl-release-bsp/build_imx7d/tmp/deploy/images/imx7dsabresd$ sudo dd if=fsl-image-validation-imx-imx7dsabresd.sdcard of=/dev/sdX bs=1M && sync
```

Here, /dev/sdX corresponds to the device node assigned to the SD card in your host system.

Working Outside of Yocto

You may find it more convenient to work on the kernel and develop drivers and applications outside of the Yocto Project build system. The **Yocto Project SDK** is going to help you with this task. The Yocto Project SDK is:

1. A cross-compile toolchain.
2. A combination of two sysroots:
 - One for the target: Contains headers and libraries for the target. Consistent with the generated image from which it is derived.
 - One for the host: Contains host specific tools. These tools ensure things are consistent and work as expected while building against the target sysroot.
3. An environment script to setup the necessary variables to make these work together.

There are several ways to build a SDK with the Yocto Project build system:

- Using bitbake meta-toolchain. This method requires you to still install the target sysroot by installing and extracting it separately.
- Using bitbake image -c populate_sdk. This method has significant advantages over the previous method because it results in a toolchain installer that contains the sysroot that matches your target root filesystem.

Remember, before using any bitbake command in a new terminal, you must source the build environment setup script:

```
~/fsl-release-bsp$ source fsl-setup-release.sh -b build_imx7d/  
~/fsl-release-bsp/build_imx7d$ bitbake -c populate_sdk fsl-image-validation-imx
```

When the bitbake command completes, the toolchain installer will be in tmp/deploy/sdk in the build directory. This toolchain installer contains the sysroot that matches your target root filesystem. The resulting toolchain and matching sysroot can be installed doing:

```
~/fsl-release-bsp/build_imx7d/tmp/deploy/sdk$ ./fsl-imx-x11-glibc-x86_64-fsl-image-validation-imx-cortexa7hf-neon-toolchain-4.9.11-1.0.0.sh
```

To develop applications on a host machine for a different target architecture, you need a cross-compiling toolchain. For this you will use the Yocto SDK which was pre-installed in the /opt/fsl-imx-x11/4.9.11-1.0.0/ directory. Let us explore the Yocto SDK. Type the following command on the host terminal:

```
~$ tree -L 3 /opt/fsl-imx-x11/  
└── 4.9.11-1.0.0  
    ├── environment-setup-cortexa7hf-neon-poky-linux-gnueabi  
    └── site-config-cortexa7hf-neon-poky-linux-gnueabi
```

```
└── sysroots
    ├── cortexa7hf-neon-poky-linux-gnueabi
    └── x86_64-pokysdk-linux
        └── version-cortexa7hf-neon-poky-linux-gnueabi
```

Here, the 4.9.11-1.0.0 folder contains scripts to export SDK environment variables. The sysroots directory contains SDK tools, libs, header files and two sub-directories, one for the host (x86_64) and one for the target (cortexa7hf). Note that you can find some information about the SDK just by reading suffixes:

- **cortexa7hf**: SDK, for Cortex-A7 hard float (with floating point unit support).
- **neon**: neon coprocessor support.
- **linux**: for the Linux operating system.
- **gnueabi**: gnu embedded application binary interface.

To setup the SDK, you have to source the environment script:

```
$ source /opt/fsl-imx-x11/4.9.11-1.0.0/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

This script will export several environment variables such us:

- **CC**: C compiler with target compilation options.
- **CFLAGS**: Additional C flags, used by the C compiler.
- **CXX**: C++ compiler.
- **CXXFLAGS**: Additional C++ flags, used by the CPP compiler.
- **LD**: Linker.
- **LDFLAGS**: Link flags, used by the linker.
- **GDB**: Debugger.
- **PATH**: SDK binaries added in standard command PATH.

You can look at the full set of environment variables sourced using the command below:

```
~$ export | more
```

The compiler is now in the current PATH:

```
~$ arm-poky-linux-gnueabi-gcc -version
arm-poky-linux-gnueabi-gcc (GCC) 6.2.0
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

\$CC provides target gcc options:

```
~$ echo $CC
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7
--sysroot=/opt/fsl-imx-x11/4.9.11-1.0.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi
```

- **arch**: armv7ve: compile for armv7ve architecture.
- **float-abi**: hard: application binary interface support hard float (fpu).
- **fpu**: neon: support ARM NEON coprocessor.
- **sysroot**: where libs and header files are located.

Now, a very simple application will be compiled to verify that your toolchain is properly installed. Create the application file using for example the **gedit** text editor:

```
~$ mkdir my_first_app
~$ cd my_first_app/
~/my_first_app$ gedit app.c
```

Add the code below:

```
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
}
```

If you try to cross compile like this you will get a fatal error:

```
~/my_first_app$ arm-poky-linux-gnueabi-gcc app.c -o app
app.c:1:19: fatal error: stdio.h: No such file or directory
 #include <stdio.h>
          ^
compilation terminated.
```

The reason is that the compiler has been configured to be generic to a wide range of ARM processors, and the fine tuning is done when you launch the compiler with the right set of C flags. You can compile app.c directly with the C compiler (\$CC):

```
~/my_first_app$ $CC app.c -o app
```

With the UNIX command "file" you can determine the file type (see: man file) and check the architecture and the linking method:

```
~/my_first_app$ file app
app: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.2.0,
BuildID[sha1]=7e2e3cf7c3647dce592ab5de92dac39cf4fb4f92, not stripped
```

Building the Linux Kernel

The kernel will be configured and built outside of the Yocto build system. Copy kernel sources from the Yocto tmp directory (that was generated after building your image with bitbake) to your own kernel directory:

```
~$ mkdir my-linux-imx
~$ cp -rpa ~/fsl-release-bsp/build_imx7d/tmp/work/imx7dsabresd-poky-linux-gnueabi/
linux-imx/4.9.11-r0/git/* ~/my-linux-imx/
~$ cd ~/my-linux-imx/
```

You can also download the kernel source code from the NXP kernel git:

```
~$ git clone http://git.freescale.com/git/cgit.cgi/imx/linux-imx.git \
-b imx_4.9.11_1.0.0_ga
```

Prior to compiling the Linux kernel, it is often a good idea to make sure that the kernel sources are clean and that there are no remnants left over from a previous build:

- **clean** - Remove most generated files but keep the config and enough build support to build external modules.
- **mrproper** - Remove all generated files + config + various backup files.
- **distclean** - mrproper + remove editor backup and patch files.

```
~/my-linux-imx$ make mrproper
```

It is often easiest to start with a base default configuration, and then customize it for your use case if needed. The **imx_v7_defconfig** located in arch/arm/configs will be used as a starting point:

```
~/my-linux-imx$ make ARCH=arm imx_v7_defconfig
```

When you want to customize the kernel configuration, the easiest way is to use the built-in kernel configuration systems. One of the most popular configuration systems is the **menuconfig** utility. Use a terminal from which the environment-setup-cortexa7hf-neon-poky-linux-gnueabi was not sourced. Use "cd /" to search in menuconfig:

```
~/my-linux-imx$ make ARCH=arm menuconfig
```

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

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

Device drivers >
[*] LED Support  --->
    <*>   LED Class Support
    -*-   LED Trigger support  --->
        <*>   LED Timer Trigger
        <*>   LED Heartbeat Trigger

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

```
-*- Industrial I/O buffering based on kfifo
<*> Enable IIO configuration via configfs
-*
- Enable triggered sampling support
<*> Enable software IIO 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
        <*> Userspace platform driver with generic irq and dynamic memory

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

Save the configuration and exit from menuconfig.

Once the kernel has been configured, it must be compiled to generate the bootable kernel image as well as any dynamic kernel modules that were selected. By default U-Boot expects **zImage** to be the type of kernel image used. Be sure that in the terminal used the environment-setup-cortexa7hf-neon-poky-linux-gnueabi script has been sourced before compiling the kernel:

```
~/my-linux-imx$ source /opt/fsl-imx-x11/4.9.11-1.0.0/environment-setup-cortexa7hf-
neon-poky-linux-gnueabi
~/my-linux-imx$ make -j4 zImage
```

Starting with Linux kernel version 3.8 each ARM board has an unique device tree binary file required by the kernel. Therefore, you will need to build and install the correct dtb for the target device. All device tree files are located under arch/arm/boot/dts/. To build an individual device tree file find the name of the dts file for the board you are using and replace the .dts extension with .dtb. The compiled device tree file will be located under arch/arm/boot/dts/. Then run the following command to compile an individual device tree file:

```
~/my-linux-imx$ make -j4 imx7d-sdb.dtb
```

To build all the device tree files:

```
~/my-linux-imx$ make -j4 dtbs
```

By default, the majority of the Linux drivers are not integrated into the kernel image (e.g., **zImage**). These drivers are built as dynamic modules. This will result in .ko (kernel object) files being placed in the kernel tree. These .ko files are the dynamic kernel modules. Whenever you make a change

to the kernel it is generally recommended that you rebuild your kernel modules and then reinstall them. Otherwise, the kernel modules may not load or run. The command to build these modules is:

```
~/my-linux-imx$ make -j4 modules
```

To compile the kernel image, modules, and all the device tree files in a single step:

```
~/my-linux-imx$ make -j4
```

Once the Linux kernel, dtb files and modules have been compiled they must be installed. In the case of the kernel image this can be installed by copying the zImage file to the location where it will be read from. The device tree binaries should also be copied to the same directory that the kernel image was copied to. You will read kernel and device tree files from a TFTP Server:

```
~/my-linux-imx$ cp /arch/arm/boot/zImage /var/lib/tftpboot/  
~/my-linux-imx$ cp /arch/arm/boot/dts/imx7d-sdb.dtb /var/lib/tftpboot/
```

During the development of the drivers for the i.MX7D and SAMA5D2 processors you will use both TFTP and NFS servers in your host system and only the U-Boot bootloader stored in the SD card will be needed. The bootloader will fetch the Linux kernel from the TFTP server and the kernel will mount the root filesystem from the NFS server. Changes to either the kernel or the root filesystem will be made available without the need to reprogram the SD.

Installing a TFTP Server

If you are not already running a TFTP server, follow the next steps to install and configure a TFTP server on your Ubuntu 14.04 host:

```
~$ sudo apt-get install tftpd-hpa
```

The tftpd-hpa configuration file is installed in /etc/default/tftpd-hpa. By default, it uses /var/lib/tftpboot as the root TFTP folder. Change the folder permissions to make it accessible to all users by using the following command:

```
~$ sudo chmod 1777 /var/lib/tftpboot/
```

Check the TFTP server status with netstat -a | grep tftp. If there is no result, the server is probably not started. By safety, stop the service then start it with: sudo service tftpd-hpa stop and sudo service tftpd-hpa start.

Installing a NFS Server

If you are not already running a NFS server, follow the next steps to install and configure one on your Ubuntu 14.04 host:

```
~$ sudo apt-get install nfs-kernel-server
```

The /nfsroot directory will be used as the root for the NFS server, so the target's root filesystem will be un-tared from the Yocto build directory, here:

```
~$ sudo mkdir -m 777 /nfsroot
~$ cd /nfsroot/
~/nfsroot$ sudo tar xvf ~/fsl-release-bsp/build_imx7d/tmp/deploy/images/
imx7dsabresd/fsl-image-validation-imx-imx7dsabresd.tar.bz2
```

Next, the NFS server will be configured to export the /nfsroot folder. Edit the /etc/exports file and add the next line of code:

```
/nfsroot/ *(rw,no_root_squash,async,no_subtree_check)
```

Then, restart the NFS server for the configuration changes to take effect:

```
~$ sudo service nfs-kernel-server restart
```

To install the kernel modules you use another make command similar to the others, but with an additional parameter which gives the base location where the modules should be installed. This command will create a directory tree from that location, such us lib/modules/<kernel version>, which will contain the dynamic modules corresponding to this version of the kernel. The base location should usually be the root of your target filesystem.

```
~/my-linux-imx$ source /opt/fsl-imx-x11/4.9.11-1.0.0/environment-setup-cortexa7hf-
neon-poky-linux-gnueabi
~/my-linux-imx$ sudo make ARCH=arm INSTALL_MOD_PATH=/nfsroot/ modules_install
```

Setting the U-Boot Environment Variables

Power ON the MCIMX7SABRE board. Launch and configure **minicom** in your host to see the booting of the system. Set the following configuration: "115.2 kbaud, 8 data bits, 1 stop bit, no parity". Make sure both hardware and software flow controls are disabled. Stop the U-Boot sequence by pressing any key.

To perform network booting set the following environment variables at the U-Boot prompt:

```
U-Boot > setenv serverip 10.0.0.1
U-Boot > setenv ipaddr 10.0.0.10
U-Boot > setenv image zImage
U-Boot > setenv fdt_file imx7d-sdb.dtb
U-Boot > setenv nfsroot /nfsroot
U-Boot > setenv ip_dyn no
```

```
U-Boot > setenv netargs 'setenv bootargs \
console=${console},${baudrate} ${smp} root=/dev/nfs rootwait \
rw ip=10.0.0.10:10.0.0.1:10.0.0.0:255.255.255.0:off:eth0:off \
nfsroot=${serverip}:${nfsroot},v3,tcp'
U-Boot > setenv bootcmd run netboot
U-Boot> saveenv
```

Reset your board; it should now boot from the network.

Building a Linux Embedded System for the Microchip SAMA5D2 Processor

The SAMA5D2 series is a high-performance, ultra-low-power ARM Cortex-A5 processor based MPU. The Cortex A5 processor runs up to 500MHz and features the ARM NEON SIMD engine a 128kB L2 cache and a floating point unit. It supports multiple memories, including latest-generation technologies such as DDR3, LPDDR3, and QSPI Flash. It integrates powerful peripherals for connectivity (EMAC, USB, dual CAN, up to 10 UARTs, etc.) and user interface applications (TFT LCD controller, PCAP and resistive touch controllers, touch controller, class D amplifier, audio PLL, CMOS sensor interface, etc.). The devices offer advanced security functions to protect customer code and secure external data transfers. These include ARM TrustZone, tamper detection, secure data storage, hardware encryption engine, on-the-fly decryption of code stored in external DDR or QSPI memory and a secure boot loader.

You can check all the info related to this family at

<http://www.microchip.com/design-centers/32-bit-mpus/microprocessors/sama5/sama5d2-series>.

For the development of the labs the **SAMA5D2B-XULT**: SAMA5D2 (Rev. B) Xplained Ultra Evaluation Kit will be used. The user guide of this board can be found at http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-44083-32-bit-Cortex-A5-Microprocessor-SAMA5D2-Rev.B-Xplained-Ultra_User-Guide.pdf.

Introduction

To get the Yocto Project expected behavior in a Linux Host Machine, the packages and utilities described below must be installed. An important consideration is the hard disk space required in the host machine. For example, when building on a machine running Ubuntu, the minimum hard disk space required is about 50 GB for the X11 backend. It is recommended that at least 120 GB is provided, which is enough to compile all backends together.

The instructions have been tested on an Ubuntu 14.04 64-bit distribution.

Host Packages

A Yocto Project build requires that some packages be installed for the build that are documented under the Yocto Project. Essential Yocto Project host packages are:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
  build-essential chrpath socat libssl1.2-dev
```

Packages for an Ubuntu 14.04 host setup are:

```
$ sudo apt-get install libssl1.2-dev xterm sed cvs subversion coreutils \
  texi2html docbook-utils python-pysqlite2 help2man make gcc g++ \
  desktop-file-utils libgl1-mesa-dev libglu1-mesa-dev mercurial autoconf \
  automake groff curl lzop asciidoc u-boot-tools
```

Yocto Project Setup and Image Building

The Yocto Project is a powerful building environment. It is build on top of several components including the famous OpenEmbedded build framework for embedded Linux. Poky is the reference system for building a whole embedded Linux distribution.

The support for the SAMA5 family is included in a particular Yocto layer: meta-atmel. The sources for this layer are hosted on Linux4SAM GitHub account: <https://github.com/linux4sam/meta-atmel>.

You can see below the step by step build procedure:

Create a directory:

```
~$ mkdir sama5d2_morty
~$ cd sama5d2_morty/
```

Clone yocto/poky git repository with the proper branch ready:

```
~/sama5d2_morty$ git clone git://git.yoctoproject.org/poky -b morty
```

Clone meta-openembedded git repository with the proper branch ready

```
~/sama5d2_morty$ git clone git://git.openembedded.org/meta-openembedded -b morty
```

Clone meta-qt5 git repository with the proper branch ready:

```
~/sama5d2_morty$ git clone git://code.qt.io/yocto/meta-qt5.git
~/sama5d2_morty$ cd meta-qt5/
~/sama5d2_morty/meta-qt5$ git checkout v5.9.1
~/sama5d2_morty$ cd ..
~/sama5d2_morty$
```

Clone meta-atmel layer with the proper branch ready:

```
~/sama5d2_morty$ git clone git://github.com/linux4sam/meta-atmel.git -b morty
```

Enter the poky directory to configure the build system and start the build process:

```
~/sama5d2_morty$ cd poky/  
~/sama5d2_morty/poky$
```

Initialize build directory:

```
~/sama5d2_morty/poky$ source oe-init-build-env
```

Add meta-atmel layer to bblayer configuration file:

```
~/sama5d2_morty/poky/build$ gedit conf/bblayers.conf
```

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf  
# changes incompatibly  
POKY_BBLAYERS_CONF_VERSION = "2"  
  
BBPATH = "${TOPDIR}"  
BBFILES ?= ""  
  
BSPDIR := "${@os.path.abspath(os.path.dirname(dgetVar('FILE', True)) +  
'../../../../../')}"  
  
BBLAYERS ?= " \  
 ${BSPDIR}/poky/meta \  
 ${BSPDIR}/poky/meta-poky \  
 ${BSPDIR}/poky/meta-yocto-bsp \  
 ${BSPDIR}/meta-atmel \  
 ${BSPDIR}/meta-openembedded/meta-oe \  
 ${BSPDIR}/meta-openembedded/meta-networking \  
 ${BSPDIR}/meta-openembedded/meta-python \  
 ${BSPDIR}/meta-openembedded/meta-ruby \  
 ${BSPDIR}/meta-openembedded/meta-multimedia \  
 ${BSPDIR}/meta-qt5 \  
 "  
  
BBLAYERS_NON_REMOVABLE ?= " \  
 ${BSPDIR}/poky/meta \  
 ${BSPDIR}/poky/meta-poky \  
 "
```

Edit local.conf to specify the machine, location of source archived, package type (rpm, deb or ipk).

Set MACHINE name to "sama5d2-xplained":

```
~/sama5d2_morty/poky/build$ gedit conf/local.conf
```

```
[...]  
[...]  
MACHINE ??= "sama5d2-xplained"  
[...]  
DL_DIR ?= "your_download_directory_path"
```

```
[...]
PACKAGE_CLASSES ?= "package_ipk"
[...]
USER_CLASSES ?= "buildstats image-mklibs"
```

To get better performance, use the "poky-atmel" distribution by also adding that line:

```
DISTRO = "poky-atmel"
```

Build demo image. Additional local.conf changes are needed for your QT demo image. You can add these two lines at the end of the file:

```
~/sama5d2_morty/poky/build$ gedit conf/local.conf
```

```
[...]
LICENSE_FLAGS_WHITELIST += "commercial"
SYSVINIT_ENABLED_GETTYS = ""

~/sama5d2_morty/poky/build$ bitbake atmel-qt5-demo-image
```

Enhancements are added on top of the official v4.9 Linux kernel tag where most of the Microchip SoC features are already supported. Note as well that Microchip re-integrate each and every stable kernel release on top of this Long Term Support (LTS) kernel revision. This means that each v4.9.x version is merged in Microchip branch. You will use the **linux4sam_5.7 tag** with integration of stable kernel updates up to **v4.9.52**. You can check further info at <https://www.at91.com/linux4sam/bin/view/Linux4SAM/LinuxKernel>.

You are going to create a SD demo image compiled from tag **linux4sam_5.7**. Go to https://www.at91.com/linux4sam/bin/view/Linux4SAM/DemoArchive5_7 and download **linux4sam-poky-sama5d2_xplained-5.7.img.bz2** yocto demo image.

To write the compressed image on the SD card, you will have to 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/>. Follow the steps of the Create a SD card with the demo section at <https://www.at91.com/linux4sam/bin/view/Linux4SAM/Sama5d2XplainedMainPage>.

Working Outside of Yocto

In this section, it will describe only the instructions to build the Yocto SDK for the SAMA5D2 processor. For further info about the meaning of these instructions check the previous Working Outside of Yocto section for the i.MX7D processor.

```
~/sama5d2_morty/poky/build$ bitbake -c populate_sdk atmel-qt5-demo-image
~/sama5d2_morty/poky/build$ cd tmp/deploy/sdk/
~/sama5d2_morty/poky/build/tmp/deploy/sdk$ ls
~/sama5d2_morty/poky/build/tmp/deploy/sdk$ ./poky-atmel-glibc-x86_64-atmel-qt5-demo-image-cortexa5hf-neon-toolchain-2.2.3.sh
```

```
Poky (Yocto Project Reference Distro) SDK installer version 2.2.3
=====
Enter target directory for SDK (default: /opt/poky-atmel/2.2.3):
You are about to install the SDK to "/opt/poky-atmel/2.2.3". Proceed[Y/n]? y

Extracting    SDK.....done
.....done
Setting it up...done
SDK has been successfully set up and is ready to be used.
```

Building the Linux Kernel

In this section, it will be described only the instructions to build the Linux kernel for the SAMA5D2 processor. For further info about the meaning of these instructions check the previous Building the Linux Kernel section for the i.MX7D processor.

Copy the kernel sources from your Yocto build to a new created folder:

```
~$ mkdir my-linux-sam
~$ cp -rpa ~/sama5d2_morty/poky/build/tmp/work/sama5d2_xplained-poky-linux-gnueabi/
linux-at91/4.9+gitAUTOINC+973820d8c6-r0/git/* ~/my-linux-sam/
```

You can also download the kernel sources from the Microchip git:

```
~$ git clone git://github.com/linux4sam/linux-at91.git
~$ cd linux-at91/
~/linux-at91$ git branch -r
~/linux-at91$ git checkout origin/linux-4.9-at91 -b linux-4.9-at91
~/linux-at91$ git checkout linux4sam 5.7
```

Compile the kernel image, modules, and all the device tree files:

```
~/linux-at91$ make mrproper  
~/linux-at91$ make ARCH=arm sama5_defconfig  
~/linux-at91$ make ARCH=arm menuconfig
```

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

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

Device drivers >
[*] LED Support --->
    <*>    LED Class Support
    -*-    LED Trigger support --->
            <*>    LED Timer Trigger
            <*>    LED Heartbeat Trigger

Device drivers >
```

```
<*> Industrial I/O support --->
    -*- Enable buffer support within IIO
    -*- Industrial I/O buffering based on kfifo
    <*> Enable IIO configuration via configfs
    -*- Enable triggered sampling support
    <*> Enable software IIO 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
        <*> Userspace platform driver with generic irq and dynamic memory

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

Save the configuration and exit from menuconfig.

Source the toolchain script and compile kernel, device tree files and modules in a single step:

```
~/my-linux-sam$ source /opt/poky-atmel/2.2.3/environment-setup-cortexa5hf-neon-poky-
linux-gnueabi
~/my-linux-sam$ make -j4
```

Once the Linux kernel, dtb files and modules have been compiled they must be installed. In the case of the kernel image this can be installed by copying the zImage file to the location where it will be read from. The device tree binaries should also be copied to the same directory that the kernel image was copied to. You will read kernel and device tree files from a TFTP Server:

```
~/my-linux-sam$ cp /arch/arm/boot/zImage /var/lib/tftpboot/
~/my-linux-sam$ cp /arch/arm/boot/dts/at91-sama5d2_xplained.dtb /var/lib/tftpboot/
```

During the development of the driver labs you will use both TFTP and NFS servers in your host system and only the U-Boot bootloader stored in the SD card will be needed. The bootloader will fetch the Linux kernel from the TFTP server and the kernel will mount the root filesystem from the NFS server. Changes to either the kernel or the root filesystem will be made available without the need to reprogram the SD.

Installing a TFTP Server

Follow the next steps to install and configure a TFTP server on your Ubuntu 14.04 host:

```
~$ sudo apt-get install tftpd-hpa
```

Change the folder permissions to make it accessible to all users by using the following command:

```
~$ sudo chmod 1777 /var/lib/tftpboot/
```

Check the TFTP server status with netstat -a | grep tftp. If there is no result, the server is probably not started. By safety, stop the service, then start it with: sudo service tftpd-hpa stop and sudo service tftpd-hpa start.

Installing a NFS Server

Follow the next steps to install and configure one on your Ubuntu 14.04 host:

```
~$ sudo apt-get install nfs-kernel-server
```

The /nfssama5d2 directory will be used as the root for the NFS server, so the target's root filesystem will be un-tared from the Yocto build directory, here:

```
~$ sudo mkdir -m 777 /nfssama5d2
~$ cd /nfssama5d2/
~/nfssama5d2$ sudo tar xfvp ~/sama5d2_morty/poky/build/tmp/deploy/images/sama5d2-xplained/atmel-qt5-demo-image-sama5d2-xplained.tar.gz
```

Next, the NFS server will be configured to export the /nfssama5d2 folder. Edit the /etc/exports file and add the next line of code:

```
/nfssama5d2/ *(rw,no_root_squash,async,no_subtree_check)
```

Then restart the NFS server for the configuration changes to take effect:

```
~$ sudo service nfs-kernel-server restart
```

To install the kernel modules you use another make command similar to the others, but with an additional parameter, which gives the base location where the modules should be installed. This command will create a directory tree from that location, such us lib/modules/<kernel version>, which will contain the dynamic modules corresponding to this version of the kernel. The base location should usually be the root of your target filesystem.

```
~/my-linux-sam$ source /opt/poky-atmel/2.2.3/environment-setup-cortexa5hf-neon-poky-linux-gnueabi
~/my-linux-sam$ sudo make ARCH=arm INSTALL_MOD_PATH=/nfssama5d2/ modules_install
```

Setting the U-Boot Environment Variables

Power ON the SAMA5D2B-XULT board. Launch and configure **minicom** in your host to see the booting of the system. Set the following configuration: "115.2 kbaud, 8 data bits, 1 stop bit, no parity". Make sure both hardware and software flow controls are disabled. Stop the U-Boot sequence by pressing any key.

To perform network booting set the following environment variables at the U-Boot prompt:

```
U-Boot > setenv serverip 10.0.0.1
U-Boot > setenv ipaddr 10.0.0.10
U-Boot > setenv nfsroot /nfssama5d2
U-Boot > setenv ip_dyn no
U-Boot > setenv bootargs console=ttyS0,115200 root=/dev/nfs rootwait \
    rw ip=10.0.0.10:10.0.0.1:10.0.0.0:255.255.255.0:off:eth0:off \
    nfsroot=${serverip}:${nfsroot},v3,tcp
U-Boot > setenv bootcmd 'tftp 0x21000000 zImage; tftp 0x22000000 at91-sama5d2_\
    xplained.dtb; bootz 0x21000000 - 0x22000000'
U-Boot > saveenv
```

Reset your board; it should now boot from the network.

Building a Linux Embedded System for the Broadcom BCM2837 Processor

The BCM2837 processor is the Broadcom chip used in the Raspberry Pi 3, and in later models of the Raspberry Pi 2. The underlying architecture of the BCM2837 is identical to the BCM2836. The only significant difference is the replacement of the ARMv7 quad core cluster with a quad-core ARM Cortex A53 (ARMv8) cluster.

The ARM cores run at 1.2GHz, making the device about 50% faster than the Raspberry Pi 2. The VideoCore IV runs at 400MHz. You can see documentation of BCM2836 at <https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md> and BCM2835 at <https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/README.md>.

For the development of the labs the **Raspberry Pi 3 Model B**: Single-board computer with wireless LAN and Bluetooth connectivity will be used. You can see more info of this board at <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>.

Raspbian

Raspbian is the recommended operating system for normal use on a Raspberry Pi. Raspbian is a free operating system based on Debian, optimised for the Raspberry Pi hardware. Raspbian comes with over 35,000 packages: precompiled software bundled in a nice format for easy installation on

your Raspberry Pi. Raspbian 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 in a SD a **Raspbian_lite** image based on **kernel 4.9.y**. Go to http://downloads.raspberrypi.org/raspbian_lite/images/ and download 2017-09-07-raspbian-stretch-lite.zip image included in the folder raspbian_lite-2017-09-08/.

To write the compressed image on the 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/>.

Follow the steps of the Writing an image to the SD card section at <https://www.raspberrypi.org/documentation/installation/installing-images/README.md>.

Building the Linux Kernel

There are two main methods for building the kernel. You can build locally on a 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.

First install Git and the build dependencies:

```
~$ sudo apt-get install git bc
```

Next get the sources:

```
~$ git clone --depth=1 -b rpi-4.9.y https://github.com/raspberrypi/linux
~$ cd linux/
```

Download the toolchain to the home folder:

```
~$ git clone https://github.com/raspberrypi/tools ~/tools
~$ export PATH=~/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin:$PATH
~$ export TOOLCHAIN=~/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/
~$ export CROSS_COMPILE=arm-linux-gnueabihf-
~$ export ARCH=arm
```

Compile the kernel, modules and device tree files:

```
~$ cd linux/
~/linux$ make mrproper
~/linux$ KERNEL=kernel7
~/linux$ make ARCH=arm bcm2709_defconfig
~/linux$ make ARCH=arm menuconfig
```

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

```
Device drivers >
[*] SPI support --->
    <*> BCM2835 SPI controller
    <*> User mode SPI device driver support

Device drivers >
I2C support --->
    I2C Hardware Bus support --->
        <*> Broadcom BCM2835 I2C controller

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

Device drivers >
[*] LED Support --->
    <*> LED Class Support
    -*- LED Trigger support --->
        <*> LED Timer Trigger
        <*> LED Heartbeat Trigger

Device drivers >
<*> Industrial I/O support --->
    -*- Enable buffer support within IIO
    -*- Industrial I/O buffering based on kfifo
    <*> Enable IIO configuration via configfs
    -*- Enable triggered sampling support
    <*> Enable software IIO 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
    <*> Userspace platform driver with generic irq and dynamic memory

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

Save the configuration and exit from menuconfig.

Compile kernel, device tree files and modules in a single step:

```
~/linux$ make -j4
```

Having built the kernel, you need to copy it onto your Raspberry Pi and install the modules; insert a uSD into a SD card reader:

```
~$ lsblk
~$ mkdir ~/mnt
~$ mkdir ~/mnt/fat32
~$ mkdir ~/mnt/ext4
~$ sudo mount /dev/mmcblk0p1 ~/mnt/fat32
~$ sudo mount /dev/mmcblk0p2 ~/mnt/ext4
~$ ls -l ~/mnt/fat32/ /* see the files in the fat32 partition, check that config.txt
is included */
```

Update the config.txt file adding the next values:

```
~$ cd mnt/fat32/
~/mnt/fat32$ sudo gedit config.txt

dtoverlay=i2c_arm
dtoverlay=spi
dtoverlay=spi0-cs
# Enable UART
enable_uart=1
kernel=kernel-rpi.img
device_tree=bcm2710-rpi-3-b.dtb
```

Update kernel, device tree files and modules:

```
~/linux$ sudo cp arch/arm/boot/zImage ~/mnt/fat32/kernel-rpi.img
~/linux$ sudo cp arch/arm/boot/dts/*.dtb ~/mnt/fat32/
~/linux$ sudo cp arch/arm/boot/dts/overlays/*.dtb* ~/mnt/fat32/overlays/
~/linux$ sudo cp arch/arm/boot/dts/overlays/README ~/mnt/fat32/overlays/
~/linux$ sudo make ARCH=arm INSTALL_MOD_PATH=~/mnt/ext4 modules_install
~$ sudo umount ~/mnt/fat32
~$ sudo umount ~/mnt/ext4
```

Extract the uSD from the SD reader and insert it in the the Raspberry Pi 3 Model B board. Power ON the board. Launch and configure **minicom** in your host to see the booting of the system. Set the following configuration: "115.2 kbaud, 8 data bits, 1 stop bit, no parity". Make sure both hardware and software flow controls are disabled.

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 your Raspberry Pi is properly set up and connected:

```
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:

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

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

```
pi@raspberrypi:~$ sudo passwd root /* set for instance password to "pi" */
```

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
```

Create a very simple application that will be compiled to verify that your toolchain is properly installed. Create the application file using for example the **gedit** text editor:

```
~$ mkdir my_first_app
~$ cd my_first_app/
~/my_first_app$ gedit app.c
```

Add the code below:

```
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
}
```

Setup the toolchain updating the \$PATH environment variables:

```
~$ export PATH=~/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin:$PATH
~$ export TOOLCHAIN=~/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/
~$ export CROSS_COMPILE=arm-linux-gnueabihf-
~$ export ARCH=arm
```

You can compile app.c directly with the C compiler:

```
~/my_first_app$ arm-linux-gnueabihf-gcc app.c -o app
```

scp is a command for sending files over SSH. This means you can copy files between computers, say from your Raspberry Pi to your desktop or laptop, or vice-versa. You can copy now the file app

from your computer to the pi user's home folder of your Raspberry Pi at the IP address 10.0.0.10 with the following command:

```
~$ scp app pi@10.0.0.10: /* enter "raspberry" password in the host */
pi@raspberrypi:~$ ls /* see the app application in you raspberry */
pi@raspberrypi:~$ ./app /* execute the application in your raspberry */
```

or

```
~$ scp app root@10.0.0.10: /* enter "pi" password that was set when you enabled your
root account */
```

You have first to log into your pi as the root user to use in the host the scp app root@10.0.0.10: command:

```
pi@raspberrypi:~$ su
Password:pi
root@raspberrypi:/home/pi# cd /root/
root@raspberrypi:~$ ls /* see the app application */
root@raspberrypi:~$ ./app /* execute the application */
```

Every time you boot your Raspberry PI you have to execute the next instructions to be able to transfer files via scp:

```
pi@raspberrypi:~$ sudo ifconfig eth0 10.0.0.10 netmask 255.255.255.0
pi@raspberrypi:~# sudo /etc/init.d/ssh restart
```

On the Raspberry PI board (the target), configure the eth0 interface with IP address 10.0.0.10 by editing the /etc/network/interfaces file. Open the file using the nano editor:

```
pi@raspberrypi:~$ sudo nano /etc/network/interfaces
```

Add the following lines:

```
auto eth0
iface eth0 inet static
    address 10.0.0.10
    netmask 255.255.255.0
```

After editing the file type "Ctrl+x", then type "yes" and press "enter" to exit. Now, you only have to restart the ssh service the next time you boot your target.

Working with Eclipse

Eclipse provides IDEs and platforms for nearly every language and architecture. They are well known for the Java IDE, C/C++, JavaScript and PHP IDEs built on extensible platforms for creating desktop, Web and cloud IDEs. These platforms deliver the most extensive collection of add-on tools available for software developers. Eclipse will help navigate within the Linux kernel sources instead of using Linux terminal commands.

You will work with the **Neon Release** to develop the drivers. You can download the Eclipse environment at <https://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/neonr>. Go to the Neon Packages Release and download the Eclipse IDE for C/C++ developers (Linux 32-bit or 64-bit depending on your Linux host system).

Installing Eclipse is simple. Just download a proper version from the web page and untar it. The system must have the proper version of Java SDK installed. Ubuntu allows multiple packages providing a Java Virtual Machine to be installed. For Neon is 8.

```
~$ sudo apt-get install openjdk-8-jdk
```

Untar the 64-bit Neon Eclipse download:

```
~/eclipse_neon$ tar -xf eclipse-cpp-neon-3-linux-gtk-x86_64.tar.gz
```

Make sure that you run Eclipse from a shell, that was configured for Yocto SDK. This makes configuring the toolchain (PATH) much easier. It will be used the SAMA5D2 kernel sources and its Yocto SDK cross-compile toolchain to show how to configure Eclipse for the development of the drivers.

```
~/eclipse_neon$ source /opt/poky-atmel/2.2.3/environment-setup-cortexa5hf-neon-poky-linux-gnueabi
~/eclipse_neon$ cd eclipse/
~$ ./eclipse & /* launch eclipse */
```

Select the workspace path, for example:

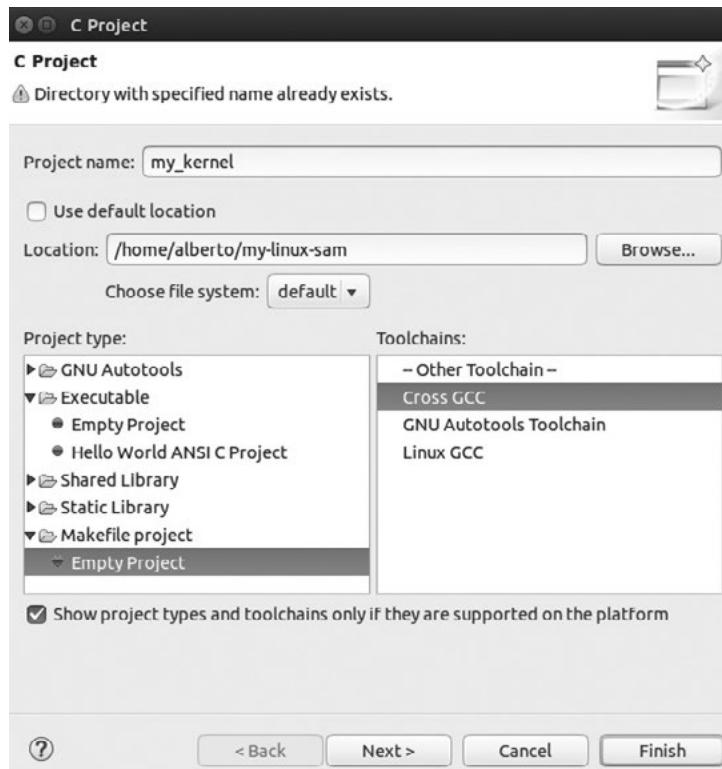
```
/home/<user>/workspace_neon
```

Eclipse Configuration for Working with Kernel Sources

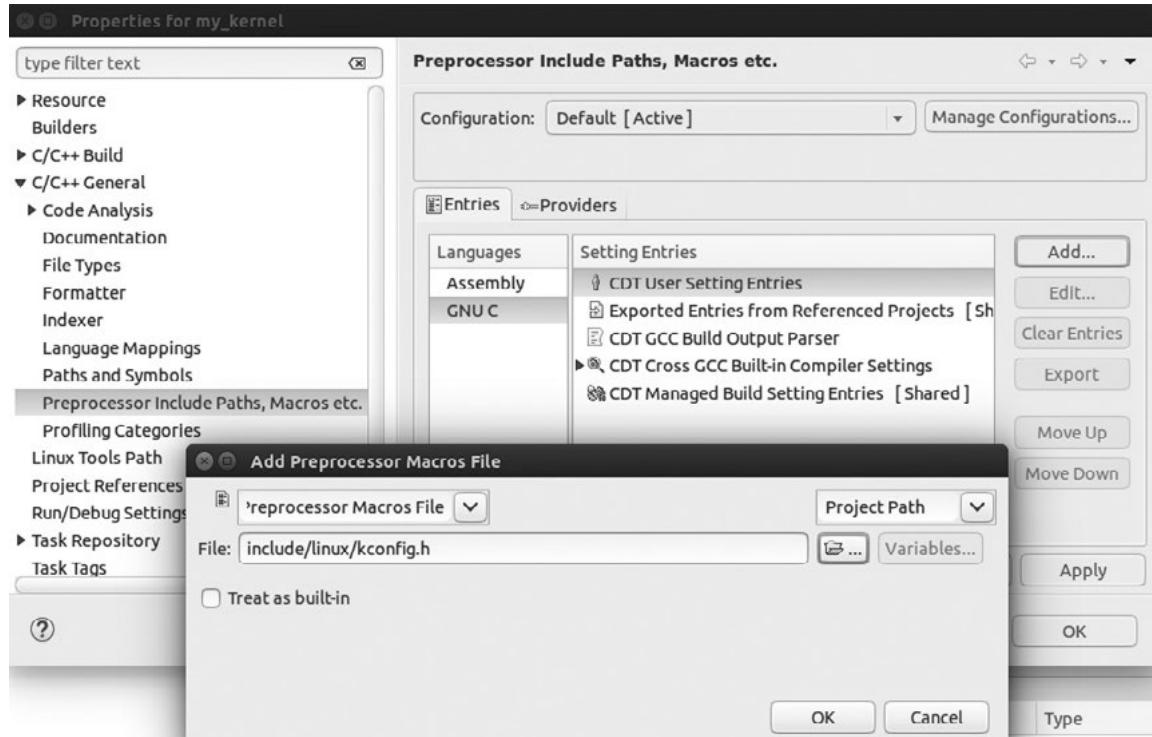
Before configuring Eclipse, the kernel must be configured and built (you have already configured and built your kernel in previous sections), to define `CONFIG_*` and generate `autoconf.h`. Some files essential for indexer are generated while configuring and building for given architecture.

1. Start up Eclipse and then create a new C project:

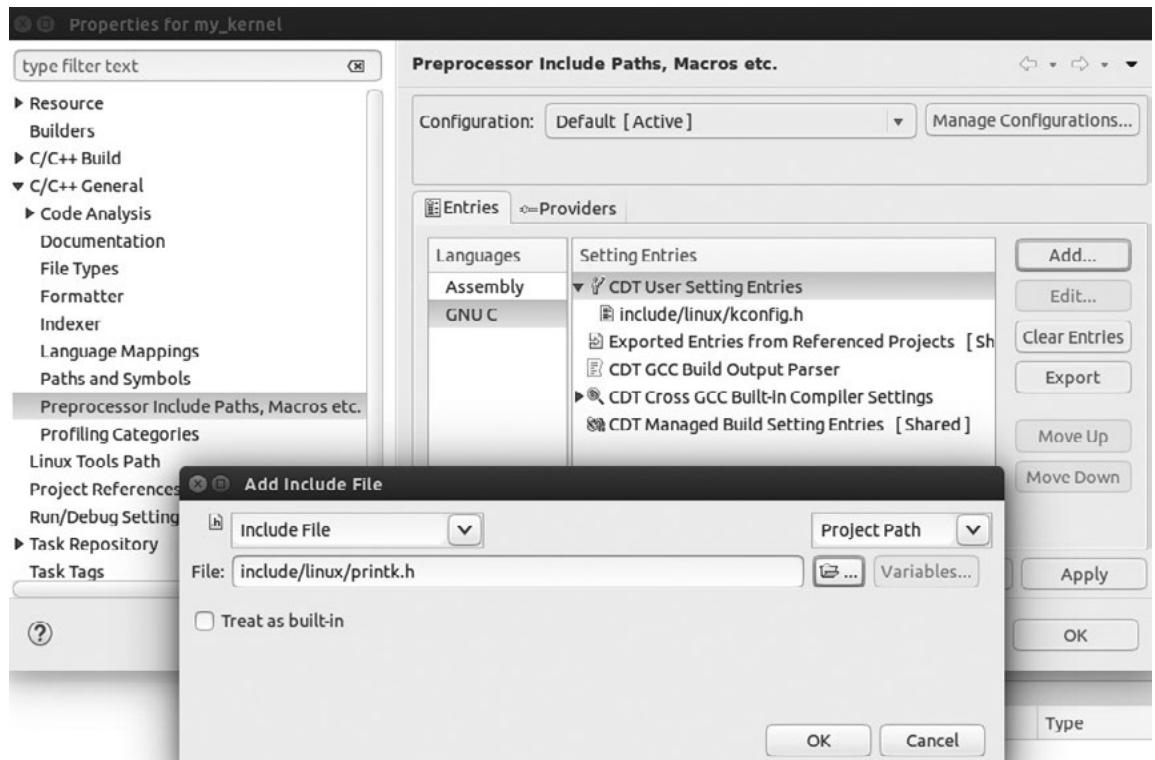
- File->New->C Project.
- Set project name (it is just for Eclipse), for example provide the name my_kernel.
- Do not use the default location (workspace). Point to the directory containing the kernel sources: /home/<user>/my-linux-sam.
- The kernel is built using make, so choose a Makefile project.
- This is using cross-toolchain.
- Next->Finish.



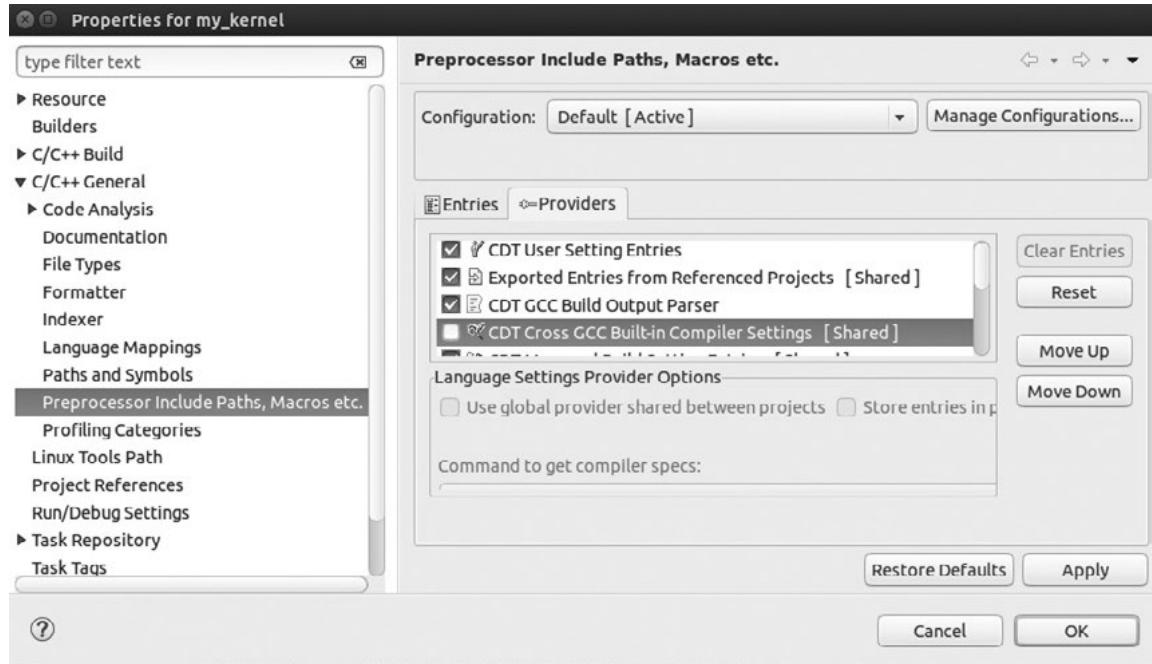
2. After creating the project modify its settings. Go to **Project->Properties** and open the **C/C++ General** selection on the left. Click on **Preprocessor Include Paths, Macros etc..**, then click on the **Entries** tab and select **GNU C** in the Languages list. Select **CDT User Setting Entries** in the **Setting Entries** list. Add include/linux/kconfig.h file as the Preprocessor macros file:



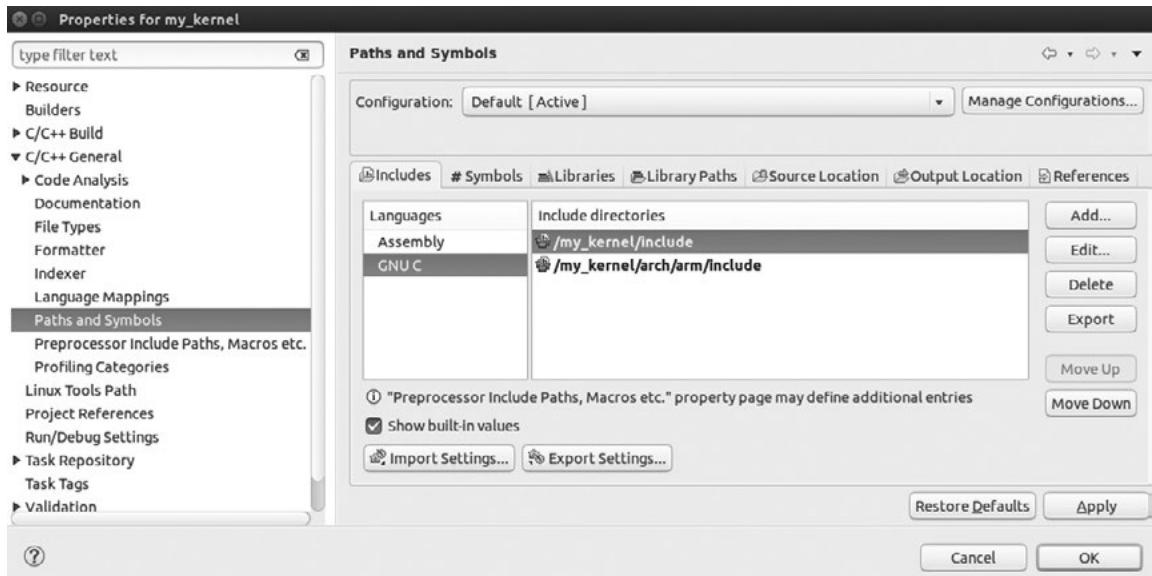
3. Add the file containing the `printk()` macro definition. Otherwise, indexer will not find it (as it is used in many places):



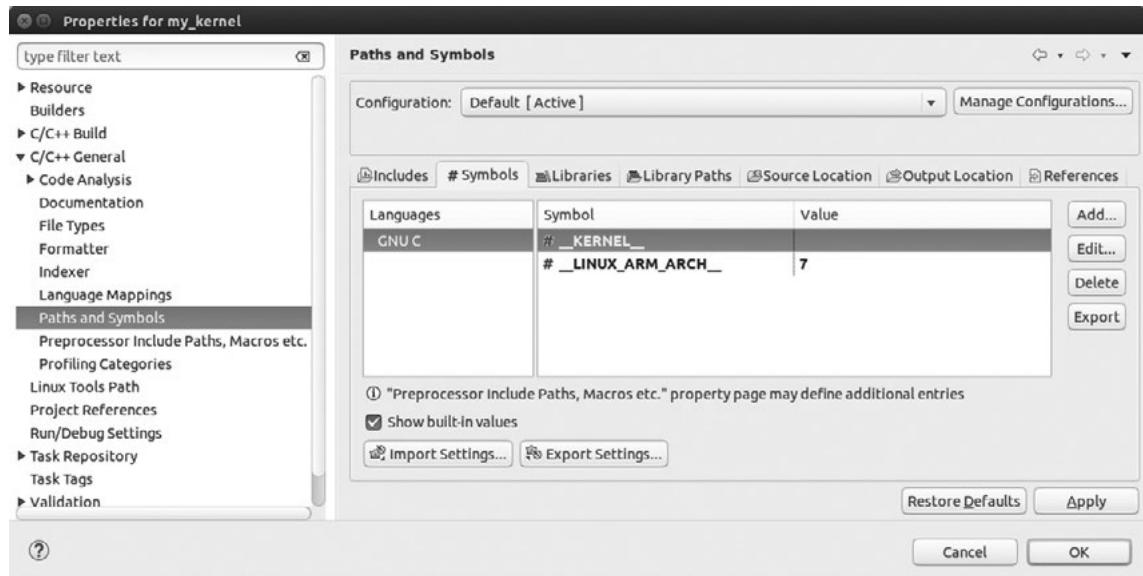
4. Click on the **Providers** tab and turn off the looking to default search path provided via cross-compiler feature. Indexer should use in-kernel headers instead of those from the toolchain sysroot path:



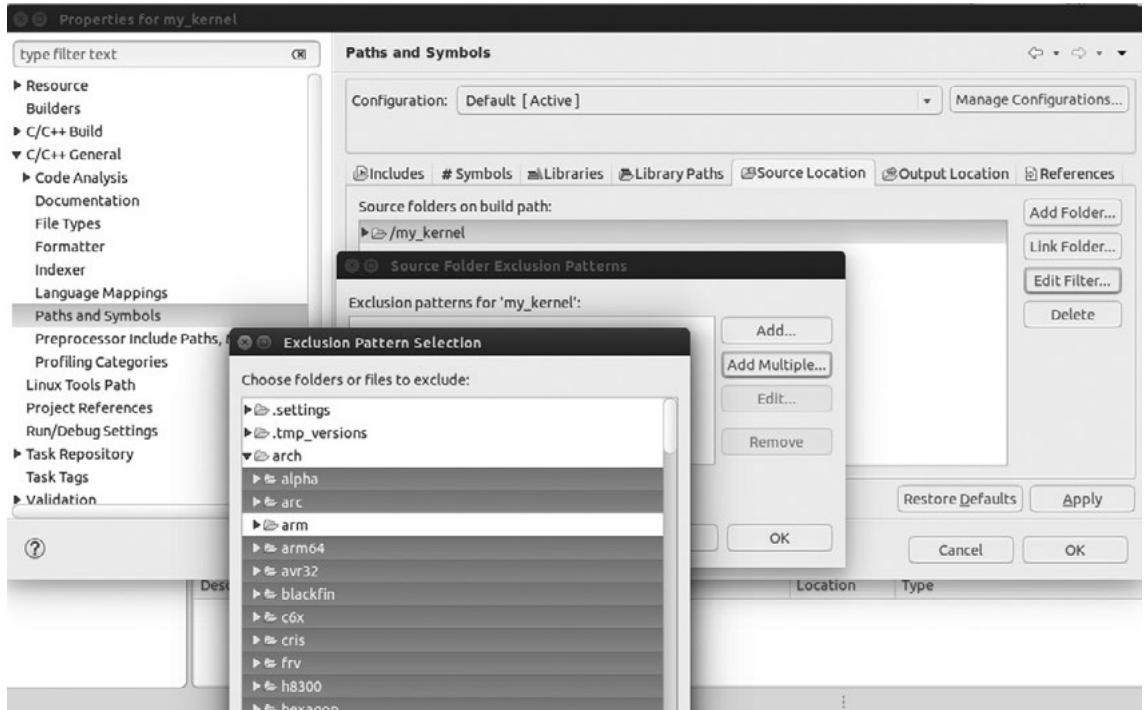
5. The in-kernel include paths have to be added. The kernel does not use any external libraries. In the **C/C++ General** selection click on **Paths and Symbols**, then click on the **Includes** tab and select **GNU C** in the Languages list. Finally, click on the **Add** tab and add the include paths:
 - /my_kernel/include: generic ones
 - /my_kernel/arch/arm/include: architecture specific



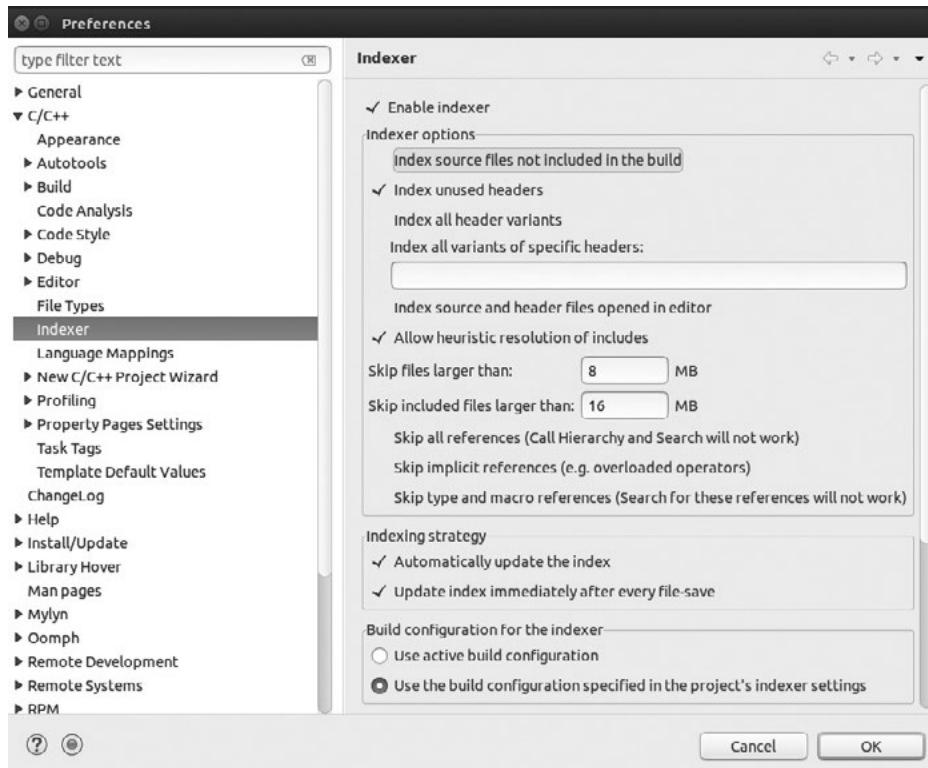
6. Add the `__KERNEL__` (Warning: two underscores KERNEL two underscores) and `__LINUX_ARM_ARCH__` (Value = 7) symbols. The `__KERNEL__` symbol is used in many places to distinguish between parts of files containing user space API (not defined) from a kernel API (defined); the symbol is set in Makefile during build. In **Paths and Symbols** click on the **Symbols** tab, then click on the **Add** tab to add the symbols.



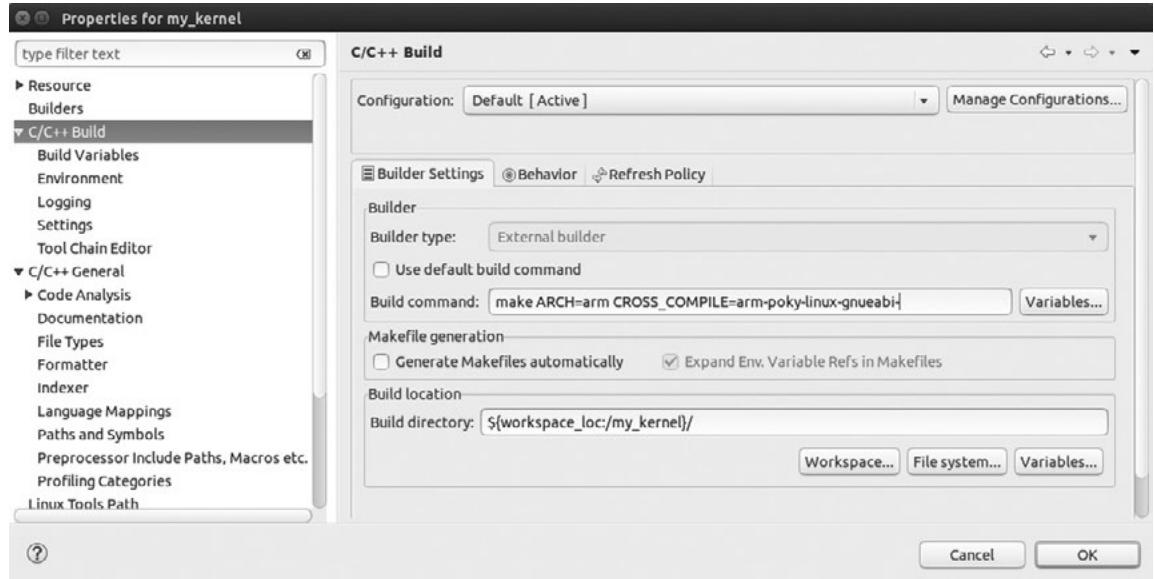
7. The arch directory, contains architecture-specific files. The filter has to be defined to exclude all subdirectories containing code for architectures not used in current configuration. In **Paths and Symbols** click on the **Source Location** tab, then click on the **Edit Filter** tab and in the new window appearing click on the **Add Multiple** tab selecting all subdirectories of arch/ but arch/arm. Also filter out the tools directory. It contains conflicting include files.



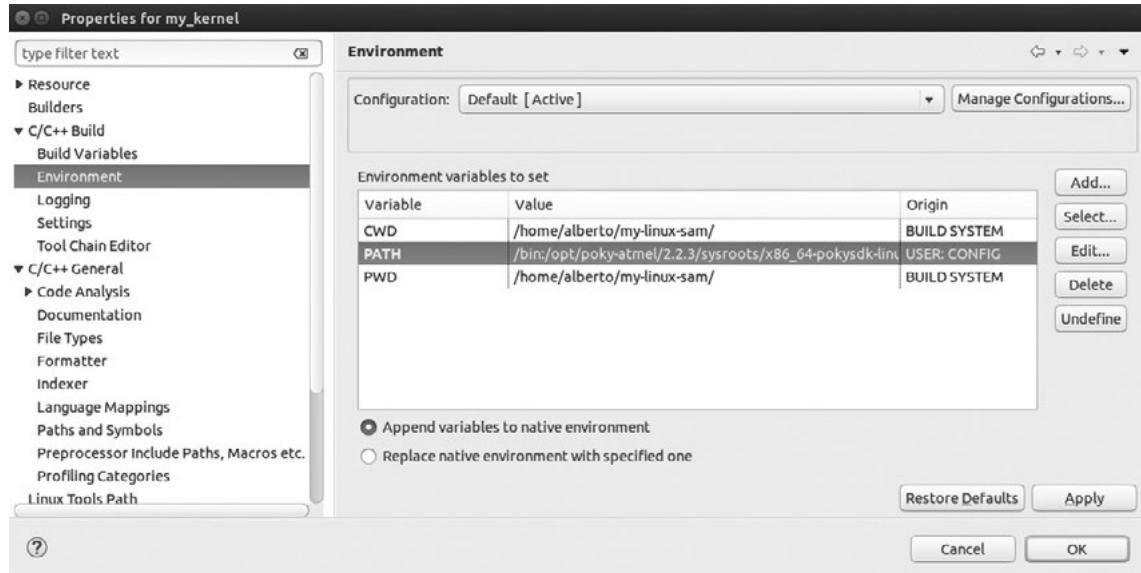
8. Turn off indexing files not included in the build. This will limit the number of files to index:



9. Set the make parameters needed for building the kernel:



10. Use the sub-page Environment to set the proper path. In this way you do not have to source the environment each time you open eclipse. Click on the variable PATH (check to make sure it is highlighted):



11. Click OK in the settings window to save project options and run the indexer. To make sure everything will be indexed properly force a rebuild. It can be done from the project manager context menu:

Project->Index->Rebuild

Indexing will take a few minutes. About 700-800MB of data will be created in the workspace directory. About 20000 files will be parsed (depending on the kernel version).

If Eclipse is configured in the properly, it should expand the symbols under the cursor. This configuration is not ideal. Some macros and symbols are highlighted as warnings or errors. Eclipse Indexer does not behave in the same way as GCC. The goal is to make working with kernel sources easier, rather than removing all warnings.

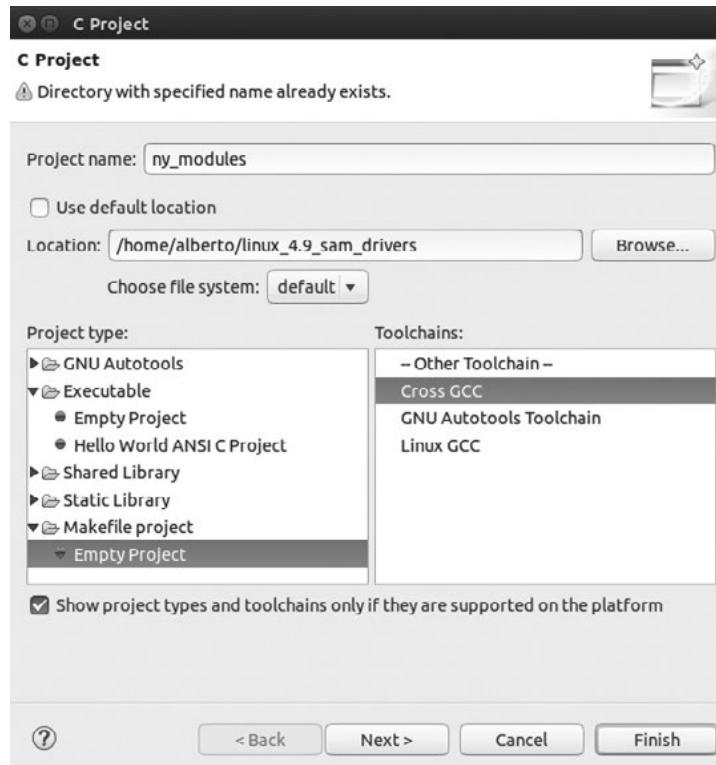
Now, Eclipse can be used to build the kernel. If you change any of your CONFIG_* settings, in order for Eclipse to recognize those changes, you may need to do a "build" from within Eclipse. Note, this does not mean to re-build the index; this means to build the kernel, by having Eclipse invoke make (this is normally bound to the Ctrl-B key in Eclipse). Eclipse should automatically detect changes to include/generated/autoconf.h, reread the compilation #defines it uses, and reindex.

Eclipse Configuration for Developing Linux Drivers

You are going to create a project in Eclipse to write, build and deploy your drivers. The configuration looks the same, as in the case of the kernel sources.

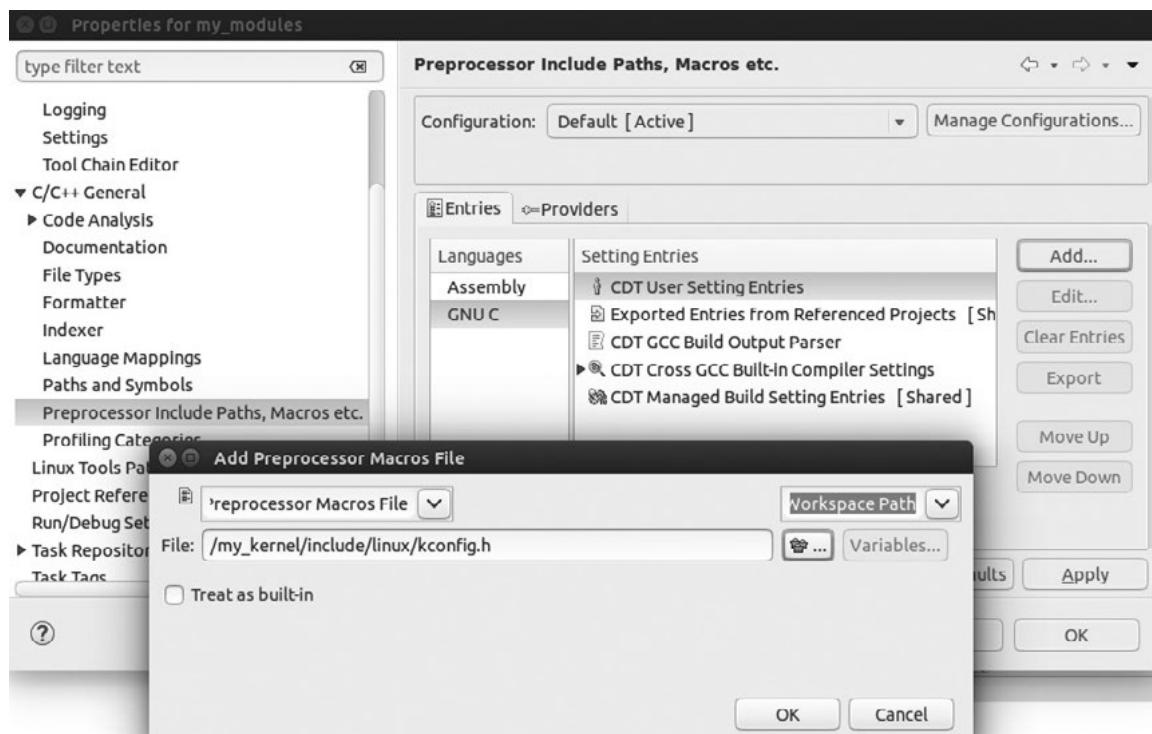
1. Create a new Makefile C project. Point to the location where you will store your driver sources. In the next figure, you can see that the path is /home/<user>/Linux_4.9_sam_drivers. Follow the steps mentioned below:

- File->New->C Project.
- Set project name (it is just for Eclipse), for example provide the name my_modules.
- Do not use default location (workspace). Point to the directory containing kernel modules sources.
- kernel is build using make, so choose a Makefile project.
- Use a cross-toolchain.
- Next->Finish.

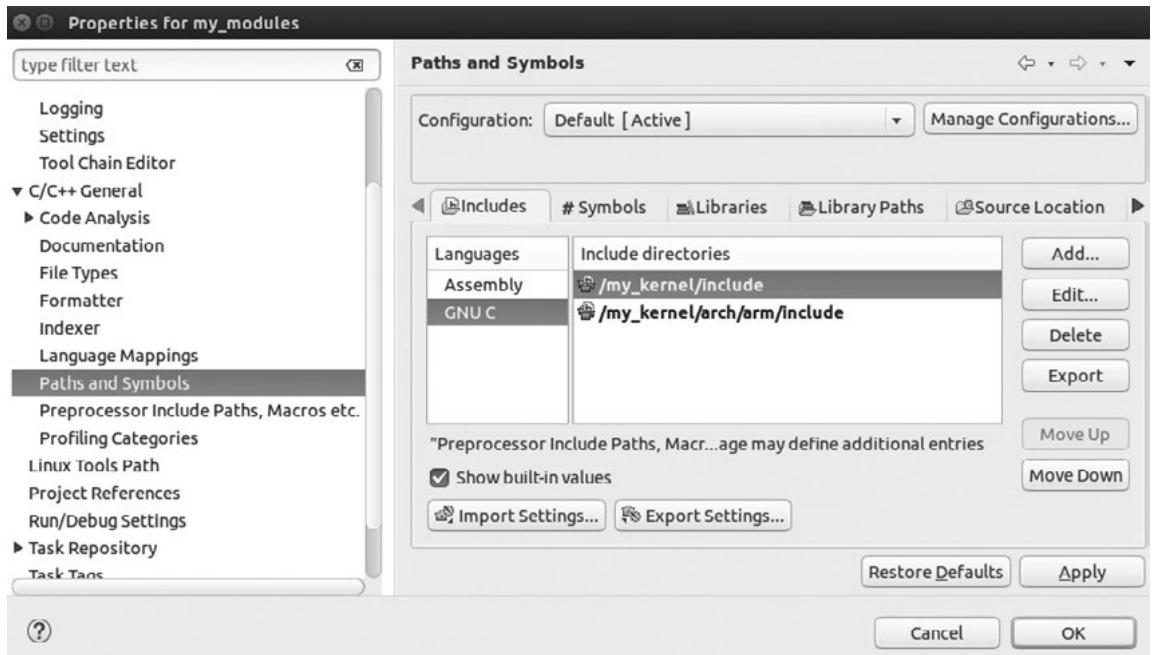
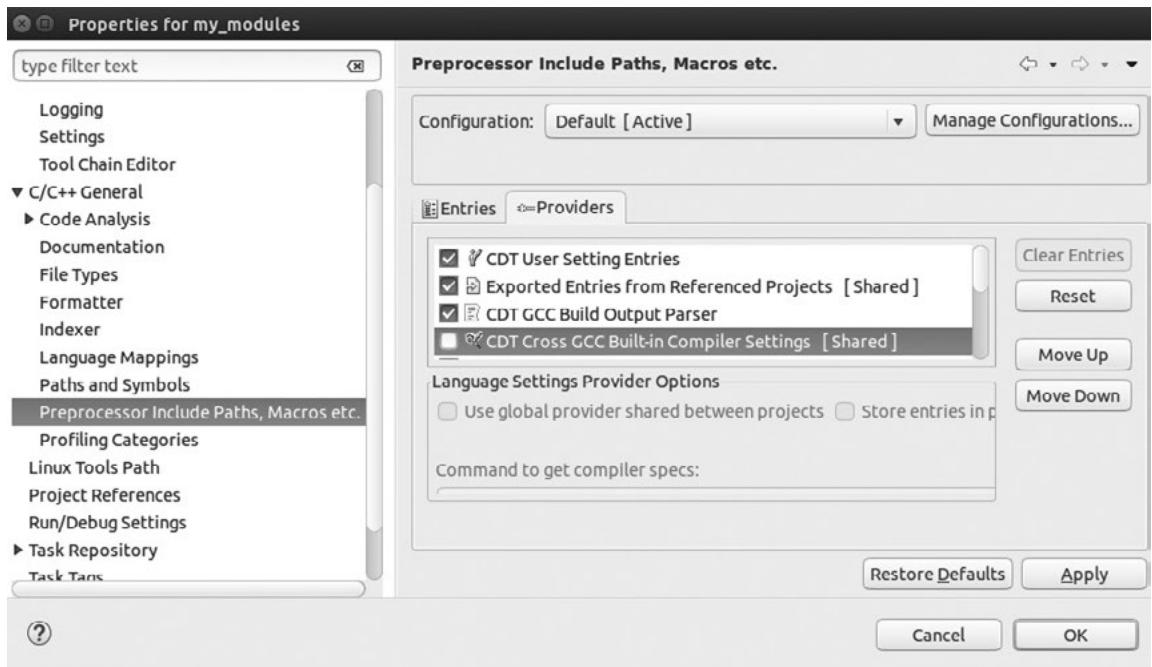


- After creating the project, modify its settings. Go to **Project->Properties** and open the **C/C++ General** selection on the left. Click on **Preprocessor Include Paths, Macros etc..**, then click on the **Entries** tab and select **GNU C** in the Languages list. Select **CDT User Setting Entries** in the **Setting Entries** list. Indexer has to be informed about the kernel configuration. You have to include kconfig.h and printk.h from the kernel project. Click the **Add** tab twice, then in the windows appearing select **Workspace Path** and add the next paths:

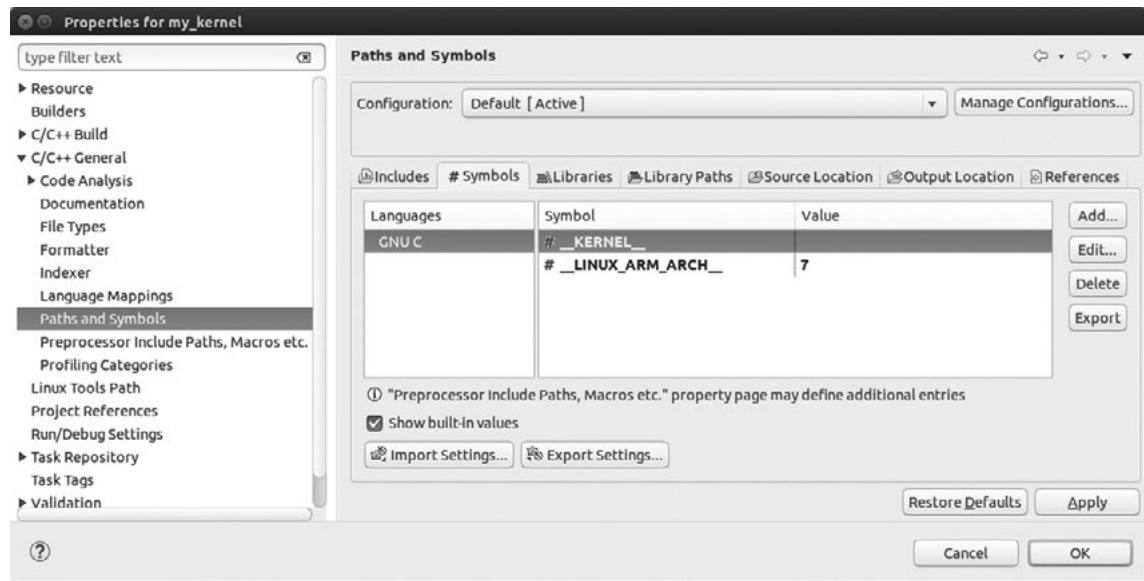
```
/my_kernel/include/linux/kconfig.h  
/my_kernel/include/linux/printk.h
```



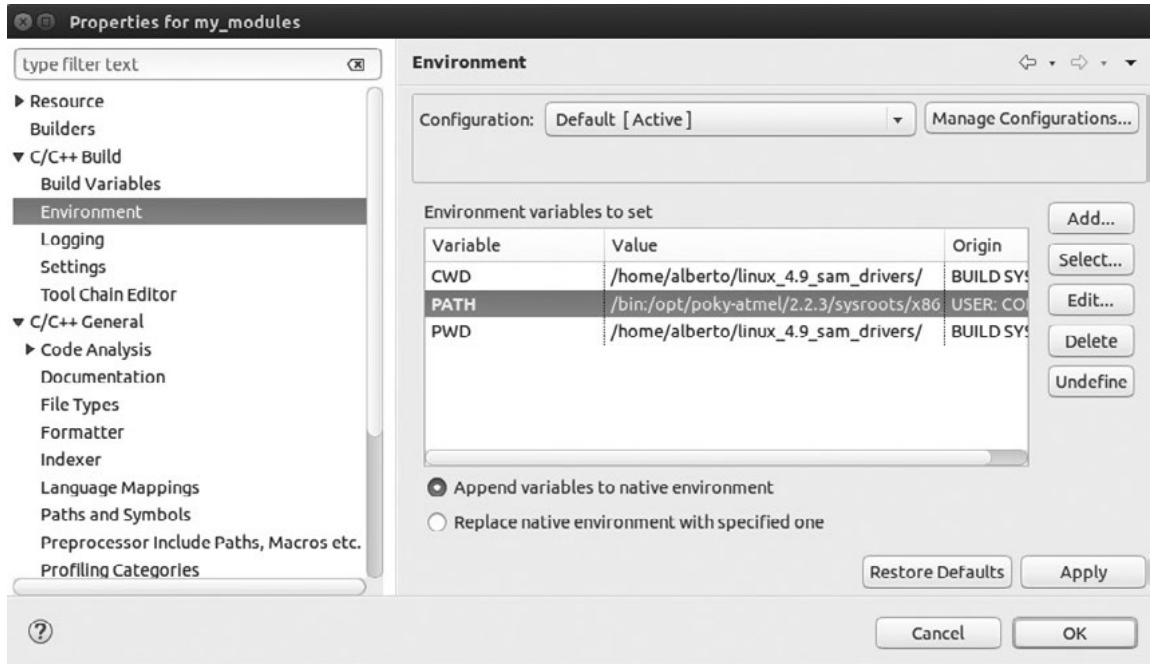
- Turn off the search of the include paths built-in into the toolchain and set the proper include path instead. These include paths are the same as in the kernel project (my_kernel/include and my_kernel/arch/arm/include).



4. Define `__KERNEL__` symbol and `__LINUX_ARM_ARCH__` (Value = 7):



5. Use the sub-page Environment to set the proper path. In this way you do not have to source the environment each time you open Eclipse. Click on the variable PATH. You should have sourced the path before opening Eclipse.

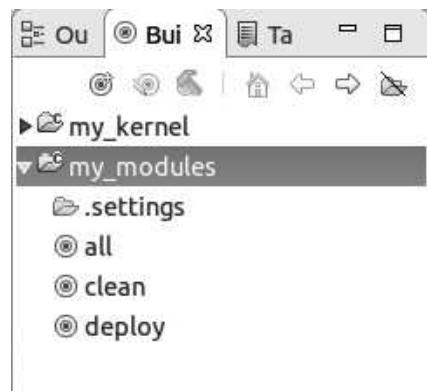


Type in the OK tab to save the settings.

Create new helloworld.c and Makefile files and save them in the kernel modules source directory. Do right-click in your my_modules project and then type New->Source File to create the files. You will write code to these files during the development of the first driver lab. At this time you can leave them empty.

In the Build Targets Tab, do right-click on my_modules, then select New and write Target name: all. Repeat the steps writing deploy and clean target names.

```
my_modules->new->all  
my_modules->new->deploy  
my_modules->new->clean
```



You can now build, clean and deploy your Linux kernel modules by clicking on the generated buttons!!

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 an 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 viceversa.
- 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 struct bus_type structure, which represent one type of bus (e.g.; USB, PCI, I2C).
2. The struct device_driver structure, which represents one driver capable of handling certain devices on a certain bus.
3. The struct 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 struct bus_type and registers it with the kernel's list of bus types. The struct bus_type structure defined in include/linux/device.h represents one type of bus (USB, PCI, I2C, etc.). The registration of the bus in a system is done using the bus_register() function. The struct bus_type is defined as:

```
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;  
};
```

An example of struct bus_type instantiation and bus registration is shown in the following code extracted from the platform core driver (drivers/base/platform.c):

```
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;
}
```

One of the struct bus_type members is a pointer to the struct subsys_private defined in drivers/base/base.h as:

```
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 struct subsys_private 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 `struct subsys_private` 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 `struct device` 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 `struct 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 `struct device` 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 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 maintenance of the device model, is similar to that of any other device driver in that, it registers itself to its bus 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

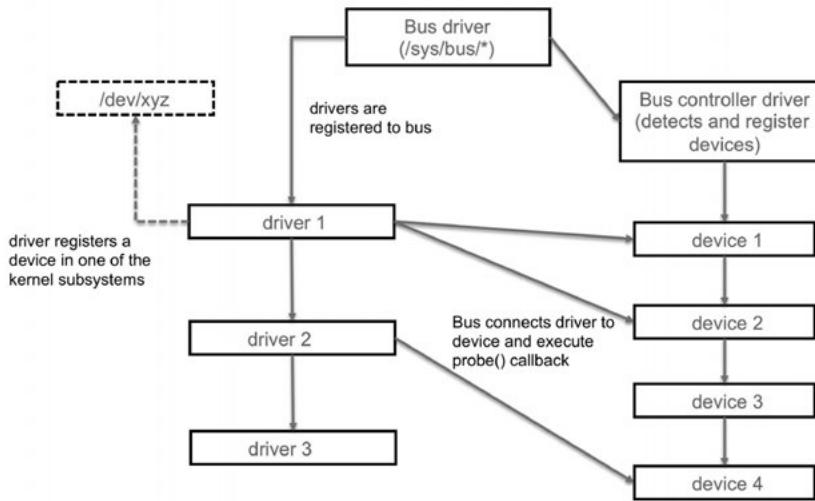
Every device driver registers itself with the bus core driver using `driver_register()`. After that, the device model core tries to bind it with a device. When a device that can be handled by a particular driver is detected, the `probe()` member of the driver is called and the device configuration data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

Each device driver is responsible for instantiating and registering an instance of the `struct device_driver` (defined in `include/linux/device.h`) with the device model core. The `struct device_driver` is defined as:

```
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;  
};
```

- The `bus` member is a pointer to the `struct bus_type` 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 shutdown.

The Linux device model is illustrated in the following figure:



Introduction to the Device Tree

The "Open Firmware Device Tree", or simply Device Tree (DT), is a data structure and language for describing hardware. More specifically, it is a description of hardware that is readable by an operating system so that the operating system doesn't need to hard code details of the machine.

Structurally, the DT is a tree with named **nodes**, and nodes may have an arbitrary number of named **properties** encapsulating arbitrary data. A mechanism also exists to create arbitrary links from one node to another outside of the natural tree structure.

Conceptually, a common set of usage conventions, called "bindings", is defined for how data should appear in the tree to describe typical hardware characteristics including 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 simply text strings, it is easy to extend existing bindings or create new ones by defining new nodes and properties.

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 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 so 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()` in `arch/arm/kernel/setup.c` will call `setup_machine_fdt()` in `arch/arm/kernel/devtree.c` 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_compatible` list in `struct machine_desc`, which is defined in `arch/arm/include/asm/mach/arch.h`.

The `compatible` property contains a sorted list of strings starting with the exact name of the machine. For example, the `sama5d2.dtsi` file under `arch/arm/boot/dts` folder includes the following `compatible` property:

```
compatible = "atmel,sama5d2";
```

Again on ARM, for each `machine_desc`, the kernel looks to see if any of the `dt_compatible` list entries appears in the `compatible` property. If one does, then that `machine_desc` is a candidate for driving the machine. See, for example, the `sama5_alt_dt_board_compatible[]` and `DT_MACHINE_START` declarations in `arch/arm/mach-at91/sama5.c`. They are used to populate a `struct machine_desc`.

```
static const char *const sama5_alt_dt_board_compatible[] __initconst = {
    "atmel,sama5d2",
    "atmel,sama5d4",
    NULL
};

DT_MACHINE_START(sama5_alt_dt, "Atmel SAMAS5")
    /* Maintainer: Atmel */
    .init_machine = sama5_dt_device_init,
    .dt_compatible = sama5_alt_dt_board_compatible,
    .l2c_aux_mask = ~0UL,
MACHINE_END
```

After searching the entire table of machine_descs, the `setup_machine_fdt()` function returns 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 function `setup_machine_fdt()` is also responsible for early scanning of the device tree after selecting `machine_desc`.

2. **Runtime configuration:** In most cases, a DT will be the sole method of communicating data from firmware to the kernel, so also gets 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 when booting Linux it will look something like this:

```
chosen {  
    bootargs = "console=ttyS0,115200 loglevel=8";  
    initrd-start = <0xc8000000>;  
    initrd-end = <0xc8200000>;  
};
```

The `bootargs` property contains the kernel arguments, and the `initrd-*` properties define the address and size of an `initrd` blob. During early boot, the `setup_machine_fdt()` function calls `of_scan_flat_dt()` several times with different helper callbacks to parse device tree data before paging is setup. 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_dt_scan_chosen()` helper is used to parse the `chosen` node including kernel parameters, `early_dt_scan_root()` to initialize the DT address space model, and `early_dt_scan_memory()` to determine the size and location of usable RAM.

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. As can be guessed by 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. For the SAMA5D2 processor `.init_machine()` will call `sama5_dt_device_init()`, which in turn calls `of_platform_populate()` function. See the `sama5_dt_device_init()` function in `arch/arm/mach-at91/sama5.c`:

```
static void __init sama5_dt_device_init(void)
{
    struct soc_device *soc;
    struct device *soc_dev = NULL;

    soc = at91_soc_init(sama5_soc);
    if (soc != NULL)
        soc_dev = soc_device_to_device(soc);

    of_platform_default_populate(NULL, NULL, soc_dev);
    sama5_pm_init();
}

int of_platform_default_populate(struct device_node *root,
                                const struct of_dev_auxdata *lookup,
                                struct device *parent)
{
    return of_platform_populate(root, of_default_bus_match_table, lookup,
                                parent);
}
EXPORT_SYMBOL_GPL(of_platform_default_populate);
```

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.

```
const struct of_device_id of_default_bus_match_table[] = {
    { .compatible = "simple-bus", },
    { .compatible = "simple-mfd", },
    { .compatible = "isa", },
#ifndef CONFIG_ARM_AMBA
    { .compatible = "arm,amba-bus", },
#endif /* CONFIG_ARM_AMBA */
    {} /* Empty terminated list */
};
```

"simple-bus" is defined in the ePAPR 1.0 specification as a property meaning a simple memory mapped bus, so the `of_platform_populate()` code could be written to just assume simple-bus compatible nodes will always be traversed. However, we pass it in as an argument so that board support code can always override the default behaviour.

<http://www.rejoiceblog.com/>

3

The Simplest Drivers

A key concept in the design of the Linux embedded 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 driver to the bus, 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 are 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 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 real kernel code, you will always see something like:

```
printh(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 and specifies the severity of the error message. The **pr_*** macros are simple shorthand definitions located in include/linux/printk.h for their respective **printk** call and should be used in newer drivers.

In the Eclipse Configuration for Developing Linux Drivers section in Chapter 1 you created the project my_modules using the Eclipse IDE. This project will be used to develop all the drivers throughout this book, although you can use your favorite text editor to write the drivers if you do not want to work with Eclipse. The helloworld.c and Makefile files were created and saved in the modules labs directory without writing any code to them. It's time to write code to these files.

You will repeat the same steps to create the driver's source file <module name>.c for the rest of the labs. The same Makefile will be reused for all the labs by simply adding the new <module name>.o to the the Makefile variable obj-m.

In the Build Targets Tab, it was added all, deploy and clean buttons to compile, clean and deploy all the developed lab modules.

See in the next **Listing 3-1** the "helloworld" driver source code (helloworld_imx.c) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (helloworld_sam.c) and BCM2837 (helloworld_rpi.c) drivers can be downloaded from the GitHub repository of this book.

Listing 3-1: helloworld_imx.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");
```

See in the next **Listing 3-2** the Makefile used to compile this first module. The new developed kernel module names will be added to this Makefile.

Secure Copy (SCP) will be added to the Makefile to transfer the modules to the target filesystem, as shown here:

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

Listing 3-2: Makefile

```
obj-m += helloworld.o

KERNEL_DIR ?= $(HOME)/my-linux-imx

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

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

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

helloworld_imx.ko Demonstration

```
root@imx7dsabresd:~# insmod helloworld_imx.ko /* load module */
root@imx7dsabresd:~# modinfo helloworld_imx.ko /* see MODULE macros defined in your
module */
root@imx7dsabresd:~# cat /proc/sys/kernel/tainted /* should be 4096 = GPL, oot */
root@imx7dsabresd:~# cat /sys/module/helloworld_imx/taint /* should be "0" =
untainted */
root@imx7dsabresd:~# rmmod helloworld_imx.ko /* remove module */

/* Now comment out the MODULE_LICENSE macro in helloworld.c. Build, deploy and load
the module again. Boot!. Work with your tainted module */

root@imx7dsabresd:~# insmod helloworld_imx.ko /* load module */
root@imx7dsabresd:~# cat /proc/sys/kernel/tainted /* should be 4097 = proprietary,
oot */
root@imx7dsabresd:~# cat /proc/modules /* helloworld_imx module should be (PO) */
root@imx7dsabresd:~# find /sys -name "*helloworld*" /* find your module in sysfs */
root@imx7dsabresd:~# ls /sys/module/helloworld_imx /* see what the directory
contains */
root@imx7dsabresd:~# cat /sys/module/helloworld_imx/taint /* should be "PO" =
proprietary, oot */
root@imx7dsabresd:~# rmmod helloworld_imx.ko /* remove module */
root@imx7dsabresd:~# cat /proc/sys/kernel/tainted /* still tainted */
```

LAB 3.2: "helloworld with parameters" Module

Many **Linux loadable kernel modules (LKM)**s 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 into the command line that will be set during the module loading. You can also read the parameters via the **sysfs** filesystem.

The sysfs is a virtual filesystem provided by the Linux kernel that exports information about various kernel subsystems, hardware devices, and associated device drivers from the kernel's device model to 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 integer `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. Change the `pr_info` statement in the `hello_init()` function, as shown below:

```
pr_info("parameter num = %d.\n", num);
```
3. Create a new `helloworld_with_parameters.c` file in `my_modules` project and add `helloworld_with_parameters.o` to your `Makefile` `obj-m` variable, then build and deploy the module using Eclipse.

```
obj-m +=helloworld_imx.o  helloworld_imx_with_parameters.o
```

See in the next **Listing 3-3** the "helloworld with parameters" driver source code (`helloworld_imx_with_parameters.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`helloworld_sam_with_parameters.c`) and BCM2837 (`helloworld_rpi_with_parameters.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 3-3: helloworld_imx_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");
```

helloworld_imx_with_parameters.ko Demonstration

```
root@imx7dsabresd:~# insmod helloworld_imx_with_parameters.ko /* insert module */
root@imx7dsabresd:~# rmmod helloworld_imx_with_parameters.ko /* remove module */
root@imx7dsabresd:~# insmod helloworld_imx_with_parameters.ko num=10 /* insert the
module again with a parameter value */

/* read parameter value using sysfs filesystem */
root@imx7dsabresd:~# cat /sys/module/helloworld_imx_with_parameters/parameters/num
root@imx7dsabresd:~# rmmod helloworld_imx_with_parameters.ko /* remove module */
```

LAB 3.3: "helloworld timing" Module

This new kernel module, when unloaded, will display the time (in seconds) that has passed since the driver was loaded.

You will use the `do_gettimeofday()` function located in `kernel/time/keeping.c` to accomplish this task. When called, it fills a `struct timeval` structure with seconds and microseconds. The `struct timeval` structure is defined as:

```
struct timeval {
    __kernel_time_t      tv_sec;    /* seconds */
    __kernel_suseconds_t tv_usec;   /* microseconds */
};
```

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

1. Include the header file that defines `do_gettimeofday()` as a function prototype:

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

2. After the `#include` statements, declare a `struct timeval` structure where the time will be stored when the module is loaded and unloaded:

```
static struct timeval start_time;
```

3. When the module is unloaded the time difference is calculated:

```
pr_info("Unloading module after %ld seconds\n",
        end_time.tv_sec - start_time.tv_sec);
```

See in the next **Listing 3-4** the "helloworld timing" driver source code (`helloworld_imx_with_timing.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`helloworld_sam_with_timing.c`) and BCM2837 (`helloworld_rpi_with_timing.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 3-4: helloworld_imx_with_timing.c

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

static int num = 10;
static struct timeval start_time;

module_param(num, int, S_IRUGO);

static void say_hello(void)
{
    int i;
    for (i = 1; i <= num; i++)
        pr_info("[%d/%d] Hello!\n", i, num);
}

static int __init first_init(void)
{
    do_gettimeofday(&start_time);
```

```
pr_info("Loading first!\n");
say_hello();
return 0;
}

static void __exit first_exit(void)
{
    struct timeval end_time;
    do_gettimeofday(&end_time);
    pr_info("Unloading module after %ld seconds\n",
            end_time.tv_sec - start_time.tv_sec);
    say_hello();
}

module_init(first_init);
module_exit(first_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a module that will print the time \
since it was loaded");
```

helloworld_imx_with_timing.ko Demonstration

```
root@imx7dsabresd:~# insmod helloworld_imx_with_timing.ko /* insert module */
root@imx7dsabresd:~# rmmod helloworld_imx_with_timing.ko /* remove module */
root@imx7dsabresd:~# insmod helloworld_imx_with_timing.ko num=20 /* insert the
module again with a parameter value */

/* read the parameter value using sysfs filesystem */
root@imx7dsabresd:~# cat /sys/module/helloworld_imx_with_timing/parameters/num
root@imx7dsabresd:~# rmmod helloworld_imx_with_timing.ko /* remove module */
```

4

Character Drivers

Typically, an operating system is designed to hide the underlying hardware details from the user or 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 the 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, serial ports, etc. 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.

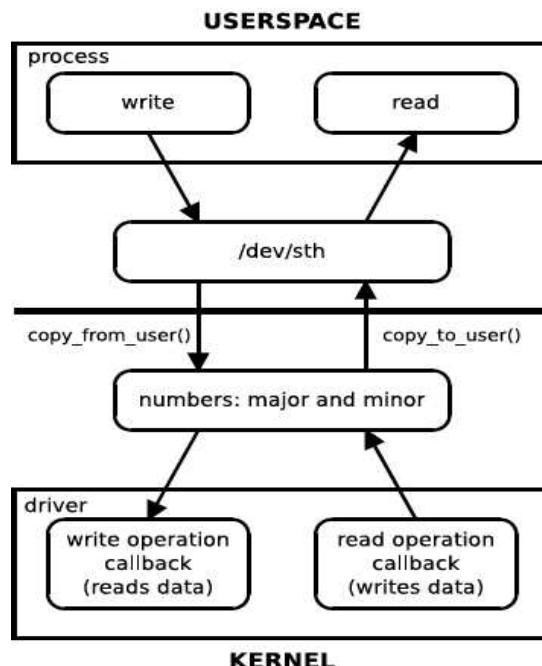
Character devices are identified by a **c** in the first column of a listing, and block devices are identified by a **b**. The access permissions, owner, and group of the device is provided for each device.

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 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 struct file_operations structure defined in include/linux/fs.h and register them. In the struct file_operations below only some of the most common operations for a character driver are shown:

```
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 the driver's operations are called when an user space application makes the corresponding system call (On the kernel side the driver implements and registers callback operations).



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 (and no data was read).
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`. 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 i.MX7D 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 i.MX7D target terminal command line below:

```
root@imx7dsabresd:~# 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 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 write its corresponding driver's callback operations on the kernel side, providing the communication between the user and kernel space.

In the first lab, you saw what a basic helloworld driver looks like. This driver didn't do much except printing some text during installation and removal. In the next lab, you will expand this driver to create a device with a major and minor number. You will also create an 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 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 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 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 function arguments:

- dev: output parameter for first assigned number
- baseminor: first of the requested range of minor numbers
- count: the number of minor numbers required
- name: the name of the associated device or driver

In the line of code below, the second function parameter reserves my_minor_count 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 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 using the cdev_init() function and registered to the kernel 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 MY_MAX_MINORS 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 sequence 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 new driver will now be described:

1. Include the header files required 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`. 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 device numbers when the module is removed:

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

5. Create a struct file_operations named my_dev_fops. This structure defines function pointers for "opening" the device, "reading from" and "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 defined in the struct 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 kernel uses a structure called struct cdev to represent character devices internally. Therefore, you create a struct cdev variable named my_dev and initialize it using the cdev_init() function call, which takes the my_dev variable and the struct file_operations structure named my_dev_fops

as parameters. Once the struct cdev structure is set up, you tell the kernel about it using the cdev_add() function call. You will call these two functions 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 line of code below to the hello_exit() function to delete the struct 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 final two parameters 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.

```
root@imx7dsabresd:~# insmod helloworld_imx_char_driver.ko
root@imx7dsabresd:~# cat /proc/devices /* registered 202 "my_char_device" */
root@imx7dsabresd:~# mknod /dev/mydev c 202 0
```

See in the next **Listing 4-1** the "helloworld character" driver source code (helloworld_imx_char_driver.c) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (helloworld_sam_char_driver.c) and BCM2837 (helloworld_rpi_char_driver.c) drivers can be downloaded from the GitHub repository of this book.

Listing 4-1: helloworld_imx_char_driver.c

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

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

/* define major 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 device numbers */
    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);

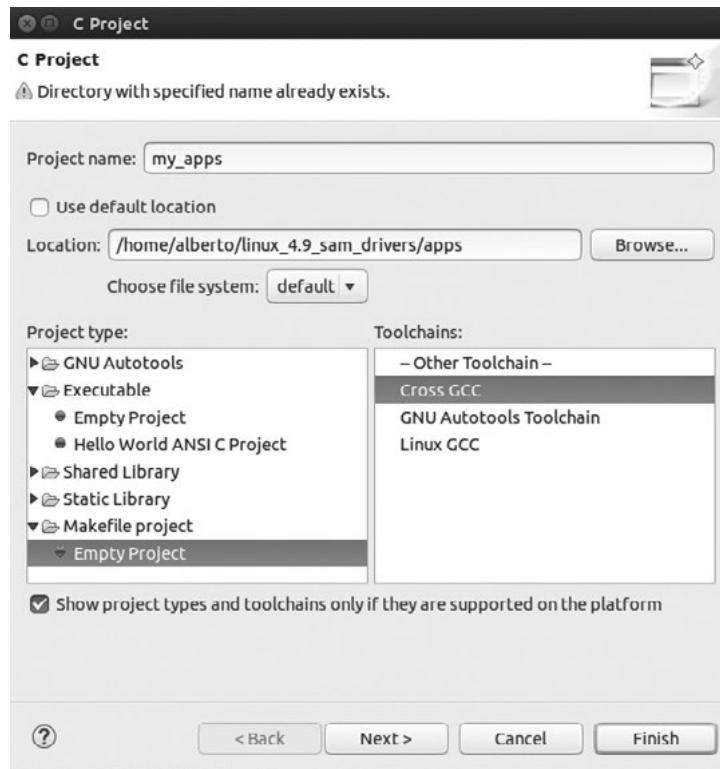
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a module that interacts with the ioctl system call");
```

Now write your application using the Eclipse IDE:

1. The first thing you need to do is create an apps subdirectory under your module's source directory:

```
~$ mkdir /home/<user_name>/linux_4.9_<cpu>_drivers/apps
```

2. Create with Eclipse a new Makefile project with project name my_apps:



3. In the Build Targets Tab of my_apps project add all, deploy and clean buttons:

```
my_apps->new->all  
my_apps->new->deploy  
my_apps->new->clean
```

4. In my_apps project create new files and store them in the apps directory located in /home/<user_name>/Linux_4.9_<cpu>_drivers/apps/:

```
New->File->ioctl_test.c  
New->File-> Makefile
```

You will use the same Makefile for all the applications developed throughout this book, just change the name of the application in the Makefile for every new application you want to build and deploy to the target processor.

Listing 4-2: Makefile

```
all: ioctl_test

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

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_imx_char_driver.ko Demonstration

```
root@imx7dsabresd:~# insmod helloworld_imx_char_driver.ko /* load module */
root@imx7dsabresd:~# cat /proc/devices /* see allocated 202 "my_char_device" */
root@imx7dsabresd:~# ls -l /dev /* mydev is not created under /dev yet */
root@imx7dsabresd:~# mknod /dev/mydev c 202 0 /* create mydev under /dev */
root@imx7dsabresd:~# ls -l /dev /* verify mydev is now created under /dev */
root@imx7dsabresd:~# ./ioctl_test /* run ioctl_test application */
root@imx7dsabresd:~# rmmod helloworld_imx_char_driver.ko /* remove module */
```

Add the Module to the Kernel Build

So far you have been building your driver as a **loadable kernel module (LKM)**, which was loaded during run-time. Now, make the driver a part of the kernel source tree and have the driver built into the kernel binary image. This way the driver is already loaded when the new kernel is booted.

In the kernel root directory, you will find the drivers/char/ folder where all the character drivers reside. First, copy your character driver to this folder:

```
~$ cp ~/linux_4.9_imx7_drivers/helloworld_imx_char_driver.c ~/my-linux-imx/drivers/char/
```

Open the Kconfig file located in the ~/my-linux-imx/drivers/char/ folder using a text editor:

```
~$ gedit ~/my-linux-imx/drivers/char/Kconfig
```

Add the lines below at the end of the file, above endmenu:

```
config HELLOWORLD
  tristate "My simple helloworld driver"
  default n
  help
    The simplest driver.
```

Open the Makefile file:

```
~$ sudo gedit ~/my-linux-imx/drivers/char/Makefile
```

Add the following lines in the end of the Makefile:

```
obj-$(CONFIG_HELLOWORLD) += helloworld_imx_char_driver.o
```

Now that you have modified the Kconfig and Makefile to include the hello_imx_char_driver as a part of the kernel instead of a loadable module, go ahead and build the new kernel image.

Open the menuconfig window. Navigate from the **main menu -> Device Drivers -> Character devices-> My simple helloworld driver**. 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.

```
~/my-linux-imx$ make menuconfig ARCH=arm
```

Opening the .config file in the kernel root directory will show that the CONFIG_HELLOWORLD symbol has been added to it.

Compile the new image and copy it to the tftp folder:

```
~/my-linux-imx$ source /opt/fsl-imx-x11/4.9.11-1.0.0/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
~/my-linux-imx$ make zImage
```

```
~/my-linux-imx$ cp /arch/arm/boot/zImage /var/lib/tftpboot/
```

Boot your i.MX7D target processor:

```
root@imx7dsabresd:~# cat /proc/devices /* registered 202 "my_char_device" */
root@imx7dsabresd:~# mknod /dev/mydev c 202 0 /* assign minor number */
root@imx7dsabresd:~# ./ioctl_test /* run ioctl_test application */
```

Creating Device Files with devtmpfs

Before Linux 2.6.32, on basic Linux systems, the device files had to be created manually 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 virtual 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.

You may wonder how `sysfs` knows about the devices present on a system and what device numbers should be used for them. Drivers that have been compiled into the kernel directly register their objects with a `sysfs` as they are detected by the kernel. For drivers compiled as modules, this registration will happen when the module is loaded. Once the `sysfs` filesystem is mounted on `/sys`, data which the drivers register with `sysfs` is available to the user space processes and to `udev` for processing (including modifications to device nodes). The kernel uses `sysfs` to export device nodes to user space to be used by `udev`.

Device files are created by the kernel via the `devtmpfs` filesystem. Any driver that wishes to register a device node will go through the `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. All device nodes are owned by root and have the default mode of 0600.

Shortly afterward, the kernel will send an uevent to `udev`. Based on the rules specified in the files within the `/etc/udev/rules.d/`, `/lib/udev/rules.d/`, and `/run/udev/rules.d/` directories, `udev` will create additional symlinks to the device node, or change its permissions, owner, or group, or modify the internal `udev` database entry (name) for that object. The rules in these three directories are numbered and all three directories are merged together. If `udev` 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_DEV TMPFS_MOUNT` kernel configuration option makes the kernel mount `devtmpfs` automatically at boot time, except when booting on an `initramfs`.

Use a terminal where the `environment-setup-cortexa7hf-neon-poky-linux-gnueabi` script was not sourced and open the `menuconfig` window. Navigate from the **main menu -> Device Drivers -> Generic Driver Options -> Maintain a devtmpfs filesystem to mount at /dev**. 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.

Compile the new image and copy it to the `tftp` folder:

```
~/my-linux-imx$ source /opt/fsl-imx-x11/4.9.11-1.0.0/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
~/my-linux-imx$ make zImage
~/my-linux-imx$ cp /arch/arm/boot/zImage /var/lib/tftpboot/
```

Boot now your i.MX7D target processor.

LAB 4.2: "class character" Module

In this kernel module lab, you will use your previous helloworld_imx_char_driver at the starting point, but this time the device node will be created using **devtmpfs** instead of doing it manually.

In your current driver, 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 `/sys` for each created device.

The driver creates/destroys the class using the next kernel APIs:

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

The driver creates the device nodes 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_imx_char_driver driver will now be described:

1. Include the next 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 the 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_imx_char_driver driver. That is because it now automatically allocates a major number to the device 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 it for
     * for teaching purposes as we only use one identifier in this
     * driver and dev_no could be used as 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 matchs 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(TITLE_MODULE, CLASS_NAME);

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

    return 0;
}
```

See in the next **Listing 4-4** the "class character" driver source code (`helloworld_imx_class_driver.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`helloworld_sam_class_driver.c`) and BCM2837 (`helloworld_rpi_class_driver.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 4-4: helloworld_imx_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 dynamically device numbers */
ret = alloc_chrdev_region(&dev_no, 0, 1, DEVICE_NAME);
if (ret < 0){
    pr_info("Unable to allocate Major 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_imx_class_driver.ko Demonstration

```
root@imx7dsabresd:~# insmod helloworld_imx_class_driver.ko /* load module */
root@imx7dsabresd:~# ls /sys/class /* check that hello_class is created */
root@imx7dsabresd:~# ls /sys/class/hello_class /* check that mydev is created */
root@imx7dsabresd:~# ls /sys/class/hello_class/mydev /* check entries under mydev */
root@imx7dsabresd:~# cat /sys/class/hello_class/mydev/dev /* see the assigned mayor
and minor numbers */
root@imx7dsabresd:~# ls -l /dev /* verify that mydev is created under /dev */
root@imx7dsabresd:~# ./ioctl_test /* run ioctl_test application */
root@imx7dsabresd:~# rmmod helloworld_imx_class_driver.ko /* remove module */
```

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 an 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 misc device is defined by a struct miscdevice structure in include/linux/miscdevice.h:

```
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 struct file_operations structure
- parent is a pointer to a struct 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 defined as function prototypes in include/linux/miscdevice.h and defined as a function in drivers/char/misc.c:

```
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 struct miscdevice structure. For other cases, the minor number requested is used.

The structure passed 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. Drivers don't need open in fops for this. A zero is returned on success and a negative errno code for failure.

The typical code sequence 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_imx_char_driver driver at the starting point. You will achieve the same result through the misc framework, but will write fewer lines of code!!

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

1. Add the header file that defines the struct miscdevice structure:

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

2. Initialize the struct 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);
```

See in the next **Listing 4-5** the "miscellaneous character" driver source code (misc_imx_driver.c) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (misc_sam_driver.c) and BCM2837 (misc_rpi_driver.c) drivers can be downloaded from the GitHub repository of this book.

Listing 4-5: misc_imx_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 & 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_imx_driver.ko Demonstration

```
root@imx7dsabresd:~# insmod misc_imx_driver.ko /* load the module */
root@imx7dsabresd:~# ls /sys/class/misc /* check that mydev is created under the
misc class folder */
root@imx7dsabresd:~# ls /sys/class/misc/mydev /* check entries under mydev */
root@imx7dsabresd:~# cat /sys/class/misc/mydev/dev /* see the assigned mayor and
minor numbers.The mayor number 10 is assigned by the misc framework */
root@imx7dsabresd:~# ls -l /dev /* verify that mydev is created under /dev */
root@imx7dsabresd:~# ./ioctl_test /* run ioctl_test application */
root@imx7dsabresd:~# rmmod misc_imx_driver.ko /* remove the module */
```

<http://www.rejoiceblog.com/>

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 an 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:

1. By **direct instantiation** of struct platform_device structures, as done on a few old ARM non-Device Tree based platforms. 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 drivers match 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's 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 devices are enumerated statically instead of being discovered dynamically.

Each platform driver is responsible for instantiating and registering an instance of a struct 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. They support power management and shutdown notifications using the standard conventions. The most important members of the struct platform_driver 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;
};
```

In the struct platform_driver you can see a function pointer variable that points to a function named probe(). The probe() function is called when the "bus driver" pairs the "device" to the "device driver". The probe() function is responsible of initializing the device and registering it in the appropriate kernel framework:

1. The probe() function 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, timers, and so on...
3. It registers the device to specific framework(s), (for example, network, misc, serial, input, industrial).

The suspend()/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 using the platform_driver_register(struct platform_driver *drv) function. Register your platform driver in your module init() function, and unregister your platform driver in the module exit() function, as shown in the following example:

```
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(my_platform_driver) macro. This is a 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().

```
/*
 * 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 init() function. When the kernel module is loaded, the **platform device driver** registers itself with the **platform bus driver** 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 struct of_device_id structure is defined in include/linux/mod_devcitable.h:

```
/*
 * 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 definitions required by platform devices/drivers:

```
#include <linux/platform_device.h>
```
2. Declare a list of devices supported by the driver. Create an array of structures struct of_device_id where you initialize the compatible fields with strings that will be used by the

kernel to bind your driver to devices represented in the device tree that include the same compatible property. This will automatically trigger your driver's probe() function if the device tree contains a compatible device entry.

```
static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,hellokeys" },
    {},
}
MODULE_DEVICE_TABLE(of, my_of_ids);
```

3. Add a struct 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 = "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_misctype);
    if (ret_val != 0) {
        pr_err("could not register the misc device mydev");
        return ret_val;
    }
    pr_info("mydev: got minor %i\n", helloworld_misctype.minor);
    return 0;
}

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

5. Register your platform driver to the platform bus core:

```
module_platform_driver(my_platform_driver);
```

6. Modify the device tree files (under arch/arm/boot/dts/ folder) to include your DT driver's device nodes. There must be a DT device node's compatible property identical to the compatible string stored in one of the driver's struct of_device_id structures.

For the **MCIMX7D-SABRE** Board open the DT file imx7d-sdb.dts and add the hellokeys node below the memory node:

```
[...]
/
{
    model = "Freescale i.MX7 SabreSD Board";
    compatible = "fsl,imx7d-sdb", "fsl,imx7d";
    memory {
        reg = <0x80000000 0x80000000>;
    };
    hellokeys {
        compatible = "arrow,hellokeys";
    };
}
[...]
```

For the **SAMA5D2B-XULT** Board open the DT file at91-sama5d2_xplained_common.dtsi and add the hellokeys node below the gpio_keys node:

```
[...]
gpio_keys {
    compatible = "gpio-keys";
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_key_gpio_default>;
    bp1 {
        label = "PB_USER";
        gpios = <&pioA 41 GPIO_ACTIVE_LOW>;
        linux,code = <0x104>;
    };
}
hellokeys {
    compatible = "arrow,hellokeys";
}
[...]
```

For the **Raspberry Pi 3 Model B** Board open the DT file `bcm2710-rpi-3-b.dts` and add the `hellokeys` node inside the `soc` node:

```
[...]
&soc {
    virtgpio: virtgpio {
        compatible = "brcm,bcm2835-virtgpio";
        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
        status = "okay";
    };
    expgpio: expgpio {
        compatible = "brcm,bcm2835-expgpio";
        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
        status = "okay";
    };
    hellokeys {
        compatible = "arrow,hellokeys";
    };
[...]
```

7. Build the modified device tree and load it to the target processor.

See in the next **Listing 5-1** the "platform device" driver source code (`hellokeys_imx.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`hellokeys_sam.c`) and BCM2837 (`hellokeys_rpi.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 5-1: `hellokeys_imx.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);

/* Define 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_imx.ko Demonstration

```
root@imx7dsabresd:~# insmod hellokeys_imx.ko /* load the module, probe() function should
be called */
root@imx7dsabresd:~# find /sys -name "*hellokeys*" /* find all "hellokeys" sysfs entries */
root@imx7dsabresd:~# ls /sys/devices/soc0 /* See devices entries under soc0. Find
hellokeys device entry */
root@imx7dsabresd:~# ls -l /sys/bus/platform/drivers/hellokeys/hellokeys /* this links to
the hellokeys device entry */
root@imx7dsabresd:~# ls -l /sys/bus/platform/devices /* See the devices at platform bus.
Find hellokeys entry */
root@imx7dsabresd:~# ls /sys/bus/platform/drivers/hellokeys /* this is the hellokeys
platform driver entry */
root@imx7dsabresd:~# ls -l /sys/module/hellokeys_imx/drivers /* this is a link to the
hellokeys driver entry */
root@imx7dsabresd:~# ls /sys/class/misc /* check that mydev is created under the misc
class folder */
root@imx7dsabresd:~# ls /sys/class/misc/mydev /* check entries under mydev */
root@imx7dsabresd:~# cat /sys/class/misc/mydev/dev /* see the assigned major and minor
numbers. The major number 10 is assigned by the misc framework */
root@imx7dsabresd:~# ls -l /dev /* verify that mydev is created under /dev */
root@imx7dsabresd:~# ./ioctl_test /* run ioctl_test application */
root@imx7dsabresd:~# rmmod hellokeys_imx.ko /* remove the module */
```

Documentation to Interact with the Hardware

During the development of the next drivers, you will interact with different devices (LEDs, pushbuttons, and I2C devices). You will also interact with some of the processor's peripheral registers, so it will be necessary to download the Technical Reference Manuals of the different processors used in the labs and the schematics of such processor's development boards.

- Go to the NXP Semiconductors site www.nxp.com and download the i.MX7Dual Applications Processor Reference Manual. At the time of this writing, the last revision is Rev. 0.1, 08/2016. You also need to download the **MCIMX7D-SABRE** schematic. The schematic used in this book has the Document number SOURCE:SCH-28590:SPF-28590 and Rev D.
- Go to the Microchip Technology Inc. site www.microchip.com and download the SAMA5D2 Series Datasheet. At the time of this writing, the Datasheet number is DS60001476B. You also need to download the SAMA5D2 (Rev. B) Xplained Ultra User Guide and the **SAMA5D2BXULT** schematic.
- Go to the RASPBERRY PI site www.raspberrypi.org and download the BCM2835 ARM Peripherals guide and the Raspberry-Pi-3B-V1.2-Schematics.

For simplicity, all documentation needed can be downloaded from the GitHub repository of this book. It's recommended to use the documentation of the repository to find in a simple way the pages referenced within this book.

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, they can be configured internally to perform different functions. This is called **pin multiplexing**.

Every **pin/pad** in the MPU has a **name** provided by the manufacturer, for instance **D12** in the i.MX7D processor.

Each pad has a **logical/canonical name**. This is the name that is shown on the schematic symbol inside the part, next to the pin and pin number. This pad name typically corresponds to the first pad functionality. For instance in the i.MX7D processor the **D12** pad name has the logical/ canonical name of **SAI1_RXC**.

The schematics assign a **net name** to the functional wire connected to the pad. This attempts to provide a description of what the wire is actually used for. The net name is usually the same as or similar to the pad multiplexing name. The SAI1_RXC pad can be multiplexed to the next functions:

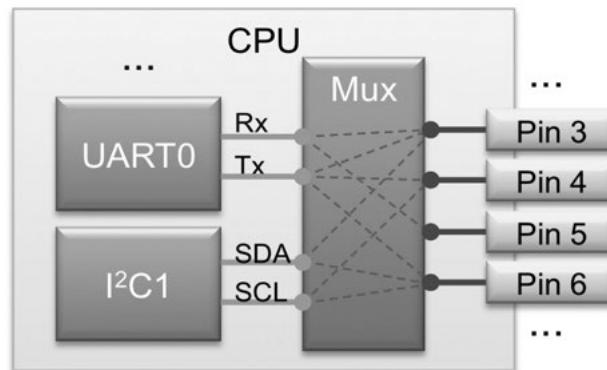
```
SAI1_RX_BCLK
NAND_CE3_B
SAI2_RX_BCLK
I2C4_SDA
FLEXTIMER2_PHA
GPIO6_I017
MQS_LEFT
SRC_CA7_RESET1_B
```

Go to the MCIMX7D-SABRE schematic and look for the D12 pad name. The net name assigned to this pad is **I2C4_SDA** that describes one of the pad functionalities shown above.

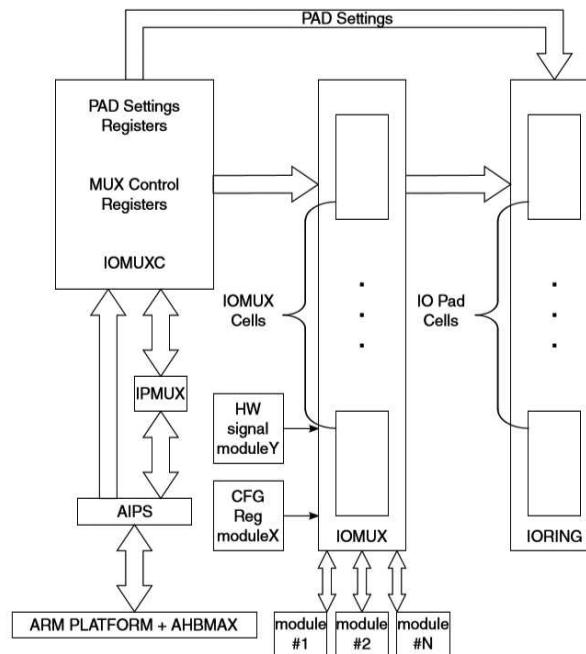
The next sections will explain in detail how the pads are being multiplexed in the i.MX7D family processors. The i.MX7D uses the IOMUX controller (IOMUXC) to accomplish this task.

Pin Controller

In this section, it will be examined the operation of a pin controller, taking the NXP IOMUXC pin controller as a reference. The IOMUX Controller (IOMUXC), together with the IOMUX, enables the IC to share one pad to several functional blocks. This sharing is done by multiplexing the pad's input and output signals, as shown in the following image:



Every module also requires a specific pad setting (such as pull up or keeper), and for each pad, there are up to eight muxing options (called ALT modes). The pad settings parameters are controlled by the IOMUXXC. The IOMUX consists only of combinatorial logic combined from several basic IOMUX cells. Each basic IOMUX cell handles only one pad signal's muxing. The figure below illustrates the IOMUX/IOMUXXC connectivity in the system:



The main IOMUXXC features are:

1. 32-bit software mux control registers (IOMUXC_SW_MUX_CTL_PAD_<PAD NAME> or IOMUXC_SW_MUX_CTL_GRP_<GROUP NAME>) to configure one of eight alternate (ALT) MUX_MODE fields for each pad or a predefined group of pads and to enable forcing of an input path for the pad(s) (SION bit).
2. 32-bit software pad control registers (IOMUXC_SW_PAD_CTL_PAD_<PAD_NAME> or IOMUXC_SW_PAD_CTL_GRP_<GROUP NAME>) to configure specific pad settings of each pad, or a predefined group of pads.
3. 32-bit general purpose registers - 14 (GPR0 to GPR13) 32-bit registers according to SoC requirements for any usage.

4. 32-bit input select control registers to control the input path to a module when more than one pad drives this module input.

Each SW MUX/PAD CTL IOMUXC register handles only one pad or one pad's group. Only the minimum number of registers required by software are implemented by hardware. For example, if only ALT0 and ALT1 modes are used on Pad x then only one bit register will be generated as the MUX_MODE control field in the software mux control register of Pad x.

The software mux control registers may allow the forcing of pads to become input (input path enabled) regardless of the functional direction driven. This may be useful for loopback and GPIO data capture.

Every NXP i.MX7D processor's pad has up to eight potential "iomux" modes. The selection of this iomux mode is controlled by a register, whose name is derived from the canonical pad name, for example, a pad with a canonical name **I2C1_SDA** has a pad mux register named **IOMUXC_SW_MUX_CTL_PAD_I2C1_SDA** (see it in pag. 1704 of the IMX7DRM) that's responsible for configuring the pad between several different modes, as shown below:

- **ALT0_I2C1_SDA** – Select mux mode: ALT0 mux port: SDA of instance: I2C1
- **ALT1_UART4 RTS_B** – Select mux mode: ALT1 mux port: RTS_B of instance: UART4
- **ALT2_FLEXCAN1_TX** – Select mux mode: ALT2 mux port: TX of instance: FLEXCAN1
- **ALT3_ECSPI3_MOSI** – Select mux mode: ALT3 mux port: MOSI of instance: ECSPi3
- **ALT4_CCM_ENET1_REF_CLK** – Select mux mode: ALT4 mux port: ENET1_REF_CLK of instance: ENET1
- **ALT5_GPIO4_IO9** – Select mux mode: ALT5 mux port: IO9 of instance: GPIO4
- **ALT6_SD3_VSELECT** – Select mux mode: ALT6 mux port: VSELECT of instance: SD3

There's also an **IOMUXC_SW_PAD_CTL_PAD_I2C1_SDA** register (see it in pag. 1899 of the IMX7DRM) for each pad that's responsible for configuring the physical drive characteristic of the pad (e.g., hysteresis, pull-up/down, speed, drive strength), which could, again, be mapped to any of the functions above.

Almost every pad has a GPIO function, in the **I2C1_SDA** pad this is **GPIO4_IO9** and GPIO functions are internally tracked by a **bank/bit** convention. There are seven banks of GPIOs with up to 32 bits each. In the i.MX7D the index is 1-based, not 0 based, but the register addresses are all 0-based, requiring you to subtract 1 from the name.

This is the Linux user space naming convention:

1. Almost every pad has a GPIO function as one of its up to eight potential iomux modes.
2. Linux uses a single integer to enumerate all pads, therefore NXP's bank/bit notation for GPIOs must be mapped.

3. The bank/bit to Linux userspace formula is: `linux gpio number = (gpio_bank - 1) * 32 + gpio_bit` so, GPIO4_IO19 maps to $(4 - 1) * 32 + 19 = 115$.

To analyze the registers related to the canonical pad name I2C1_SDA follow the next steps:

1. Look for the Pad Mux Register associated with I2C1_SDA in the IMX7DRM (you can find it on page 1704). The address of the IOMUXC_SW_MUX_CTL_PAD_I2C1_SDA register has an offset of **0x14C** over the IOMUXC peripheral base address **0x30330000**. This base address is the first register address of the IOMUXC controller. The **ALT5 MUX_MODE** configures this pad as a GPIO signal.
2. Look for the Pad Control Register associated with I2C1_SDA in the IMX7DRM (you can find it on page 1899). The address of the IOMUXC_SW_PAD_CTL_PAD_I2C1_SDA register has an offset of **0x3BC** over the base address **0x30330000**.

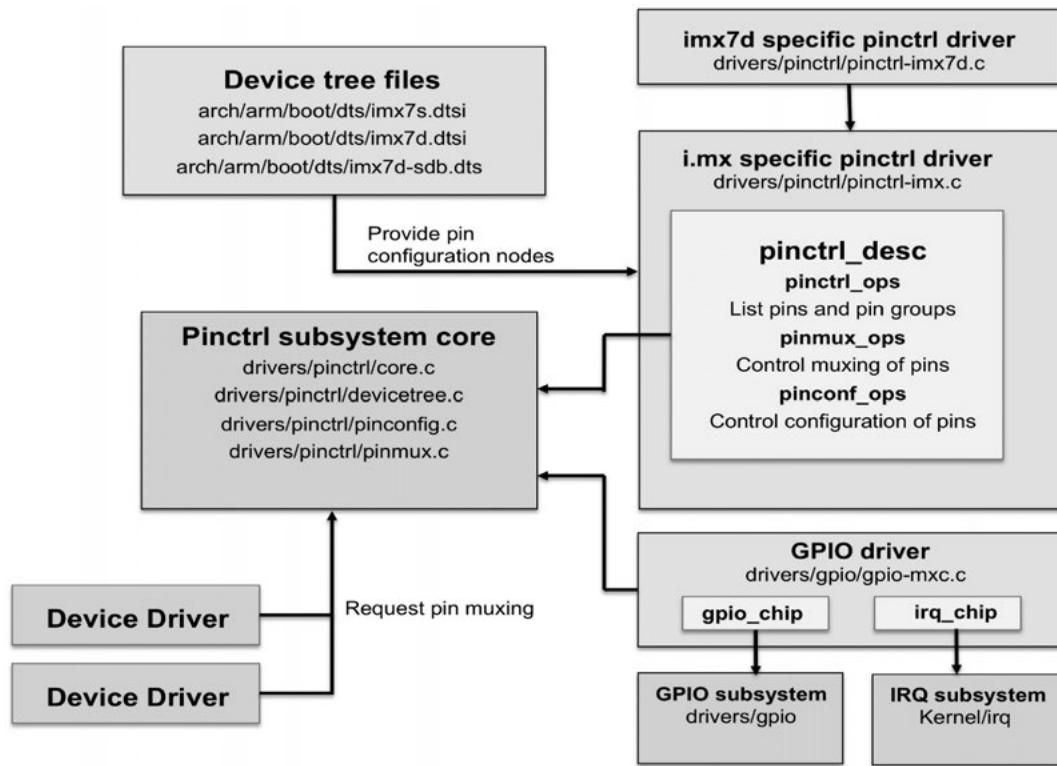
Pin Control Subsystem

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 couldn't be requested by device drivers.

The new **Pinctrl subsystem** mainly developed and maintained by Linus Walleij, from Linaro/ST-Ericsson aims at solving these problems. It is implemented in `drivers/pinctrl/` and provides:

- An API to register a pinctrl driver, for example entities knowing the list of pins, their functions, and how to configure them. Used by SoC specific drivers (for example, `pinctrl-imx7d.c`) to expose pin-muxing capabilities.
- An API for device drivers to request the muxing of a certain set of pins.
- Interaction with the SoC GPIO drivers.

In the next image you can see the interaction of the i.MX7D pinctrl driver and the i.MX7D gpio controller driver with the Pinctrl subsystem:



The Pinctrl subsystem in Linux deals with:

- Enumerating and naming controllable pins.
- Multiplexing of pins.
- 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 i.MX7D pads are named for the Pinctrl subsystem in the pinctrl-imx7d.c pinctrl driver located under drivers/pinctrl/ folder.

```
#include <linux/pinctrl/pinctrl.h>
#include "pinctrl-imx.h"

enum imx7d_pads {
    MX7D_PAD_RESERVED0 = 0,
```

```
MX7D_PAD_RESERVE1 = 1,
MX7D_PAD_RESERVE2 = 2,
MX7D_PAD_RESERVE3 = 3,
MX7D_PAD_RESERVE4 = 4,
MX7D_PAD_GPIO1_IO08 = 5,
MX7D_PAD_GPIO1_IO09 = 6,
MX7D_PAD_GPIO1_IO10 = 7,
MX7D_PAD_GPIO1_IO11 = 8,
MX7D_PAD_GPIO1_IO12 = 9,
MX7D_PAD_GPIO1_IO13 = 10,
MX7D_PAD_GPIO1_IO14 = 11,
MX7D_PAD_GPIO1_IO15 = 12,
[...]
}

enum imx7d_lpsr_pads {
    MX7D_PAD_GPIO1_IO00 = 0,
    MX7D_PAD_GPIO1_IO01 = 1,
    MX7D_PAD_GPIO1_IO02 = 2,
    MX7D_PAD_GPIO1_IO03 = 3,
    MX7D_PAD_GPIO1_IO04 = 4,
    MX7D_PAD_GPIO1_IO05 = 5,
    MX7D_PAD_GPIO1_IO06 = 6,
    MX7D_PAD_GPIO1_IO07 = 7,
};

/* Pad names for the pinmux subsystem */
static const struct pinctrl_pin_desc imx7d_pinctrl_pads[] = {
    IMX_PINCTRL_PIN(MX7D_PAD_RESERVE0),
    IMX_PINCTRL_PIN(MX7D_PAD_RESERVE1),
    IMX_PINCTRL_PIN(MX7D_PAD_RESERVE2),
    IMX_PINCTRL_PIN(MX7D_PAD_RESERVE3),
    IMX_PINCTRL_PIN(MX7D_PAD_RESERVE4),
    IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_IO08),
    IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_IO09),
    IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_IO10),
    IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_IO11),
    IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_IO12),
    IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_IO13),
    IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_IO14),
    IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_IO15),
[...]
}

/* Pad names for the pinmux subsystem */
static const struct pinctrl_pin_desc imx7d_lpsr_pinctrl_pads[] = {
```

```
IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_I000),
IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_I001),
IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_I002),
IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_I003),
IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_I004),
IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_I005),
IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_I006),
IMX_PINCTRL_PIN(MX7D_PAD_GPIO1_I007),
};

static struct imx_pinctrl_soc_info imx7d_pinctrl_info = {
    .pins = imx7d_pinctrl_pads,
    .npins = ARRAY_SIZE(imx7d_pinctrl_pads),
    .gpr_compatible = "fsl,imx7d-iomuxc-gpr",
};

static struct imx_pinctrl_soc_info imx7d_lpsr_pinctrl_info = {
    .pins = imx7d_lpsr_pinctrl_pads,
    .npins = ARRAY_SIZE(imx7d_lpsr_pinctrl_pads),
    .flags = ZERO_OFFSET_VALID,
};

static struct of_device_id imx7d_pinctrl_of_match[] = {
    { .compatible = "fsl,imx7d-iomuxc", .data = &imx7d_pinctrl_info, },
    { .compatible = "fsl,imx7d-iomuxc-lpsr", .data = &imx7d_lpsr_pinctrl_info },
    { /* sentinel */ }
};
```

The imx7d_pinctrl_probe() function will call the imx_pinctrl_probe() function located in drivers/pinctrl/freescale/pinctrl-imx.c. This function configures a struct pinctrl_desc structure with all the information regarding the pin controller and calls imx_pinctrl_probe_dt() to parse the DT finding the number of pin function nodes under the iomuxc node, as well as the number of pin configuration nodes below each pin function node, allocating pin groups. Finally, struct pinctrl_desc is registered against the Pinctrl subsystem via the devm_pinctrl_register() function located in drivers/pinctrl/core.c.

It is important to understand the inner works of the imx_pinctrl_probe_dt() function. After allocating all the pin groups, this function will call imx_pinctrl_parse_functions() once per each pin node found under the iomuxc DT entry, allocating the "mux" and "config" registers associated with each pin.

```
int imx_pinctrl_probe(struct platform_device *pdev,
                      struct imx_pinctrl_soc_info *info)
{
    [...]
    struct pinctrl_desc *imx_pinctrl_desc;
    [...]
```

```
info->pin_regs = devm_kmalloc(&pdev->dev, sizeof(*info->pin_regs) *  
                                info->npins, GFP_KERNEL);  
[...]  
imx_pinctrl_desc = devm_kzalloc(&pdev->dev, sizeof(*imx_pinctrl_desc),  
                                GFP_KERNEL);  
  
imx_pinctrl_desc->name = dev_name(&pdev->dev);  
imx_pinctrl_desc->pins = info->pins;  
imx_pinctrl_desc->npins = info->npins;  
imx_pinctrl_desc->pctlops = &imx_pctrl_ops;  
imx_pinctrl_desc->pmxops = &imx_pmx_ops;  
imx_pinctrl_desc->confops = &imx_pinconf_ops;  
imx_pinctrl_desc->owner = THIS_MODULE;  
  
ret = imx_pinctrl_probe_dt(pdev, info);  
[...]  
ipctl->info = info;  
ipctl->dev = info->dev;  
platform_set_drvdata(pdev, ipctl);  
ipctl->pctl = devm_pinctrl_register(&pdev->dev,  
                                    imx_pinctrl_desc, ipctl);  
[...]  
dev_info(&pdev->dev, "initialized IMX pinctrl driver\n");  
  
return 0;  
}
```

The devm_pinctrl_register() function will call the pinctrl_register() function, that registers all the pins and calls pinctrl_get(), pinctrl_lookup_state() and pinctrl_select_state() functions.

- pinctrl_get() is called in a process context to obtain a handle to all pinctrl information for a given client device. It will allocate a struct from the kernel memory to hold the pinmux state. All mapping table parsing or similar slow operations take place within this API.
- pinctrl_lookup_state() is called in a process context to obtain a handle to a specific state for a client device. This operation may be slow, too.
- pinctrl_select_state() programs the pin controller hardware according to the definition of the state as given by the mapping table.

In the i.MX7D pin controller driver, the pinctrl_select_state() function located in drivers/pinctrl/core.c will take the PINCTRL_STATE_DEFAULT argument (defined as "default" in include/linux/pinctrl/pinctrl-state.h) performing the next sequence of functions:

1. pinmux_enable_setting() -> imx_pmx_set():

The imx_pmx_set() function located in drivers/pinctrl/freescale/pinctrl-imx.c will configure the mux mode for each pin in the group for a specific function. See below the struct pinmux_ops - pinmux operations, declared by the i.MX7D pin controller, where the set_mux field points to the imx_pmx_set function.

```
static const struct pinmux_ops imx_pmx_ops = {
    .get_functions_count = imx_pmx_get_funcs_count,
    .get_function_name = imx_pmx_get_func_name,
    .get_function_groups = imx_pmx_get_groups,
    .set_mux = imx_pmx_set,
    .gpio_request_enable = imx_pmx_gpio_request_enable,
    .gpio_disable_free = imx_pmx_gpio_disable_free,
    .gpio_set_direction = imx_pmx_gpio_set_direction,
};
```

See below part of the code inside the pinmux_enable_setting() function located in drivers/pinctrl/pinmux.c, where imx_pmx_set() function is called:

```
/* Now that we have acquired the pins, encode the mux setting */
for (i = 0; i < num_pins; i++) {
    desc = pin_desc_get(pctldev, pins[i]);
    if (desc == NULL) {
        dev_warn(pctldev->dev,
                 "could not get pin desc for pin %d\n",
                 pins[i]);
        continue;
    }
    desc->mux_setting = &(setting->data.mux);
}

ret = ops->set_mux(pctldev, setting->data.mux.func,
                     setting->data.mux.group);
```

2. pinconf_apply_setting() -> imx_pinconf_set():

The imx_pinconf_set() function located in drivers/pinctrl/freescale/pinctrl-imx.c will configure the pad setting for each pin in the group for a specific function. See below the struct pinconf_ops - pin config operations, declared by the i.MX7D pin controller, where the pin_config_set field points to the imx_pinconfig_set function:

```
static const struct pinconf_ops imx_pinconf_ops = {
    .pin_config_get = imx_pinconf_get,
    .pin_config_set = imx_pinconf_set,
    .pin_config_dbg_show = imx_pinconf_dbg_show,
    .pin_config_group_dbg_show = imx_pinconf_group_dbg_show,
};
```

See below part of the code inside the pinconf_apply_setting() function located in drivers/pinctrl/pinconfig.c, where imx_pinconfig_set() function is called:

```
switch (setting->type) {
case PIN_MAP_TYPE_CONFIGS_PIN:
    if (!ops->pin_config_set) {
        dev_err(pctldev->dev, "missing pin_config_set op\n");
        return -EINVAL;
    }
    ret = ops->pin_config_set(pctldev,
                               setting->data.configs.group_or_pin,
                               setting->data.configs.configs,
                               setting->data.configs.num_configs);
    if (ret < 0) {
        dev_err(pctldev->dev,
                "pin_config_set op failed for pin %d\n",
                setting->data.configs.group_or_pin);
        return ret;
    }
    break;
}
```

When a device driver is about to probe, the device core will automatically attempt to issue pinctrl_get_select_default() on these devices. However, when doing fine-grained state selection and not using the "default" state, you may have to do some device driver handling of the pinctrl handles and states. See for instance the i2c0 controller DT node declaration for the NXP Vybrid vf610 SoC. There are declared two different states: "default" and "gpio":

```
i2c0: i2c@40066000 { /* i2c0 on vf610 */
    compatible = "fsl,vf610-i2c";
    reg = <0x40066000 0x1000>;
    interrupts = <0 71 0x04>;
    dmas = <&edma0 0 50>,
           <&edma0 0 51>;
    dma-names = "rx", "tx";
    pinctrl-names = "default", "gpio";
    pinctrl-0 = <&pinctrl_i2c1>;
    pinctrl-1 = <&pinctrl_i2c1_gpio>;
    scl-gpios = <&gpio5 26 GPIO_ACTIVE_HIGH>;
    sda-gpios = <&gpio5 27 GPIO_ACTIVE_HIGH>;
};
```

You can check now the NXP I2C controller driver located in drivers/i2c/busses/i2c-imx.c to see how both states are implemented in the driver. See below the functions where the states are selected. The two functions are called within the driver's probe() function:

```
static const struct of_device_id i2c_imx_dt_ids[] = {
    { .compatible = "fsl,imx1-i2c", .data = &imx1_i2c_hwdata, },
    { .compatible = "fsl,imx21-i2c", .data = &imx21_i2c_hwdata, },
```

```
{ .compatible = "fsl,vf610-i2c", .data = &vf610_i2c_hwdata, },
{ /* sentinel */ }
};

MODULE_DEVICE_TABLE(of, i2c_imx_dt_ids);

static void i2c_imx_prepare_recovery(struct i2c_adapter *adap)
{
    struct imx_i2c_struct *i2c_imx;
    i2c_imx = container_of(adap, struct imx_i2c_struct, adapter);
    pinctrl_select_state(i2c_imx->pinctrl, i2c_imx->pinctrl_pins_gpio);
}

static void i2c_imx_unprepare_recovery(struct i2c_adapter *adap)
{
    struct imx_i2c_struct *i2c_imx;
    i2c_imx = container_of(adap, struct imx_i2c_struct, adapter);
    pinctrl_select_state(i2c_imx->pinctrl, i2c_imx->pinctrl_pins_default);
}
```

In this driver, the dual functionality is used in order to recover the I2C bus from a fault condition requiring some signal management not allowed by the I2C HW peripheral.

Device Tree Pin Controller Bindings

As you have already seen in the previous Pin Controller section, 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. In the i.MX7D for each PAD there are up to eight muxing options (called ALT modes). Since different modules require different PAD settings (e.g. pull up, keeper) the pin controller controls also the PAD settings parameters. Each pin controller must be represented as a node in device tree, just like any other hardware module.

Hardware modules whose signals are affected by pin configuration are designated "client devices". Again, each client device must be represented as a node in 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. Some client devices need a single static pin configuration, for example, set up during initialization. Others need to reconfigure pins at run-time, for example, to tri-state pins when the device is inactive. Hence, each client device can define a set of named **states**. The number and names of those states is defined by the client device's own binding.

For each client device individually, every pin state is assigned an integer ID. These numbers start at 0, and are contiguous. For each state ID, a unique property exists to define the pin configuration. Each state may also be assigned a name. When names are used, another property exists to map from those names to the integer IDs.

Each client device's own binding determines the set of states that must be defined in its device tree node, and whether to define the set of state IDs that must be provided, or whether to define the set of state names that must be provided. These are the required properties:

- pinctrl-0: List of phandles, each pointing at a **pin configuration node**. These referenced pin configuration nodes must be child nodes of the pin controller that they configure. Multiple entries may exist in this list so that multiple pin controllers may be configured, or so that a state may be built from multiple nodes for a single pin controller, each contributing part of the overall configuration.

These are the optional properties:

- pinctrl-1: List of phandles, each pointing at a pin configuration node within a pin controller.
- [...]
- pinctrl-n: List of phandles, each pointing at a pin configuration node within a pin controller.
- pinctrl-names: The list of names to assign states. List entry 0 defines the name for integer state ID 0, list entry 1 for state ID 1, and so on. For example:

```
/* For a client device requiring named states */
device {
    pinctrl-names = "active", "idle";
    pinctrl-0 = <&state_0_node_a>;
    pinctrl-1 = <&state_1_node_a &state_1_node_b>;
};
```

The pin controller device should contain the pin configuration nodes that client devices reference. For example:

```
pincontroller {
    ... /* Standard DT properties for the device itself elided */

    state_0_node_a {
        ...
    };
    state_1_node_a {
        ...
    };
    state_1_node_b {
        ...
    };
}
```

The contents of each of those pin configuration child nodes is defined entirely by the binding for the individual pin controller device. There exists no common standard for this content. The pinctrl framework only provides generic helper bindings that the pin controller driver can use. You are going to see now how these bindings are defined for the NXP i.MX7D pin controller (IOMUXC).

Open the imx7s.dtsi file located under arch/arm/boot/dts/ folder and look for the iomuxc_lpsr and iomuxc nodes:

```
iomuxc_lpsr: iomuxc-lpsr@302c0000 {  
    compatible = "fsl,imx7d-iomuxc-lpsr";  
    reg = <0x302c0000 0x10000>;  
    fsl,inputsel = <&iomuxc>;  
};  
  
iomuxc: iomuxc@30330000 {  
    compatible = "fsl,imx7d-iomuxc";  
    reg = <0x30330000 0x10000>;  
};
```

The i.MX7D processor supports two iomuxc controllers, the fsl,imx7d-iomuxc controller that is similar as previous iMX SoC generation and the fsl,imx7d-iomuxc-lpsr, which provides low power state retention capabilities on gpios that are part of iomuxc-lpsr (GPIO1_IO7..GPIO1_IO0). While iomuxc-lpsr provides its own set of registers for mux and pad control settings, it shares the input select register from main iomuxc controller for daisy chain settings, the fsl,inputsel property extends fsl,pinctrl driver to support iomuxc-lpsr controller.

The 0x302c0000 and 0x30330000 values in the reg properties are the base address of each of the IOMUXC pin controller registers.

The compatible property fsl,imx7d-iomuxc matchs the struct of_device_id compatible entry of the i.MX7D pin controller driver. Open the pinctrl-imx7d.c file located under drivers/pinctrl/freescale/ directory and look for the compatible properties that match the iomuxc and iomuxc_lpsr DT nodes compatible properties:

```
static const struct of_device_id imx7d_pinctrl_of_match[] = {  
    { .compatible = "fsl,imx7d-iomuxc", .data = &imx7d_pinctrl_info, },  
    { .compatible = "fsl,imx7d-iomuxc-lpsr", .data = &imx7d_lpsr_pinctrl_info },  
    { /* sentinel */ }  
};
```

The DT **pin configuration node** is a node of a **group of pins** that can be used for a specific device or function. This node represents both **mux** and **config** of the pins in that group. The "mux" selects the function mode (also named mux mode) this pin can work on and the "config" configures various pad settings such as pull-up, open drain, drive strength, etc. Each client device node can have a **pinctrl-0 property** with a list of phandles, each pointing to a pin configuration node. The pin configuration node is defined under the **iomuxc** controller node to represent what pinmux functions this SoC supports.

fsl,pins is the required property located inside each pin configuration node: each entry consists of six integers and represents the mux and config setting for one pin. The first five integers

`<mux_reg conf_reg input_reg mux_val input_val>` are specified using a `PIN_FUNC_ID` macro, which can be found in "imx*-pinfunc.h" under the Linux kernel DT folder (imx7d-pinfunc.h for the i.MX7D). The last integer, `CONFIG`, is the pad setting value, for example pull-up, on this pin.

While `iomuxc-lpsr` is intended to be used by dedicated peripherals to take advantages of LPSR power mode, it is also possible that a processor's peripheral uses pads from any of the `iomux` controllers. For example, the `I2C1` controller can use the `SCL` pad from the `iomuxc-lpsr` controller and the `SDA` pad from the `iomuxc` controller, as shown below:

```
i2c1: i2c@30a20000 {  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_i2c1_1 &pinctrl_i2c1_2>;  
};  
  
iomuxc-lpsr@302c0000 {  
    compatible = "fsl,imx7d-iomuxc-lpsr";  
    reg = <0x302c0000 0x10000>;  
    fsl,inputsel = <&iomuxc>;  
  
    pinctrl_i2c1_1: i2c1grp-1 { /* pin configuration node */  
        fsl,pins = <  
                    MX7D_PAD_GPIO1_IO04_I2C1_SCL 0x4000007f  
                >;  
    };  
};  
  
iomuxc@30330000 {  
    compatible = "fsl,imx7d-iomuxc";  
    reg = <0x30330000 0x10000>;  
  
    pinctrl_i2c1_2: i2c1grp-2 {  
        fsl,pins = <  
                    MX7D_PAD_I2C1_SDA_I2C1_SDA 0x4000007f  
                >;  
    };  
};
```

In the previous section you looked for the registers related to the canonical pad name `I2C1_SDA`. You checked that the address of the `IOMUXC_SW_MUX_CTL_PAD_I2C1_SDA` register has an offset of `0x14C` over the `IOMUXC` peripheral base address `0x30330000` and the address of the `IOMUXC_SW_PAD_CTL_PAD_I2C1_SDA` register has an offset of `0x3BC` over the base address `0x30330000`. Find the `PIN_FUNC_ID` macro `MX7D_PAD_I2C1_SDA_GPIO4_IO9`, in the `imx7d-pinfunc.h` file under the Linux kernel DT folder `arch/arm/boot/dts/`.

```
#define MX7D_PAD_I2C1_SDA_GPIO4_IO9 0x014C 0x03BC 0x0000 0x5 0x0
```

The relationship of the PIN_FUNC_ID macros values with the registers used to do the "GPIO" I2C1_SDA pad mux setting will now be analyzed:

- **0x014C** is the offset of the Pad Mux Reg IOMUXC_SW_MUX_CTL_PAD_I2C1_SDA
- **0x03BC** is the offset of the Pad Control Reg IOMUXC_SW_PAD_CTL_PAD_I2C1_SDA
- **0x5** is the ALT5 mode of the Pad Mux Reg IOMUXC_SW_MUX_CTL_PAD_I2C1_SDA

See the documents below for further information about the i.MX7D pin controller DT bindings:

- linux/Documentation/devicetree/bindings/pinctrl/fsl,imx7d-pinctrl.txt
- linux/Documentation/devicetree/bindings/pinctrl/fsl,imx-pinctrl.txt

See the documents below for the SAMA5D2 and BCM283x pin controller DT bindings:

- linux/Documentation/devicetree/bindings/pinctrl/atmel,at91-pio4-pinctrl.txt
- linux/Documentation/devicetree/bindings/pinctrl/brcm,bcm2835-gpio.txt

GPIO Controller Driver

Each GPIO controller driver needs to include the following header, which defines the structures used to define a GPIO driver:

```
#include <linux/gpio/driver.h>
```

Inside a GPIO 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 purely internal: the hardware number of a particular GPIO descriptor is never made visible outside of the driver.

On top of this internal number, each GPIO also needs to have a global number in the integer GPIO namespace so that it can be used with the legacy GPIO interface. Each chip has a "base" number (which can be automatically assigned), and for each GPIO the global number will be (base + hardware number). Although the integer representation is considered deprecated, it still has many users and thus needs to be maintained.

In the gpiolib framework each GPIO controller is packaged as a struct gpio_chip structure (see include/linux/gpio/driver.h for its complete definition) with members common to each controller of that type:

- methods to establish GPIO line direction
- methods used to access GPIO line values
- method to set electrical configuration to a given GPIO line
- method to return the IRQ number associated to 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

The code implementing a struct gpio_chip should support multiple instances of the controller, possibly using the driver model. That code will configure each struct gpio_chip and issue gpiochip_add[_data]() or devm_gpiochip_add_data().

GPIO controllers (GPIO chips) can also provide interrupts, usually cascaded off a parent interrupt controller. The IRQ portions of the GPIO block are implemented using an **irqchip** using the header <linux/irq.h>. So basically such a driver is utilizing two sub-systems simultaneously: gpio and irq.

GPIO irqchips usually fall 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(...);
```

See below the sequence for the i.MX7D GPIO controller driver (drivers/gpio/gpio-mxc.c):

```
/* handle 32 interrupts in one status register */
static void mxc_gpio_irq_handler(struct mxc_gpio_port *port, u32 irq_stat)
{
    while (irq_stat != 0) {
        int irqoffset = fls(irq_stat) - 1;

        if (port->both_edges & (1 << irqoffset))
            mxc_flip_edge(port, irqoffset);

        generic_handle_irq(irq_find_mapping(port->domain, irqoffset));

        irq_stat &= ~(1 << irqoffset);
    }
}

/* MX1 and MX3 has one interrupt *per* gpio port */
static void mx3_gpio_irq_handler(struct irq_desc *desc)
{
    u32 irq_stat;
```

```
struct mxc_gpio_port *port = irq_desc_get_handler_data(desc);
struct irq_chip *chip = irq_desc_get_chip(desc);

chained_irq_enter(chip, desc);

irq_stat = readl(port->base + GPIO_ISR) & readl(port->base + GPIO_IMR);

mxc_gpio_irq_handler(port, irq_stat);

chained_irq_exit(chip, desc);
}
```

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 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 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 then 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);
```

GPIO Descriptor Consumer Interface

This section describes the new 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 `gpiod_` 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

With the descriptor-based interface, GPIOs are identified with an opaque, non-forgeable handler that must be obtained through a call to one of the `gpiod_get()` functions. Like many other kernel subsystems, `devm_gpiod_get()` takes the device that will use the GPIO and the function the requested GPIO is supposed to fulfill:

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

If a function is implemented by using several GPIOs together (for example, a simple LED device that displays digits), an additional index argument can be specified:

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

The `flags` parameter is used to optionally specify a direction and initial value for the GPIO. Values can be:

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

Both functions return either a valid GPIO descriptor, or an error code checkable with `IS_ERR()`.

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

```
void devm_gpiod_put(struct device *dev, struct gpio_desc *desc)
```

Using GPIOs

The first thing a driver must do with a GPIO is setting its direction. If no direction-setting flags have been given to `gpiod_get*()`, this is done by invoking one of the `gpiod_direction_*()` functions:

```
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 and similar contexts.

Use the following calls 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);
```

The values are boolean, zero for low, nonzero for high. When reading the value of an output pin, the value returned should be what's seen on the pin. That won't always match the specified output value, because of issues including open-drain signaling and output latencies.

The get/set calls do not return errors because "invalid GPIO" should have been reported earlier from gpiod_direction_(). However, note that not all platforms can read the value of output pins; those that can't should always return zero. Also, using these calls for GPIOs that can't safely be accessed without sleeping is an error.

Some GPIO controllers must be accessed using message based buses like I2C or SPI. Commands to read or write those GPIO values require waiting to get to the head of a queue to transmit a command and get its response. This requires sleeping, which can't be done from inside IRQ handlers. Platforms that support this type of GPIO distinguish them from other GPIOs by returning nonzero from this call:

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

To access such GPIOs, a different set of accessors is defined:

```
int gpiod_get_value_cansleep(const struct gpio_desc *desc)
void gpiod_set_value_cansleep(struct gpio_desc *desc, int value)
```

Accessing such GPIOs requires a context which may sleep, for example a threaded IRQ handler, and those accessors must be used instead of spinlock-safe accessors without the cansleep() name suffix.

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.

With this, 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
gpiod_set_value(desc, 0);	default (active-high)	low
gpiod_set_value(desc, 1);	default (active-high)	high
gpiod_set_value(desc, 0);	active-low	high
gpiod_set_value(desc, 1);	active-low	low

GPIOs Mapped to IRQs

GPIO lines can quite often be used as IRQs. You can get the Linux IRQ number corresponding to a given GPIO using the following call:

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

It will return a Linux IRQ number, or a negative errno code if the mapping can't be done (most likely because that particular GPIO cannot be used as IRQ). Using a GPIO that wasn't set up as an input using `gpiod_direction_input()`, or using an IRQ number that didn't originally come from `gpiod_to_irq()` results in an unchecked error. The `gpiod_to_irq()` function is not allowed to sleep.

Non-error values returned from `gpiod_to_irq()` can be passed to `request_irq()` or `free_irq()`. They will often be stored into IRQ resources for platform devices, by the board-specific initialization code. Note that IRQ trigger options are part of the IRQ interface, for example, `IRQF_TRIGGER_FALLING`, as are system wakeup capabilities.

GPIOs in Device Tree

GPIOs can easily be 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's node, in a property named `<function>-gpios`, where `<function>` is the function the driver will request through `gpiod_get()`. For example:

```
foo_device {
    compatible = "acme,foo";
    ...
    led-gpios = <&gpio 15 GPIO_ACTIVE_HIGH>, /* red */
                <&gpio 16 GPIO_ACTIVE_HIGH>, /* green */
                <&gpio 17 GPIO_ACTIVE_HIGH>; /* blue */
    ...
    power-gpios = <&gpio 1 GPIO_ACTIVE_LOW>;
};
```

Properties named `<function>-gpio` are also considered valid and old bindings use it but are only supported for compatibility reasons and should not be used for newer bindings since it has been deprecated.

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

```
struct gpio_desc *red, *green, *blue, *power;
red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_HIGH);
```

```
green = gpiod_get_index(dev, "led", 1, GPIOD_OUT_HIGH);
blue = gpiod_get_index(dev, "led", 2, GPIOD_OUT_HIGH);
power = gpiod_get(dev, "power", GPIOD_OUT_HIGH);
```

The led GPIOs will be active-high, while the power GPIO will be active-low. The second parameter of the gpiod_get() functions, the `con_id` string, has to be the <function>-prefix of the GPIO suffixes ("gpios" or "gpio", automatically looked up by the gpiod functions internally) used in the device tree. With above "led-gpios" example, use the prefix without the "-" as `con_id` parameter: "led".

Exchanging Data between Kernel and User Space

Modern operating systems not only prevent one process from accessing another process but also prevent processes from accidentally accessing or manipulating kernel data and services (as the kernel is shared by all the processes). Operating systems achieve this protection by segmenting the whole memory into two logical halves, the user and kernel space. This bifurcation ensures that all processes that are assigned address spaces are mapped to the user space section of memory and kernel data and services run in kernel space. System calls are the kernel's interfaces to expose its services to application processes; they are also called kernel entry points. As system calls are implemented in kernel space, the respective handlers are provided through APIs in user space. When a process requests a kernel service through a system call, the kernel will execute on behalf of the caller process. The kernel is now said to be executing in process context. Similarly, the kernel also responds to interrupts raised by other hardware entities; here, the kernel executes in interrupt context. When in interrupt context, the kernel is not running on behalf of any process.

A driver for a device is the interface between an application and hardware. As a result, you often have to access a given user-space driver device. Accessing process address space from the kernel cannot be done directly (by de-referencing a user-space pointer). Direct access of an user-space pointer can lead to incorrect behavior (depending on architecture, an 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 kernel variable `val` gets the value pointed by the user space pointer `address`.

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

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

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 from the address referenced by from in user-space to the address referenced by to.

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

copy_from_user() copies n bytes from user-space from the address referenced by from in kernel-space to the address referenced 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 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
- Example on x86: IN and OUT instructions

The three processors described in this book use the MMIO access, so this method will be described more in detail during this section.

The Linux driver cannot access physical I/O addresses directly - **MMU mapping** is needed. To access 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 two different functions:

1. Map and remove mapping 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 using the `devm_ioremap()`/`devm_iounmap()` managed functions (defined as a function prototype in `include/linux/io.h` and defined as a function in `lib/devres.c`) that simplify driver code and error handling. Using `ioremap()` in device drivers is now deprecated. You should use the below "managed" functions instead, which simplify driver coding and error handling:

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

Each `struct device` (basic device structure) manages a linked list of resources via its included `struct 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 remove the needed resource releases on error handling, replacing `goto's` and other resource releases with just a return. It also remove resource releases in `remove()` function.

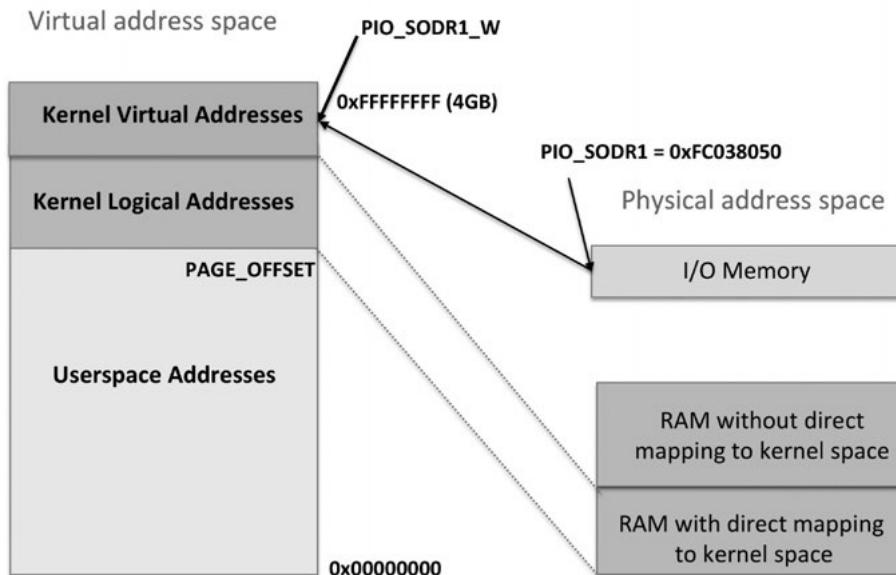
Dereferencing the pointer returned by `devm_ioremap()` is not reliable. Cache and synchronization issues may occur. The kernel provides functions to read and write to virtual addresses. To do PCI-style, little-endian accesses, conversion being done automatically use the functions below:

```
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 SAMA5D2 **PIO_SODR1** register physical address mapping. In the next driver's code you can see how this register is mapped to a virtual address using the `devm_ioremap()` function.



```
PIO_SODR1_W = devm_ioremap(&pdev->dev, PIO_SODR1, sizeof(u32));
```

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 SoC's peripheral register addresses. You will create a character device per each LED using the misc framework and control the LEDs exchanging data between the kernel and user space using `write()` and `read()` driver's operations. You will use `copy_to_user()` and `copy_from_user()` functions to exchange character arrays between kernel and user space.

LAB 5.2 Hardware Description for the i.MX7D Processor

The i.MX7D GPIO general-purpose input/output peripheral provides dedicated general-purpose pins that can be configured as either inputs or outputs. When configured as an output, it is possible to write to an internal register to control the state driven on the output pin. When

configured as an input, it is possible to detect the state of the input by reading the state of an internal register.

The GPIO functionality is provided through registers, an edge-detect circuit, and interrupt generation logic. The registers are:

- Data register (GPIO_DR)
- GPIO direction register (GPIO_GDIR)
- Pad sample register (GPIO_PSR)
- Interrupt control registers (GPIO_ICR1, GPIO_ICR2)
- Edge select register (GPIO_EDGE_SEL)
- Interrupt mask register (GPIO_IMR)
- Interrupt status register (GPIO_ISR)

The GPIO subsystem contains 7 GPIO blocks, which can generate and control up to 32 signals for general purpose. These are the GPIO general purpose input/output logic capabilities:

- Drives specific data to output using the data register (GPIO_DR).
- Controls the direction of the signal using the GPIO direction register (GPIO_GDIR).
- Enables the core to sample the status of the corresponding inputs by reading the pad sample register (GPIO_PSR).

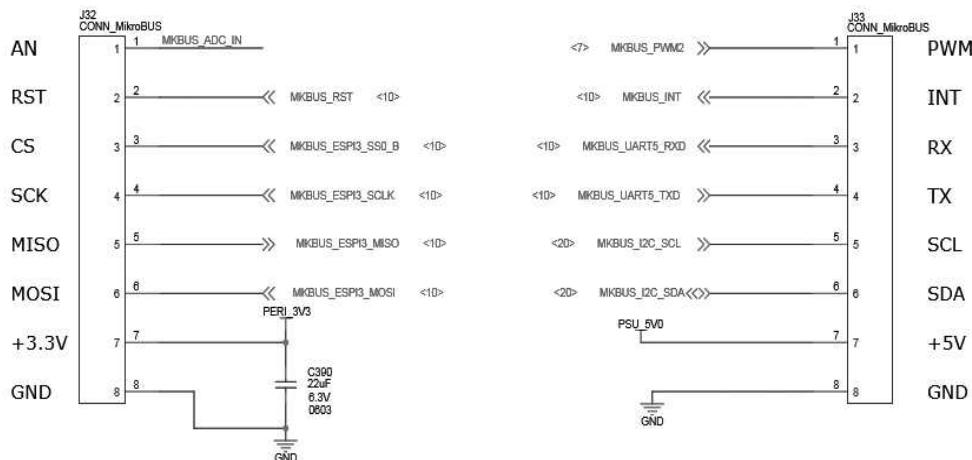
You will write to the GPIO_GDIR and GPIO_DR registers to control the LEDs used in this lab. For further information about i.MX7D GPIOs see the 8.3 General Purpose Input/Output (GPIO) section of the i.MX 7Dual Applications Processor Reference Manual, Rev. 0.1, 08/2016.

You will use three pins of the i.MX7D to control each LED. These pins must be multiplexed as GPIOs in the DT.

The MCIMX7D-SABRE board integrates a **mikroBUS™** socket offering an easy hardware configuration to MikroElektronika's wide range of click boards™ add-on modules. The mikroBUS™ pinout and the click board ecosystem is especially suitable for developers who are working on multi-purpose, modular products.

The purpose of mikroBUS™ is to enable easy hardware expandability with a large number of standardized compact add-on boards, each one carrying a single sensor, transceiver, display, encoder, motor driver, connection port, or any other electronic module or integrated circuit. Created by MikroElektronika, mikroBUS™ is an open standard - anyone can implement mikroBUS™ in their hardware design, as long as the requirements set by the MikroBUS specification are being met.

Go to the pag.20 of the MCIMX7D-SABRE schematic to see the MikroBUS connector:



You will use the MOSI pin to control the green LED, the SCK pin to control the blue LED and the PWM pin to control the red LED.

To obtain the LEDs, you will use the Color click™ accessory board with mikroBUS™ form factor. It's a compact and easy solution for adding red, green, blue and clear light sensing to your design. It features TCS3471 color RGB light-to-digital converter, three NPN resistor-equipped transistors as well as RGB LED. In this lab, you will only use the RGB LED. See the Color click™ accessory board board at <https://www.mikroe.com/color-click>. You can download the schematic from that link or from the GitHub repository of this book.

Connect the MCIMX7D-SABRE mikroBUS™ PWM pin to the Color click™ RD pin, MOSI pin to GR, and SCK to BL. Supply VCC = +5V from MCIMX7D-SABRE board to the Color click™ accessory board and connect GND between both boards.

LAB 5.2 Hardware Description for the SAMA5D2 Processor

The SAMA5D2 Parallel Input/Output Controller (PIO) manages up to 128 fully programmable input/output lines (the number of I/O groups is 4, PA, PB, PC and PD). Each I/O line may be dedicated as a general purpose I/O or be assigned to a function of an embedded peripheral. This ensures effective optimization of the pins of the product.

Each I/O line of the PIO Controller features:

- An input change interrupt enabling level change detection on any I/O line
- Rising edge, falling edge, both edge, low-level or high-level detection on any I/O line
- A glitch filter providing rejection of glitches lower than one-half of PIO clock cycle

- A debouncing filter providing rejection of unwanted pulses from key or push button operations
- Multi-drive capability similar to an open drain I/O line
- Control of the pull-up and pull-down of the I/O line
- Input visibility and output control
- Secure or Non-Secure management of the I/O line

Each pin is configurable, depending on the product, as either a general purpose I/O line only, or as an I/O line multiplexed with up to 6 peripheral I/Os. As the multiplexing is hardware defined and thus product-dependent, the hardware designer and programmer must carefully determine the configuration of the PIO Controllers required by their application. When an I/O line is general purpose only, i.e., not multiplexed with any peripheral I/O, programming of the PIO Controller regarding the assignment to a peripheral has no effect and only the PIO Controller can control how the pin is driven by the product.

To configure I/O lines it must be first defined which I/O line in the group will be targeted by writing a 1 to the corresponding bit in the PIO Mask Register (PIO_MSKRx). Several I/O lines in an I/O group can be configured at the same time by setting the corresponding bits in PIO_MSKRx. Then, writing the PIO Configuration Register (PIO_CFGRx) apply the configuration to the I/O line(s) defined in PIO_MSKRx.

The PIO Controller provides multiplexing of up to 6 peripheral functions on a single pin. The selection is performed by writing the FUNC field in PIO_CFGRx. The selected function is applied to the I/O line(s) defined in PIO_MSKRx. When FUNC is 0, no peripheral is selected and the General Purpose PIO (GPIO) mode is selected (in this mode, the I/O line is controlled by the PIO Controller). When FUNC is not 0, the peripheral selected to control the I/O line depends on the FUNC value.

When the I/O line is assigned to a peripheral function, i.e., the corresponding FUNC field of the line configuration is not 0, the drive of the I/O line is controlled by the peripheral. According to the FUNC value, the selected peripheral determines whether the pin is driven or not.

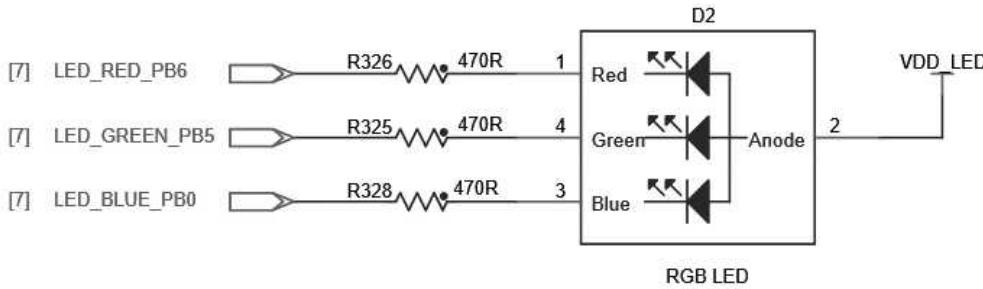
When the FUNC field of an I/O line is 0, then the I/O line is set in General Purpose mode and the I/O line can be configured to be driven by the PIO Controller instead of the peripheral.

If the DIR bit of the I/O line configuration (PIO_CFGRx) is set (OUTPUT) then the I/O line can be driven by the PIO Controller. The level driven on an I/O line can be determined by writing in the PIO Set Output Data Register (PIO_SODRx) and the PIO Clear Output Data Register (PIO_CODRx). These write operations, respectively, set and clear the PIO Output Data Status Register (PIO_ODSRx), which represents the data driven on the I/O lines. Writing PIO_ODSRx directly is possible and only affects the I/O line set to 1 in PIO_MSKRx. When the DIR bit of the I/O line configuration is at zero, the corresponding I/O line is used as an input only.

For further information about SAMA5D2 GPIOs see the 34. Parallel Input/Output Controller (PIO) section of the SAMA5D2 SERIES DS60001476B data-sheet.

You will use three pins of the SAMA5D2 to control each LED. These pins must be multiplexed as GPIOs in the DT.

The SAMA5D2B-XULT board integrates a RGB LED. Go to the pag.11 of the SAMA5D2B-XULT schematic to see the RGB LED:



LAB 5.2 Hardware Description for the BCM2837 Processor

The BCM2837 processor is the Broadcom chip used in the Raspberry Pi 3, and in later models of the Raspberry Pi 2. The underlying architecture of the BCM2837 is identical to the BCM2836. The only significant difference is the replacement of the ARMv7 quad core cluster with a quad-core ARM Cortex A53 (ARMv8) cluster. The BCM2835 processor is the the Broadcom chip used in the Raspberry Pi Model A, B, B+, the Compute Module, and the Raspberry Pi Zero.

The BCM2837 has 54 general-purpose I/O (GPIO) lines split into two banks. All GPIO pins have at least two alternative functions within BCM. 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 nth 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 an input (by default) then the value in the SET{n} field is ignored. However, if the pin is subsequently defined

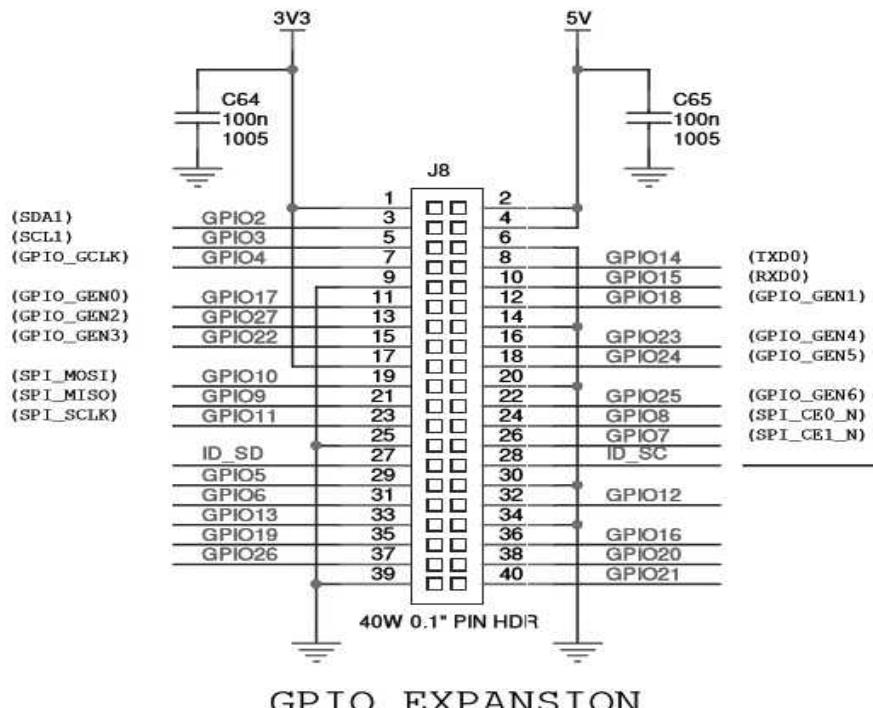
as an output then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations.

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 an 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 BCM2837 GPIOs see the section 6 General Purpose I/O (GPIO) of the BCM2835 ARM Peripherals guide.

You will use three pins of the BCM2837 to control each LED. These pins must be multiplexed as GPIOs in the DT.

To obtain the GPIOs, you will use the GPIO expansion connector. Go to the Raspberry-Pi-3B-V1.2-Schematics to see the connector:



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

Connect the GPIO EXPANSION GPIO27 pin to the Color click™ RD pin, GPIO22 pin to GR, and GPIO26 to BL.

LAB 5.2 Device Tree for the i.MX7D Processor

From the MCIMX7D-SABRE mikroBUS™ socket, you see that MOSI pin connects to the SAI2_TXC pad of the i.MX7D processor, the SCK pin to the SAI2_RXD pad and the PWM pin to the GPIO1_IO02 pad. You have to configure these SAI2_TXC, SAI2_RXD and GPIO1_IO02 pads as GPIO signals. To look for the macros that assign the required functionality (GPIO) go to the imx7d-pinfunc.h file under arch/arm/boot/dts/ and find the next macros:

```
#define MX7D_PAD_SAI2_TX_BCLK_GPIO06_IO20      0x0220 0x0490 0x0000 0x5 0x0
#define MX7D_PAD_SAI2_RX_DATA_GPIO06_IO21        0x0224 0x0494 0x0000 0x5 0x0
```

Go now to the imx7d-pinfunc-lpsr.h file under arch/arm/boot/dts/ and find the next macro:

```
#define MX7D_PAD_GPIO1_IO02_GPIO1_IO2 0x0008 0x0038 0x0000 0x0 0x0
```

The five integers of the macro above are:

- IOMUX register offset (0x0008)
- Pad configuration register offset (0x0038)
- Select input daisy chain register offset (0x0000)
- IOMUX configuration setting (0x0)
- Select input daisy chain setting (0x0)

The GPIO1_IO02 pad is part of the fsl,imx7d-iomuxc-lpsr controller.

You need a sixth integer that corresponds to the configuration for the PAD control register. This number defines the low-level physical settings of the pin. You can build this integer using the information found under the device tree pinctrl documentation binding folder located at Documentation/devicetree/bindings/pinctrl/. For the i.MX7D check the fsl,imx7d-pinctrl.txt file. You can also copy and then modify another pin definition that has similar functionality in the DT file. You can use the 0x11 value for your selected PADs.

```
CONFIG bits definition:
PAD_CTL_PUS_100K_DOWN      (0 << 5)
PAD_CTL_PUS_5K_UP          (1 << 5)
PAD_CTL_PUS_47K_UP         (2 << 5)
PAD_CTL_PUS_100K_UP        (3 << 5)
PAD_CTL_PUE                (1 << 4)
PAD_CTL_HYS                (1 << 3)
```

PAD_CTL_SRE_SLOW	(1 << 2)
PAD_CTL_SRE_FAST	(0 << 2)
PAD_CTL_DSE_X1	(0 << 0)
PAD_CTL_DSE_X2	(1 << 0)
PAD_CTL_DSE_X3	(2 << 0)
PAD_CTL_DSE_X4	(3 << 0)

Now, you will modify the device tree file imx7d-sdb.dts adding the next code in bold below:

```
/ {
    model = "Freescale i.MX7 SabreSD Board";
    compatible = "fsl,imx7d-sdb", "fsl,imx7d";

    memory {
        reg = <0x80000000 0x80000000>;
    };

    [...]

    ledred {
        compatible = "arrow,RGBleds";
        label = "ledred";
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_gpio_leds &pinctrl_gpio_led>;
    };

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

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

    [...]

    &iomuxc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hog_1>;

        imx7d-sdb {

            pinctrl_hog_1: hoggrp-1 {
                fsl,pins = <
                    MX7D_PAD_EPDC_BDR0__GPIO2_IO28      0x59
                >;
            };
        };
    };
}
```

For simplicity, you will perform the IOMUX setting of the pads in the first **ledred** device. You need to be careful not to configure the same pad twice in the device tree. IOMUX configurations are set by the drivers in the order the kernel probes the configured device. If the same pad is configured differently by two drivers, the settings associated with the last-probed driver will apply. If you look for the **ecspi3** node in the device tree file `imx7d-sdb.dts` you can see that the pin configuration

defined on the pinctrl-0 property assigns the "default" name and points to the pinctrl_ecspi3 and pinctrl_ecspi3_cs pin function nodes:

```
pinctrl_ecspi3_cs: ecspi3_cs_grp {
    fsl,pins = <
        MX7D_PAD_SD2_CD_B__GPIO5_I09      0x80000000
        MX7D_PAD_SAI2_TX_DATA__GPIO6_I022  0x2
    >;
};

pinctrl_ecspi3: ecspi3grp {
    fsl,pins = <
        MX7D_PAD_SAI2_TX_SYNC__ECSPI3_MISO  0x2
        MX7D_PAD_SAI2_TX_BCLK__ECSPI3_MOSI  0x2
        MX7D_PAD_SAI2_RX_DATA__ECSPI3_SCLK  0x2
    >;
};
```

The SAI2_TX_BCLK and SAI2_RX_DATA pads are being multiplexed for two different drivers (the ecspi3 and your LED RGB one). You can comment out the entire definition for ecspi3 or disable it by changing status to "disabled". Use the code below if you choose the second option:

```
&ecspi3 {
    fsl,spi-num-chipselects = <1>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ecspi3 &pinctrl_ecspi3_cs>;
    cs-gpios = <&gpio5 9 GPIO_ACTIVE_HIGH>, <&gpio6 22 0>;
    status = "disabled";

    [...]
}
```

LAB 5.2 Device Tree for the SAMA5D2 Processor

From the SAMA5D2B-XULT board, you see that LED_RED_PB6 pin connects to the PB6 pad of the SAMA5D2 processor, the LED_GREEN_PB5 pin to the PB5 pad and the LED_BLUE_PB0 pin to the PB0 pad. You have to configure the PB6, PB5 and PB0 pads as GPIO signals. To look for the macro that assigns the required functionality (GPIO) go to the sama5d2-pinfunc.h file under arch/arm/boot/dts/ and find the next macros:

```
#define PIN_PB6__GPIO      PINMUX_PIN(PIN_PB6, 0, 0)
#define PIN_PB5__GPIO      PINMUX_PIN(PIN_PB5, 0, 0)
#define PIN_PB0__GPIO      PINMUX_PIN(PIN_PB0, 0, 0)
```

According to the data-sheet you can see that the pads can be used for several functionalities. See below the PB5 and PB6 pads functions:

B7	D7	D6	VDDIOP0	GPIO_QSPI	PB5	I/O	-	-	A	TCLK2	I	1	PIO, I, PU, ST
									B	D10	I/O	1	
									C	PWMH2	O	1	
									D	QSPI1_SCK	O	2	
									F	GTSUCOMP	O	3	
									A	TIOA2	I/O	1	
C7	B5	A3	VDDIOP0	GPIO	PB6	I/O	-	-	B	D11	I/O	1	PIO, I, PU, ST
									C	PWML2	O	1	
									D	QSPI1_CS	O	2	
									F	GTXER	O	3	

You can see in the sama5d2-pinfuc.h file the macros associated to the PB5 pin, the last two numbers of the macro correspond to the function of the pad and the IO Set of the signal. For instance PIN_PB5__TCLK2 has function number 1 (A) and TCLK2 signal correspond to the IO Set 1. PIN_PB5__D10 has function number 2 (B) and D10 signal correspond to the IO Set 1.

```
#define PIN_PB5 37
#define PIN_PB5__GPIO PINMUX_PIN(PIN_PB5, 0, 0)
#define PIN_PB5__TCLK2 PINMUX_PIN(PIN_PB5, 1, 1)
#define PIN_PB5__D10 PINMUX_PIN(PIN_PB5, 2, 1)
#define PIN_PB5__PWMH2 PINMUX_PIN(PIN_PB5, 3, 1)
#define PIN_PB5__QSPI1_SCKPINMUX_PIN(PIN_PB5, 4, 2)
#define PIN_PB5__GTSUCOMP PINMUX_PIN(PIN_PB5, 6, 3)
```

Note: I/Os for each peripheral are grouped into IO sets, listed in the column "IO Set" in the pinout tables. For all peripherals, it is mandatory to use I/Os that belong to the same IO set. The timings are not guaranteed when IOs from different IO sets are mixed.

Each pin function node will list the pins it needs and how to configure these pins:

```
node {
    pinmux = <PIN_NUMBER_PINMUX>;
    GENERIC_PINCONFIG;
};
```

These are the properties:

- **pinmux:** integer array. Each integer represents a pin number plus mux and ioset settings. Use the macros from arch/arm/boot/dts/<soc>-pinfunc.h file to get the right representation of the pin.
- **GENERIC_PINCONFIG:** generic pinconfig options to use, bias-disable, bias-pull-down, bias-pull-up, drive-open-drain, input-schmitt-enable, input-debounce, output-low, output-high.

For further info go to the device tree pinctrl documentation binding folder located at Documentation/devicetree/bindings/pinctrl/ and examine the atmel,at91-pio4-pinctrl.txt file.

You can see in the device tree file at91-sama5d2_xplained_common.dtsi that the pinctrl_led_gpio_default pin function node is already configured under pinctrl node:

```
pinctrl@fc038000 {  
  
    pinctrl_adc_default: adc_default {  
        pinmux = <PIN_PD23_GPIO>;  
        bias-disable;  
    };  
  
    [...]  
  
    pinctrl_led_gpio_default: led_gpio_default {  
        pinmux = <PIN_PB0_GPIO>,  
                <PIN_PB5_GPIO>,  
                <PIN_PB6_GPIO>;  
        bias-pull-up;  
    };  
  
    [...]  
}  
  
/  
{  
    model = "Atmel SAMA5D2 Xplained";  
    compatible = "atmel,sama5d2-xplained", "atmel,sama5d2", "atmel,sama5";  
  
    chosen {  
        stdout-path = "serial0:115200n8";  
    };  
  
    [...]  
  
    ledred {  
        compatible = "arrow,RGBleds";  
        label = "ledred";  
        pinctrl-0 = <&pinctrl_led_gpio_default>;  
    };  
  
    ledgreen {  
        compatible = "arrow,RGBleds";  
        label = "ledgreen";  
    };  
  
    ledblue {
```

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

    [...]
};
```

You can see that the "gpio-leds" driver is configuring the same LEDs. Disable it by changing status to "disabled".

```
leds {
    compatible = "gpio-leds";
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_led_gpio_default>;
    status = "disabled";

    red {
        label = "red";
        gpios = <&pioA 38 GPIO_ACTIVE_LOW>;
    };

    green {
        label = "green";
        gpios = <&pioA 37 GPIO_ACTIVE_LOW>;
    };

    blue {
        label = "blue";
        gpios = <&pioA 32 GPIO_ACTIVE_LOW>;
        linux,default-trigger = "heartbeat";
    };
};
```

LAB 5.2 Device Tree for the BCM2837 Processor

From the Raspberry Pi 3 Model B board, you see that GPIO EXPANSION GPIO27 pin connects to the GPIO27 pad of the BCM2837 processor, the GPIO22 pin to GPIO22 pad, and GPIO26 pin to GPIO26 pad.

Each pin configuration node lists the pin(s) to which it applies, and one or more of the mux function to select on those pin(s), and pull-up/down configuration. These are the properties:

- **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.

Now, you can modify the device tree file bcm2710-rpi-3-b.dts adding the next code in bold below:

```
/ {  
    model = "Raspberry Pi 3 Model B";  
};  
  
&gpio {  
    sdhost_pins: sdhost_pins {  
        brcm,pins = <48 49 50 51 52 53>;  
        brcm,function = <4>; /* alt0 */  
    };  
    [...]  
  
    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";
};

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

[...]

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 used for the SAMA5D2 processor:

```
#define PIO_PB0_MASK (1 << 0) /* blue */
#define PIO_PB5_MASK (1 << 5) /* green */
#define PIO_PB6_MASK (1 << 6) /* red */
#define PIO_CFGR1_MASK (1 << 8) /* masked bits direction (output), no PUEN, no PDEN */
*/
#define PIO_MASK_ALL_LEDS (PIO_PB0_MASK | PIO_PB5_MASK | PIO_PB6_MASK)
```

3. Declare physical I/O register addresses. See below the addresses used in the SAMA5D2 processor:

```
static int PIO_SODR1 = 0xFC038050;
static int PIO_CODR1 = 0xFC038054;
static int PIO_MSKR1 = 0xFC038040;
static int PIO_CFGR1 = 0xFC038044;
```

4. Declare **__iomem** pointers that will hold the virtual addresses returned by the `dev_ioremap()` function:

```
static void __iomem *PIO_SODR1_W;
static void __iomem *PIO_CODR1_W;
static void __iomem *PIO_MSKR1_V;
static void __iomem *PIO_CFGR1_V;
```

5. You now need to create a private structure that will hold each device specific information. In this driver, you will handle multiple char devices, so a struct miscdevice will be created for each device, then initialized and added to your device specific private data structure. The second field of the private structure is a led_mask variable that will hold a red, green, or blue mask depending of 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 this private structure and allocate it for each new probed device. The `probe()` function will be called tree times (once per each DT node matched which includes the "arrow,RGBleds" compatible property) allocating the corresponding devices:

```
struct led_dev *led_device;
led_device = devm_kzalloc(&pdev->dev, sizeof(struct led_dev), GFP_KERNEL);
```

7. Obtain virtual addresses in the probe() routine using devm_ioremap() function and store them in the __iomem pointers. See below the mappings for the SAMA5D2 processor:

```
PIO_MSKR1_V = devm_ioremap(&pdev->dev, PIO_MSKR1, sizeof(u32));
PIO_SODR1_W = devm_ioremap(&pdev->dev, PIO_SODR1, sizeof(u32));
PIO_CODR1_W = devm_ioremap(&pdev->dev, PIO_CODR1, sizeof(u32));
PIO_CFR1_V = devm_ioremap(&pdev->dev, PIO_CFR1, sizeof(u32));
```

8. Initialize each struct 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 property label of each probed 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 struct file_operations structure is needed to define which driver's functions to call when an user opens, closes, reads, and writes to the char file. This structure will be stored in the struct miscdevice 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 struct miscdevice that you create to the private data field for the file that's being opened. This is useful, so in your write/read functions you can get access to the struct miscdevice, which will let you get access to the registers and other custom values for this specific device:

```
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. Register each device with the kernel in the probe() routine using the misc_register() function. The platform_set_drvdata() function will attach each private structure to each struct platform_device one. This will allow you to access your private data structure in other parts

of the driver. You will recover the private structure in each remove() function call (called three times) 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 private struct led_dev structure. However, as the struct miscdevice is accessible through file->private_data, and is a member of the struct lev_dev structure, you can use a magic macro to compute the address of the parent structure; the container_of() macro gets the structure that struct miscdevice is stored inside of (which is your private struct led_dev structure). The copy_from_user() function will get the on/off command from user space, then you will write to the corresponding processor's register 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);

    copy_from_user(led_device->led_value, buff, count);

    led_device->led_value[count-1] = '\0';

    /* compare strings to switch on/off the LED */
    if(!strcmp(led_device->led_value, led_on)) {
        iowrite32(led_device->led_mask, PIO_CODR1_W);
    }
    else if (!strcmp(led_device->led_value, led_off)) {
        iowrite32(led_device->led_mask, PIO_SODR1_W);
    }
    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. Recover the private structure using the container_of() macro and return to the user application the device's private structure variable led_value (on/off) using the copy_to_user() function:

```
static ssize_t led_read(struct file *file, char __user *buff,
                       size_t count, loff_t *ppos)
{
    int len;
    struct led_dev *led_device;

    led_device = container_of(file->private_data, struct led_dev,
                             led_misc_device);

    if(*ppos == 0){
        len = strlen(led_device->led_value);
        led_device->led_value[len] = '\n'; /* add \n after on/off */
        copy_to_user(buff, &led_device->led_value, len+1);

        *ppos+=1;
        return sizeof(led_device->led_value); /* exit first func call */
    }

    return 0; /* exit and do not recall func again */
}
```

13. Declare a list of devices supported by the driver. Create an array of structures struct of_device_id where you initialize with strings the compatible fields that will be used by the kernel to bind your driver with 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 struct 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 using the platform_driver_register() function:

```
static int led_init(void)
{
    ret_val = platform_driver_register(&led_platform_driver);
    return 0;
}
```

16. Build the modified device tree, and load it to the target processor.

See in the next **Listing 5-2** the "RGB LED platform device" driver source code (ledRGB_sam_platform.c) for the SAMA5D2 processor.

Note: The source code for the i.MX7D (ledRGB_imx_platform.c) and BCM2837 (ledRGB_rpi_platform.c) drivers can be downloaded from the GitHub repository of this book.

Listing 5-2: ledRGB_sam_platform.c

```
#include <linux/module.h>
#include <linux/fs.h> /* struct file_operations */
/* platform_driver_register(), platform_set_drvdata() */
#include <linux/platform_device.h>
#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() */

/* declare a private structure */
struct led_dev
{
    struct miscdevice led_misc_device; /* assign char device for each led */
    u32 led_mask; /* different mask if led is R,G or B */
    const char *led_name; /* assigned value cannot be modified */
    char led_value[8];
};

/* Declare physical addresses */
static int PIO_SODR1 = 0xFC038050;
static int PIO_CODR1 = 0xFC038054;
static int PIO_MSKR1 = 0xFC038040;
static int PIO_CFGR1 = 0xFC038044;

/* Declare __iomem pointers that will keep virtual addresses */
static void __iomem *PIO_SODR1_W;
static void __iomem *PIO_CODR1_W;
```

```
static void __iomem *PIO_MSKR1_V;
static void __iomem *PIO_CFGR1_V;

/* Declare masks to configure the different registers */
#define PIO_PB0_MASK (1 << 0) /* blue */
#define PIO_PB5_MASK (1 << 5) /* green */
#define PIO_PB6_MASK (1 << 6) /* red */
#define PIO_CFGR1_MASK (1 << 8) /* masked bits direction (output), no PUEN, no PDEN
*/
#define PIO_MASK_ALL_LEDS (PIO_PB0_MASK | PIO_PB5_MASK | PIO_PB6_MASK)

/* send on/off value from your terminal to control each led */
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);

    /*
     * terminal echo add \n character.
     * led_device->led_value = "on\n" or "off\n" after copy_from_user"
     * count = 3 for "on\n" and 4 for "off\n"
     */
    if(copy_from_user(led_device->led_value, buff, count)) {
        pr_info("Bad copied value\n");
        return -EFAULT;
    }

    /*
     * Replace \n for \0 in led_device->led_value
     * char array to create a char string
     */
    led_device->led_value[count-1] = '\0';

    pr_info("This message is received from User Space: %s\n",
           led_device->led_value);

    /* compare strings to switch on/off the LED */
    if(!strcmp(led_device->led_value, led_on)) {
        iowrite32(led_device->led_mask, PIO_CODR1_W);
    }
}
```

```
else if (!strcmp(led_device->led_value, led_off)) {
    iowrite32(led_device->led_mask, PIO_SODR1_W);
}
else {
    pr_info("Bad value\n");
    return -EINVAL;
}

pr_info("led_write() is exit.\n");
return count;
}

/*
 * read each LED status on/off
 * use cat from terminal to read
 * led_read is entered until *ppos > 0
 * twice in this function
 */
static ssize_t led_read(struct file *file, char __user *buff,
                       size_t count, loff_t *ppos)
{
    int len;
    struct led_dev *led_device;

    led_device = container_of(file->private_data, struct led_dev,
                             led_misc_device);

    if(*ppos == 0){
        len = strlen(led_device->led_value);
        pr_info("the size of the message is %d\n", len); /* 2 for on */
        led_device->led_value[len] = '\n'; /* add \n after on/off */
        if(copy_to_user(buff, &led_device->led_value, len+1)){
            pr_info("Failed to return led_value to user space\n");
            return -EFAULT;
        }
        *ppos+=1; /* increment *ppos to exit the function in next call */
        return sizeof(led_device->led_value); /* exit first func call */
    }

    return 0; /* exit and do not recall func again */
}

static const struct file_operations led_fops = {
    .owner = THIS_MODULE,
    .read = led_read,
    .write = led_write,
};
```

```
static int __init led_probe(struct platform_device *pdev)
{
    /* create a private structure */
    struct led_dev *led_device;
    int ret_val;

    /* initialize all the leds to off */
    char led_val[8] = "off\n";

    pr_info("led_probe enter\n");

    /* Get virtual addresses */
    PIO_MSKR1_V = devm_ioremap(&pdev->dev, PIO_MSKR1, sizeof(u32));
    PIO_SODR1_W = devm_ioremap(&pdev->dev, PIO_SODR1, sizeof(u32));
    PIO_CODR1_W = devm_ioremap(&pdev->dev, PIO_CODR1, sizeof(u32));
    PIO_CFGR1_V = devm_ioremap(&pdev->dev, PIO_CFGR1, sizeof(u32));

    /* Initialize all the virtual registers */
    iowrite32(PIO_MASK_ALL_LEDS, PIO_MSKR1_V); /* Enable all leds */
    iowrite32(PIO_CFGR1_MASK, PIO_CFGR1_V); /* set enabled leds to output */
    iowrite32(PIO_MASK_ALL_LEDS, PIO_SODR1_W); /* Clear all the leds */

    /* Allocate a private structure */
    led_device = devm_kzalloc(&pdev->dev, sizeof(struct led_dev), GFP_KERNEL);

    /*
     * read each node label property in each probe() call
     * probe() is called 3 times, once per compatible = "arrow,RGBleds"
     * found below each ledred, ledgreen and ledblue node
     */
    of_property_read_string(pdev->dev.of_node, "label", &led_device->led_name);

    /* create a device for each led found */
    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;

    /* Assigns a different mask for each led */
    if (strcmp(led_device->led_name,"ledred") == 0) {
        led_device->led_mask = PIO_PB6_MASK;
    }
    else if (strcmp(led_device->led_name,"ledgreen") == 0) {
        led_device->led_mask = PIO_PB5_MASK;
    }
    else if (strcmp(led_device->led_name,"ledblue") == 0) {
        led_device->led_mask = PIO_PB0_MASK;
    }
    else {
```

```
pr_info("Bad device tree value\n");
return -EINVAL;
}

/* Initialize each led status to off */
memcpy(led_device->led_value, led_val, sizeof(led_val));

/* register each led device */
ret_val = misc_register(&led_device->led_misc_device);
if (ret_val) return ret_val; /* misc_register returns 0 if success */

/*
 * Attach the private structure to the pdev structure
 * to recover it in each remove() function call
 */
platform_set_drvdata(pdev, led_device);

pr_info("leds_probe exit\n");

return 0;
}

/* The remove() function is called 3 times, once per led */
static int __exit 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;
    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 led_exit(void)
{
    pr_info("led driver enter\n");

    /* Clear all the leds before exiting */
    iowrite32(PIO_MASK_ALL_LEDS, PIO_SODR1_W);

    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_sam_platform.ko Demonstration

```
root@sama5d2-xplained:~# insmod ledRGB_sam_platform.ko /* load module */
root@sama5d2-xplained:~# ls /dev/led* /* see led devices */
root@sama5d2-xplained:~# echo on > /dev/ledblue /* set led blue ON */
root@sama5d2-xplained:~# echo on > /dev/ledred /* set led red ON */
root@sama5d2-xplained:~# echo on > /dev/ledgreen /* set led green ON */
```

```
root@sama5d2-xplained:~# echo off > /dev/ledgreen /* set led green OFF */
root@sama5d2-xplained:~# echo off > /dev/ledred /* set led red OFF */
root@sama5d2-xplained:~# cat /dev/ledblue /* check led blue status */
root@sama5d2-xplained:~# cat /dev/ledgreen /* check led green status */
root@sama5d2-xplained:~# cat /dev/ledred /* check led red status */
root@sama5d2-xplained:~# rmmod ledRGB_sam_platform.ko /* remove module */
```

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, 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 at include/linux/platform_device.h. This resource array has been filled with the DT device node resource properties (for example, reg, clocks, interrupts). See the different resource properties that can be accessed by index in the resource-names.txt file located under Documentation/devicetree/bindings/.

```
struct platform_device {
    const char      *name;
    u32              id;
    struct device   dev;
    u32              num_resources;
    struct resource *resource;
};
```

See below the definition of the struct 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 element included in the previous 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 from the resource array:

1. **platform_get_resource()** defined as a function in drivers/base/platform.c gets a resource for a device, and returns a struct resource structure filled with the resource values so that you can use these values later in the driver code, for example, to map them in the virtual space using devm_ioremap() if they are physical memory addresses (specified with IORESOURCE_MEM type). In the platform_get_resource() function all the array resources are checked until the resource type is found, and then the struct resource is returned. See the code of the platform_get_resource() function below:

```
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 tells the function which device you are interested in, so 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 at include/linux/ioport.h. 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 (DT reg property) with the next 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 struct resource variable:

The resource_size() function will return from the struct resource 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()` will extract the struct resource from the struct `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 to 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 a 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 a LED class device first needs to allocate and fill a struct `led_classdev` structure defined in `include/linux/leds.h`, then will call the `devm_led_classdev_register()` function defined in `drivers/leds/led-class.c`, which registers a new LED class object.

```
struct led_classdev {
    const char          *name;
    enum led_brightness brightness;
    enum led_brightness max_brightness;

[...]

/* Set LED brightness Level
 * Must not sleep. 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 a 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);

[...]

#endif 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 using the misc framework and writing to several GPIO registers. You used the kernel write file operation and the kernel 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 simplifying the code and adding more functionalities such as blinking each LED with a given period and duty cycle.

LAB 5.3 DT for the i.MX7D, SAMA5D2 and BCM2837 Processors

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's 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 to walk through 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 is done, 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 a GPIO register base address, and the size of the address region it is assigned. After the driver and the device are probed, the `platform_get_resource()` function returns a struct resource filled with the `reg` property values so that you can use these values later in the driver code, mapping them in the virtual space using the `devm_ioremap()` function.

For the **MCIMX7D-SABRE** board modify the device tree file `imx7d-sdb.dts` adding the next code in bold. The i.MX7D GPIO memory map is shown in the section 8.3.5 GPIO Memory Map/Register Definition of the i.MX 7Dual Applications Processor Reference Manual, Rev. 0.1, 08/2016. The **0x30200000** base address of the `reg` property is the GPIO data register (GPIO1_DR) address.

```
/ {  
    model = "Freescale i.MX7 SabreSD Board";  
    compatible = "fsl,imx7d-sdb", "fsl,imx7d";  
  
    memory {  
        reg = <0x80000000 0x80000000>;  
    };  
  
    [...]  
  
    ledclassRGB {  
        compatible = "arrow,RGBclassleds";  
        reg = <0x30200000 0x60000>;  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_gpio_leds &pinctrl_gpio_led>;  
        red {  
            label = "red";  
        };  
  
        green {  
            label = "green";  
        };  
  
        blue {  
            label = "blue";  
            linux,default-trigger = "heartbeat";  
        };  
    };
```

[...]

For the **SAMA5D2B-XULT** board modify the device tree file `at91-sama5d2_xplained_common.dtsi` adding the next code in bold. The **0xFC038000** base address of the `reg` property is the PIO Mask Register (PIO_MSKR0) address. See the section 34.7.1 PIO Mask Register of the SAMA5D2 Series Datasheet.

```
/ {  
    model = "Atmel SAMA5D2 Xplained";  
    compatible = "atmel,sama5d2-xplained", "atmel,sama5d2", "atmel,sama5";  
  
    chosen {  
        stdout-path = "serial0:115200n8";  
    };  
  
    [...]  
  
    ledclassRGB {  
        compatible = "arrow,RGBclassleds";  
        reg = <0xFC038000 0x4000>;  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_led_gpio_default>;  
        status = "okay";  
  
        red {  
            label = "red";  
        };  
  
        green {  
            label = "green";  
        };  
  
        blue {  
            label = "blue";  
            linux,default-trigger = "heartbeat";  
        };  
    };  
};  
[...]
```

For the **Raspberry Pi 3 Model B** board modify the device tree file `bcm2710-rpi-3-b.dts` adding the next code in bold. The **0x7e200000** base address of the `reg` property is the GPFSEL0 register address. See the section 6.1 Register View of the BCM2835 ARM Peripherals guide.

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

```
&soc {  
    [...]  
  
    expgpio: expgpio {  
        compatible = "brcm,bcm2835-expgpio";  
        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. You will take the base address included in the DT reg property as a reference, adding an offset to it for each of the registers. See below the masks used for the SAMA5D2 processor:

```
#define PIO_SODR1_offset 0x50
#define PIO_CODR1_offset 0x54
#define PIO_CFGR1_offset 0x44
#define PIO_MSKR1_offset 0x40

#define PIO_PB0_MASK (1 << 0)
#define PIO_PB5_MASK (1 << 5)
#define PIO_PB6_MASK (1 << 6)
#define PIO_CFGR1_MASK (1 << 8)

#define PIO_MASK_ALL_LEDS (PIO_PB0_MASK | PIO_PB5_MASK | PIO_PB6_MASK)
```

3. You need to create a private structure that will hold the RGB LED device's specific information. In this driver, the first field of the private structure is a 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 holding the GPIO register base address. The last field of the private structure is a struct led_classdev 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 an extract of 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's 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 using the devm_kzalloc() function, then initialize the struct led_classdev field of each allocated private structure.
- The devm_led_classdev_register() function registers each LED class device to the LED subsystem.

```
static int __init ledclass_probe(struct platform_device *pdev)
{
    void __iomem *g_ioremap_addr;
    struct device_node *child;
    struct resource *r;
    struct device *dev = &pdev->dev;
    int count, ret;

    /* get your first memory resource from device tree */
    r = platform_get_resource(pdev, IORESOURCE_MEM, 0);

    /* ioremap your memory region */
    g_ioremap_addr = devm_ioremap(dev, r->start, resource_size(r));
    if (!g_ioremap_addr) {
        dev_err(dev, "ioremap failed \n");
        return -ENOMEM;
    }

    [...]

    /* parse each children device under LED RGB parent node */
    for_each_child_of_node(dev->of_node, child){

        struct led_device *led_device;
        /* create a Led_classdev struct for each child device */
        struct led_classdev *cdev;

        /* Allocate a private structure in each "for" iteration */
        led_device = devm_kzalloc(dev, sizeof(*led_device), GFP_KERNEL);
        cdev = &led_device->cdev;
        led_device->base = g_ioremap_addr;

        /* assigns a mask to each children (child) device */
        of_property_read_string(child, "label", &cdev->name);
    }
}
```

```
if (strcmp(cdev->name, "red") == 0) {
    led_device->led_mask = PIO_PB6_MASK;
    led_device->cdev.default_trigger = "heartbeat";
}
else if (strcmp(cdev->name, "green") == 0) {
    led_device->led_mask = PIO_PB5_MASK;
}
else if (strcmp(cdev->name, "blue") == 0) {
    led_device->led_mask = PIO_PB0_MASK;
}
else {
    dev_info(dev, "Bad device tree value\n");
    return -EINVAL;
}

/* Initialize each led_classdev struct */
/* Disable timer trigger until Led is on */
led_device->cdev.brightness = LED_OFF;
led_device->cdev.brightness_set = led_control;

/* register each LED class device */
ret = devm_led_classdev_register(dev, &led_device->cdev);
}

dev_info(dev, "leds_probe exit\n");

return 0;
}
```

5. Write the LED brightness led_control() function. Every time you write to the sysfs brightness 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 private structure associated to each device is recovered using the container_of() function, then you can write to each register using the iowrite32() function, that takes as a first parameter the recovered led_mask value associated to each LED. See below the led_control() function for the SAMA5D2 processor:

```
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(PIO_MASK_ALL_LEDS, led->base + PIO_SODR1_offset);

    if (b != LED_OFF) /* LED ON */
        iowrite32(led->led_mask, led->base + PIO_CODR1_offset);
    else
        /* LED OFF */
}
```

```
        iowrite32(led->led_mask, led->base + PIO_SODR1_offset);  
    }
```

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 struct 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 using the platform_driver_register() function:

```
static int led_init(void)  
{  
    ret_val = platform_driver_register(&led_platform_driver);  
    return 0;  
}
```

9. Build the modified device tree, and load it to the target processor.

See in the next **Listing 5-3** the "RGB LED class" driver source code (ledRGB_sam_class_platform.c) for the SAMA5D2 processor.

Note: The source code for the i.MX7D (ledRGB_imx_class_platform.c) and BCM2837 (ledRGB_rpi_class_platform.c) drivers can be downloaded from the GitHub repository of this book.

Listing 5-3: ledRGB_sam_class_platform.c

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/platform_device.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/leds.h>

#define PIO_SODR1_offset 0x50
#define PIO_CODR1_offset 0x54
#define PIO_CFGR1_offset 0x44
#define PIO_MSKR1_offset 0x40

#define PIO_PB0_MASK (1 << 0)
#define PIO_PB5_MASK (1 << 5)
#define PIO_PB6_MASK (1 << 6)
#define PIO_CFGR1_MASK (1 << 8)

#define PIO_MASK_ALL_LEDS (PIO_PB0_MASK | PIO_PB5_MASK | PIO_PB6_MASK)

struct led_dev
{
    u32 led_mask; /* different mask 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(PIO_MASK_ALL_LEDS, led->base + PIO_SODR1_offset);

    if (b != LED_OFF) /* LED ON */
        iowrite32(led->led_mask, led->base + PIO_CODR1_offset);
    else
        iowrite32(led->led_mask, led->base + PIO_SODR1_offset); /* LED OFF */
}

static int __init ledclass_probe(struct platform_device *pdev)
{
    void __iomem *g_ioremap_addr;
    struct device_node *child;
    struct resource *r;
    struct device *dev = &pdev->dev;
    int count, ret;
```

```
dev_info(dev, "platform_probe enter\n");

/* get your first memory resource from 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 your memory region */
g_ioremap_addr = devm_ioremap(dev, r->start, resource_size(r));
if (!g_ioremap_addr) {
    dev_err(dev, "ioremap failed \n");
    return -ENOMEM;
}

count = of_get_child_count(dev->of_node);
if (!count)
    return -EINVAL;

dev_info(dev, "there are %d nodes\n", count);

/* Enable all leds and set dir to output */
iowrite32(PIO_MASK_ALL_LEDS, g_ioremap_addr + PIO_MSKR1_offset);
iowrite32(PIO_CFGR1_MASK, g_ioremap_addr + PIO_CFGR1_offset);

/* Switch off all the leds */
iowrite32(PIO_MASK_ALL_LEDS, g_ioremap_addr + PIO_SODR1_offset);

for_each_child_of_node(dev->of_node, 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->base = g_ioremap_addr;
    of_property_read_string(child, "label", &cdev->name);

    if (strcmp(cdev->name, "red") == 0) {
        led_device->led_mask = PIO_PB6_MASK;
        led_device->cdev.default_trigger = "heartbeat";
    }
}
```

```
        }
        else if (strcmp(cdev->name,"green") == 0) {
            led_device->led_mask = PIO_PB5_MASK;
        }
        else if (strcmp(cdev->name,"blue") == 0) {
            led_device->led_mask = PIO_PB0_MASK;
        }
        else {
            dev_info(dev, "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);
        if (ret) {
            dev_err(dev, "failed to register the led %s\n", cdev->name);
            of_node_put(child);
            return ret;
        }
    }

    dev_info(dev, "leds_probe exit\n");

    return 0;
}

static int __exit ledclass_remove(struct platform_device *pdev)
{
    dev_info(&pdev->dev, "leds_remove enter\n");
    dev_info(&pdev->dev, "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_sam_class_platform.ko Demonstration

```
root@sama5d2-xplained:~# insmod ledRGB_sam_class_platform.ko /* load module, see the
led red blinking due to the heartbeat default trigger */
root@sama5d2-xplained:/sys/class/leds# ls /* check the devices under the LED class
*/
root@sama5d2-xplained:/sys/class/leds/red# echo 0 > brightness /* set led red OFF */
root@sama5d2-xplained:/sys/class/leds/red# echo 1 > brightness /* set led red ON */
root@sama5d2-xplained:/sys/class/leds/blue# echo 1 > brightness /* set led blue ON
and red OFF */
root@sama5d2-xplained:/sys/class/leds/green# echo 1 > brightness /* set led green ON
and blue OFF */
root@sama5d2-xplained:/sys/class/leds/green# ls /* check the sysfs entries under
green device */
root@sama5d2-xplained:/sys/class/leds/green# echo timer > trigger /* set the timer
trigger and see the led green blinking */
root@sama5d2-xplained:~# rmmod ledRGB_sam_class_platform.ko /* remove the module */
```

Platform Device Drivers in the 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 for a device, especially in applications where no two applications will require 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 an 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 of 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.

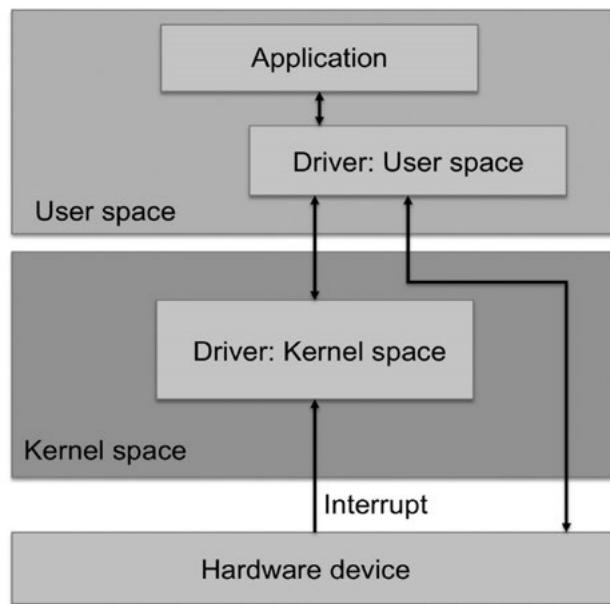
3. Kernel space driver advantages:

- 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 an user space driver might be designed. The application interfaces to the user space part of the driver. The user space part handles 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 that allows you to write user space drivers that are able to access device registers and handle interrupts.

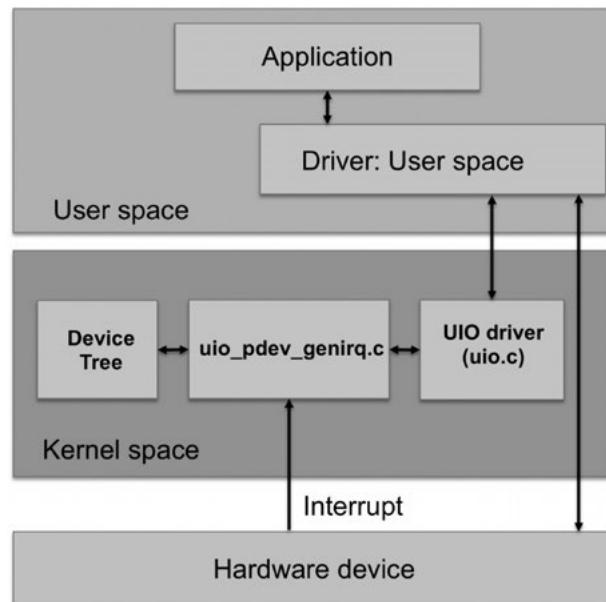
There are two distinct UIO device drivers provided by Linux under drivers/uio/ folder:

1. UIO driver (drivers/uio.c):

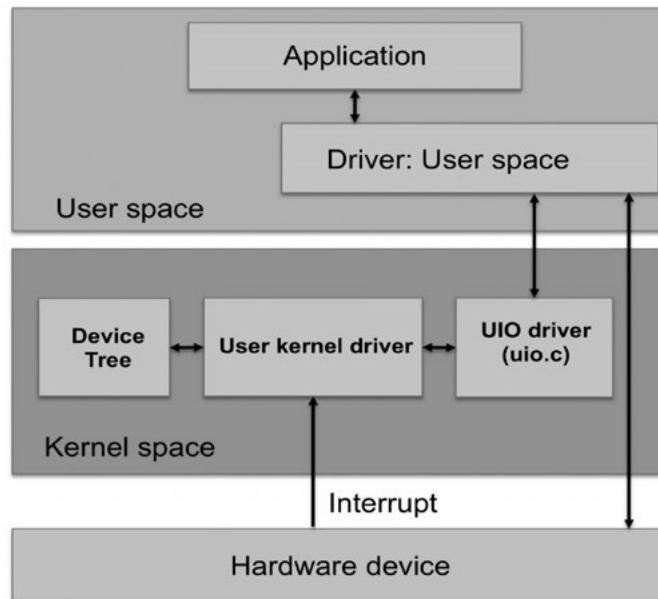
- The UIO driver creates files attributes in the sysfs describing the UIO device. It also maps the device memory into the process address space using its `mmap()` function.
- A minimal kernel space driver `uio_pdrv_genirq` ("UIO platform driver with generic interrupts"), or **user provided kernel driver** is required to setup 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 for the device needs to use "generic-uio" in its compatible property.
- The UIO platform device driver is configured from the device tree and registers an 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 an UIO device inside the `probe()` function. The device tree node for the device 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. Compile the new image and copy it to the tftp folder.

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 in the kernel creates file attributes in the sys filesystem describing the UIO device. 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 struct `uio_info` tells the framework the details of your driver. Some of the members are required, others are optional. These are some of the struct `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 struct `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 userspace by writing to /dev/uioX, you can implement this function. The parameter irq_on will be 0 to disable interrupts and 1 to enable them.

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 struct uio_mem structure in the mem[] array. Here's a description of the fields of the struct 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 kmalloc()). 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 struct uio_info 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 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 an **UIO user space driver** that controls one of the LEDs used in the previous lab. The main function of an 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 struct `uio_info` with these parameters. You will also register the UIO device within the `probe()` function of the kernel driver.

LAB 5.4 DT for the i.MX7D, SAMA5D2 and BCM2837 Processors

In this lab 5.4, you will keep the same DT GPIO multiplexing of the previous lab 5.3, as you will use the same processor's pad to control the LED. 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 children node inside the main node.

For the **MCIMX7D-SABRE** board modify the device tree file `imx7d-sdb.dts` adding the next code in bold. The i.MX7D GPIO memory map is shown in the section 8.3.5 GPIO Memory Map/Register Definition of the i.MX 7Dual Applications Processor Reference Manual, Rev. 0.1, 08/2016. The **0x30200000** base address of the `reg` property is the GPIO data register (GPIO1_DR) address.

```
/ {  
    model = "Freescale i.MX7 SabreSD Board";  
    compatible = "fsl,imx7d-sdb", "fsl,imx7d";  
  
    memory {  
        reg = <0x80000000 0x80000000>;  
    };  
  
    [...]  
  
    UIO {  
        compatible = "arrow,UIO";  
        reg = <0x30200000 0x60000>;  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_gpio_leds &pinctrl_gpio_led>;  
    };  
    [...]
```

For the **SAMA5D2B-XULT** board modify the device tree file `at91-sama5d2_xplained_common.dtsi` adding the next code in bold. The **0xFC038000** base address of the `reg` property is the PIO Mask

Register (PIO_MSKR0) address. See the section 34.7.1 PIO Mask Register of the SAMA5D2 Series Datasheet.

```
/ {  
    model = "Atmel SAMA5D2 Xplained";  
    compatible = "atmel,sama5d2-xplained", "atmel,sama5d2", "atmel,sama5";  
  
    chosen {  
        stdout-path = "serial0:115200n8";  
    };  
  
    [...]  
  
    UIO {  
        compatible = "arrow,UIO";  
        reg = <0xFC038000 0x4000>;  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_led_gpio_default>;  
    };  
  
    [...]  
};
```

For the **Raspberry Pi 3 Model B** board modify the device tree file bcm2710-rpi-3-b.dts adding the next code in bold. The **0x7e200000** base address of the reg property is the GPFSEL0 register address. See the section 6.1 Register View of the BCM2835 ARM Peripherals guide.

```
/ {  
    model = "Raspberry Pi 3 Model B";  
};  
  
&soc {  
  
    [...]  
  
    expgpio: expgpio {  
        compatible = "brcm,bcm2835-expgpio";  
        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 struct uio_info:

```
static struct uio_info the_uio_info;
```

3. In the probe() function, the platform_get_resource() function gets the struct resource filled with the values described by the DT reg property. The devm_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);

/* ioremap your memory region and get virtual address */
g_ioremap_addr = devm_ioremap(dev, r->start, resource_size(r));
```

4. Initialize the struct 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 struct 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. Build the modified device tree, and load it to the target processor.

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, "0x%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 `the_uio_info.mem[0].internal_addr` variable.

See in the next **Listing 5-4** and **Listing 5-5** the "LED UIO platform" kernel driver source code (`led_sam_UIO_platform.c`) and the "LED UIO platform" user space driver source code for the SAMA5D2 processor (`UIO_app.c`).

Note: The source code for the i.MX7D (`led_imx_UIO_platform.c` & `UIO_app.c`) and BCM2837 (`led_rpi_UIO_platform.c` & `UIO_app.c`) kernel and user space drivers can be downloaded from the GitHub repository of this book.

Listing 5-4: led_sam_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 your first memory resource from 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 your memory region and get 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;
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 */

/* 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 GPIO4_GDIR_offset 0x04
#define GPIO_DIR_MASK 1<<29
#define GPIO_DATA_MASK 1<<29

#define PIO_SODR1_offset 0x50
#define PIO_CODR1_offset 0x54
#define PIO_CFGR1_offset 0x44
#define PIO_MSKR1_offset 0x40

#define PIO_PB0_MASK (1 << 0)
#define PIO_PB5_MASK (1 << 5)
#define PIO_PB6_MASK (1 << 6)
#define PIO_CFGR1_MASK (1 << 8)

#define PIO_MASK_ALL_LEDS (PIO_PB0_MASK | PIO_PB5_MASK | PIO_PB6_MASK)

#define UIO_SIZE "/sys/class/uio/uio0/maps/map0/size"

int main()
{
    int ret, devuio_fd;
    unsigned int uio_size;
    void *temp;
    void *demo_driver_map;
    char sendstring[BUFFER_LENGTH];
    char *led_on = "on";
    char *led_off = "off";
    char *Exit = "exit";

    printf("Starting led example\n");
    devuio_fd = open("/dev/uio0", O_RDWR | O_SYNC);
    if (devuio_fd < 0){
        perror("Failed to open the device");
        exit(EXIT_FAILURE);
    }
```

```
/* read the size that has to be mapped */
FILE *size_fp = fopen(UIO_SIZE, "r");
fscanf(size_fp, "0x%08X", &uio_size);
fclose(size_fp);

/* do the mapping */
demo_driver_map = mmap(NULL, uio_size, PROT_READ|PROT_WRITE,
                      MAP_SHARED, devuio_fd, 0);

if(demo_driver_map == MAP_FAILED) {
    perror("devuio mmap");
    close(devuio_fd);
    exit(EXIT_FAILURE);
}

temp = demo_driver_map + PIO_MSKR1_offset;
*(int *)temp |= PIO_MASK_ALL_LEDS;

/* select output */
temp = demo_driver_map + PIO_CFGR1_offset;
*(int *)temp |= PIO_CFGR1_MASK;

/* clear all the leds */
temp = demo_driver_map + PIO_SODR1_offset;
*(int *)temp |= PIO_MASK_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 + PIO_CODR1_offset;
        *(int *)temp |= PIO_PB0_MASK;
    }
    else if(strncmp(led_off, sendstring, 2) == 0)
    {
        temp = demo_driver_map + PIO_SODR1_offset;
        *(int *)temp |= PIO_PB0_MASK;
    }
    else if(strncmp(Exit, sendstring, 4) == 0)
        printf("Exit application\n");
    else {
        printf("Bad value\n");
        temp = demo_driver_map + PIO_SODR1_offset;
        *(int *)temp |= PIO_PB0_MASK;
        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_sam_UIO_platform.ko with UIO_app Demonstration

```
root@sama5d2-xplained:~# insmod led_sam_UIO_platform.ko /* load the module */
root@sama5d2-xplained:~# ./UIO_app /* start your application to turn on/off the blue led */
root@sama5d2-xplained:~# ./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

root@sama5d2-xplained:~# rmmod led_sam_UIO_platform.ko /* remove the module */
```

<http://www.rejoiceblog.com/>

6

I2C Client Drivers

I2C (pronounce: I squared C) is a protocol developed by Philips. It is a slow two-wire protocol (variable speed, up to 400 kHz), with a high speed extension (3.4 MHz). It provides an inexpensive bus for connecting many types of devices with infrequent or low bandwidth communications needs. I2C is widely used with embedded systems. Some systems use variants that don't meet branding requirements, and so are not advertised as being I2C.

SMBus (System Management Bus) is based on the I2C protocol, and is mostly a subset of I2C protocols and signaling. Many I2C devices will work on a SMBus, but some SMBus protocols add semantics beyond what is required to achieve I2C branding. Modern PC mainboards rely on SMBus. The most common devices connected through SMBus are RAM modules configured using I2C EEPROMs, and hardware monitoring chips. 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);
```

These routines read and write some bytes from/to a client. The client contains the I2C address, so you do not have to include it. The second parameter contains the bytes to read/write, the third the number of bytes to read/write (must be less than the length of the buffer, also should be less than 64k since msg.len is u16.). Returned is the actual number of bytes read/written.

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

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 struct i2c_msg 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 a 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 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.

The Linux I2C Subsystem

The Linux I2C subsystem is based in the Linux device model and is composed of several drivers:

1. The **I2C bus core** of the I2C subsystem is located in the i2c-core.c file under drivers/i2c/ directory. The I2C core in the device model is a collection of code that provides interface support between an individual client driver and some I2C bus masters such as the i.MX7D I2C controllers. 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 declare the I2C struct bus_type structure. The I2C core API is a set of functions used for an **I2C client device driver** to send/receive data to/from a device connected to an I2C bus.
2. The **I2C controller drivers** are located under drivers/i2c/busses/ directory. 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 specific I2C controller hardware I/O addresses. There is a specific code for each I2C controller of the processor. This controller specific functions are called by the 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 (for example, the i2c-imx.c) you have to declare a private structure that includes a struct i2c_adapter variable.

```
struct imx_i2c_struct {
    struct i2c_adapter    adapter;
    struct clk            *clk;
    void __iomem          *base;
    wait_queue_head_t     queue;
    unsigned long          i2csr;
    unsigned int           disable_delay;
    int                  stopped;
    unsigned int          ifdr; /* IMX_I2C_IFDR */
    unsigned int           cur_clk;
    unsigned int           bitrate;
    const struct imx_i2c_hwdata *hwdata;
    struct i2c_bus_recovery_info rinfo;

    struct pinctrl *pinctrl;
    struct pinctrl_state *pinctrl_pins_default;
    struct pinctrl_state *pinctrl_pins_gpio;

    struct imx_i2c_dma   *dma;
};
```

In the probe() function, you initialize this adapter structure for each I2C controller that has been probed:

```
/* Setup i2c_imx driver structure */
strcpy(i2c_imx->adapter.name, pdev->name, sizeof(i2c_imx->adapter.name));
i2c_imx->adapter.owner        = THIS_MODULE;
i2c_imx->adapter.algo         = &i2c_imx_algo;
i2c_imx->adapter.dev.parent  = &pdev->dev;
i2c_imx->adapter.nr           = pdev->id;
i2c_imx->adapter.dev.of_node  = pdev->dev.of_node;
i2c_imx->base
```

The i2c_imx_algo structure includes a pointer variable to the i2c_imx_xfer() function, that contains the specific code that will write/read the registers of the I2C hardware controller:

```
static struct i2c_algorithm i2c_imx_algo = {
    .master_xfer  = i2c_imx_xfer,
    .functionality = i2c_imx_func,
};
```

Finally, in the probe() function, each I2C controller is added to the I2C bus core by calling the i2c_add_numbered_adapter() function (located in drivers/i2c/i2c-core.c):

```
i2c_add_numbered_adapter(&i2c_imx->adapter);
```

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 declared in `drivers/i2c/i2c-core.c`, you can see that this function is calling 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 is calling the `i2c_smbus_xfer_emulated()` function, which in turn calls `i2c_adapter.algo->master_xfer()` function following the next sequence:

`i2c_smbus_xfer_emulated() -> i2c_transfer() -> __i2c_transfer() -> adap->algo->master_xfer()`

This function `master_xfer()` was initialized to the `i2c_imx_xfer()` function in your I2C driver controller, so it is calling to the specific code that is controlling your I2C controller registers:

```
s32 i2c_smbus_xfer(struct i2c_adapter *adapter, u16 addr,
                     unsigned short flags, char read_write, u8 command,
                     int protocol, union i2c_smbus_data *data)
{
    unsigned long orig_jiffies;
    int try;
    s32 res;

    flags &= I2C_M_TEN | I2C_CLIENT_PEC | I2C_CLIENT_SCCB;

    if (adapter->algo->smbus_xfer) {
        i2c_lock_adapter(adapter);

        /* Retry automatically on arbitration loss */
        orig_jiffies = jiffies;
        for (res = 0, try = 0; try <= adapter->retries; try++) {
            res = adapter->algo->smbus_xfer(adapter, addr, flags,
                                              read_write, command,
                                              protocol, data);
            if (res != -EAGAIN)
```

```
        break;
    if (time_after(jiffies,
                    orig_jiffies + adapter->timeout))
        break;
}
i2c_unlock_adapter(adapter);

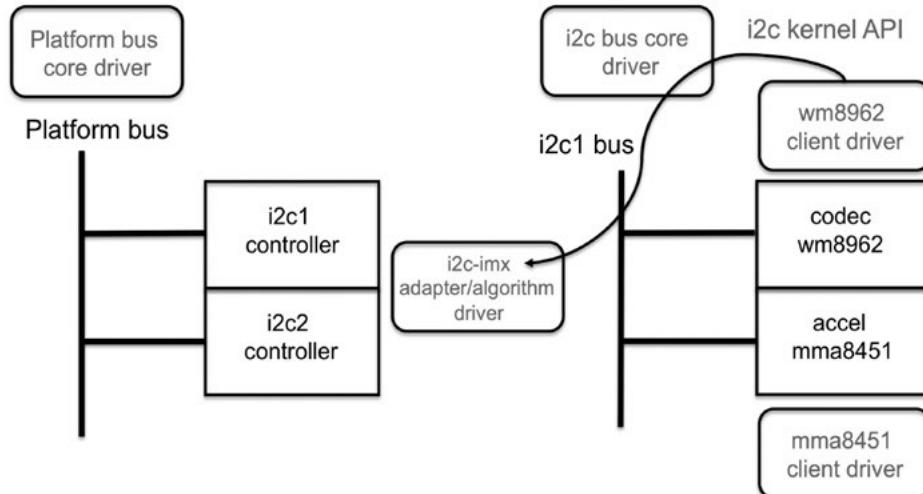
if (res != -EOPNOTSUPP || !adapter->algo->master_xfer)
    return res;
/*
 * Fall back to i2c_smbus_xfer_emulated if the adapter doesn't
 * implement native support for the SMBus operation.
 */
}

return i2c_smbus_xfer_emulated(adapter, addr, flags, read_write,
                               command, protocol, data);
}
```

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**. In the i.MX7D processor, the `i2c-imx.c` controller driver registers itself to the **platform bus core**. The I2C client drivers are registered by themselves to the **I2C bus core**. You can see in the figure the next flow of function calls:

I2C client driver-> I2C bus core driver -> I2C controller driver:

`i2c_smbus_write_byte_data()` -> `i2c_smbus_xfer()` -> `i2c_imx_xfer()`



Writing I2C Client Drivers

You will focus now in the writing of I2C client drivers. In this and in successive chapters, you will develop several I2C client drivers that control I/O expanders, DACs, accelerometers and Multidisplay LED controllers. In the next 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 a struct `i2c_driver` structure (inherited from the struct `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 a struct `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/unregister** the driver. They are included in the `init()/exit()` kernel module 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);
```

The `module_i2c_driver()` macro can be used to simplify the code above:

```
module_i2c_driver(ioaccel_driver);
```

In your device driver create an array of struct of_device_id structures where you specify .compatible strings that should store the same value of the DT device node's compatible property. The struct of_device_id is defined in include/linux/mod_devicetable.h as:

```
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 struct i2c_driver is a pointer to the array of struct 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 of initializing the device with the configuration values obtained from the matching DT device node and also to register the device to the appropriate kernel framework.

In your I2C device driver, you have also to define an array of struct i2c_device_id structures:

```
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)
```

You can use id->driver_data (which is unique to each device), for specific device data. For example, for the "mma8451" device, the driver_data will be 1.

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 device by id table.

Declaration of I2C Devices in Device Tree

In the device tree, the I2C controller device is typically declared in the .dtsi file that describes the processor (for i.MX7D see arch/arm/boot/dts/imx7s.dtsi). The DT I2C controller definition is normally declared with status = "disabled". For example, in the imx7s.dtsi file there are declared four DT I2C controller devices that will be registered to the I2C bus core through the of_platform_populate() function. For the i.MX7D, the i2-imx.c driver will register itself to the I2C bus core using the module_i2c_driver() function ; the probe() function will be called four times (one for each compatible = "fsl,imx21-i2c" matching) initializing a struct i2c_adapter structure for each controller and registering it with the I2C bus core using the i2c_add_numbered_adapter() function. See below the declaration of the i.MX7D DT I2C controller nodes:

```
i2c1: i2c@30a20000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx7d-i2c", "fsl,imx21-i2c";
    reg = <0x30a20000 0x10000>;
    interrupts = <GIC_SPI 35 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clks IMX7D_I2C1_ROOT_CLK>;
    status = "disabled";
};

i2c2: i2c@30a30000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx7d-i2c", "fsl,imx21-i2c";
    reg = <0x30a30000 0x10000>;
    interrupts = <GIC_SPI 36 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clks IMX7D_I2C2_ROOT_CLK>;
    status = "disabled";
};

i2c3: i2c@30a40000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx7d-i2c", "fsl,imx21-i2c";
    reg = <0x30a40000 0x10000>;
    interrupts = <GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clks IMX7D_I2C3_ROOT_CLK>;
    status = "disabled";
};

i2c4: i2c@30a50000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx7d-i2c", "fsl,imx21-i2c";
```

```
reg = <0x30a50000 0x10000>;
interrupts = <GIC_SPI 38 IRQ_TYPE_LEVEL_HIGH>;
clocks = <&clks IMX7D_I2C4_ROOT_CLK>;
status = "disabled";
};
```

The device tree declaration of I2C devices is done as sub-nodes of the master controller. At the board/platform level (arch/arm/boot/dts/imx7d-sdb.dts):

- The I2C controller device is enabled (status = "okay").
- The I2C bus frequency is defined, using the clock-frequency property.
- The I2C devices on the bus are described as children of the I2C controller node, where the reg property provides the I2C slave address on the bus.
- In the I2C device node check that the compatible property matchs with one of the driver's of_device_id compatible strings.

Find in the DT imx7d-sdb.dts file the i2c4 controller node declaration. The i2c4 controller is enabled writing "okay" to the status property. In the codec sub-node device declaration the reg property provides the I2C address of the wm8960 device.

```
&i2c4 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c4>;
    status = "okay";

    codec: wm8960@1a {
        compatible = "wlf,wm8960";
        reg = <0x1a>;
        clocks = <&clks IMX7D_AUDIO_MCLK_ROOT_CLK>;
        clock-names = "mclk";
        wlf,shared-1rc1k;
    };
};
```

The pinctrl-0 property inside the i2c4 node points to the pinctrl_i2c4 pin function node, where the i2c4 controller pads are being multiplexed with I2C functionality:

```
pinctrl_i2c4: i2c4grp {
    fsl,pins = <
        MX7D_PAD_SAI1_RX_BCLK__I2C4_SDA      0x4000007f
        MX7D_PAD_SAI1_RX_SYNC__I2C4_SCL      0x4000007f
    >;
};
```

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 several PCF8574 I/O expander devices connected to the I2C bus. You can use one of the multiple boards based on this device to develop this lab, for example, the next one: <https://www.waveshare.com/pcf8574-io-expansion-board.htm>.

LAB 6.1 Hardware Description for the i.MX7D Processor

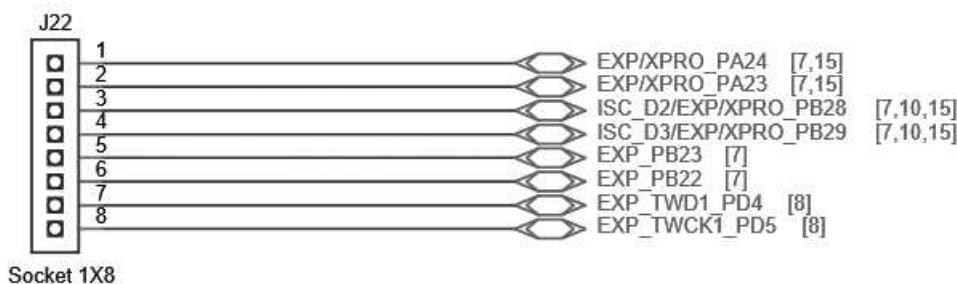
In this lab, you will use the I2C pins of the MCIMX7D-SABRE mikroBUS™ to connect to the PCF8574 I/O expander board.

Go to page 20 of the MCIMX7D-SABRE schematic to see the MikroBUS connector and look for the SDA and the SCL pins. Connect these pins to the equivalent ones of the PCF8574 I/O expander board. Connect also VCC 3.3V and GND between the two boards.

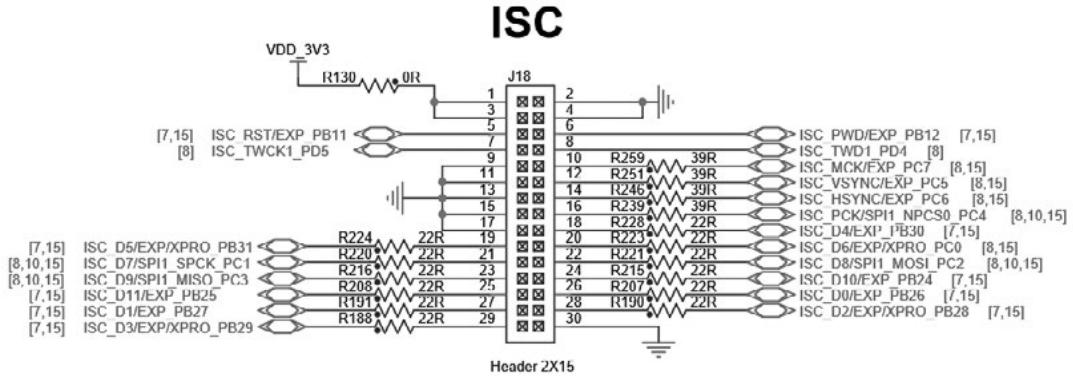
LAB 6.1 Hardware Description for the SAMA5D2 Processor

For the SAMA5D2 processor, open the SAMA5D2B-XULT schematic and look for connectors on board with pins that provide I2C signals. The SAMA5D2B-XULT board has five 8-pin, one 6-pin, one 10-pin and one 36-pin headers (J7, J8, J9, J16, J17, J20, J21, J22) that enable the PIO connection of various expansion cards. These headers' physical and electrical implementation match the Arduino R3 extension ("shields") system.

You can access to the I2C signals using the J22 header, as shown in the next figure:

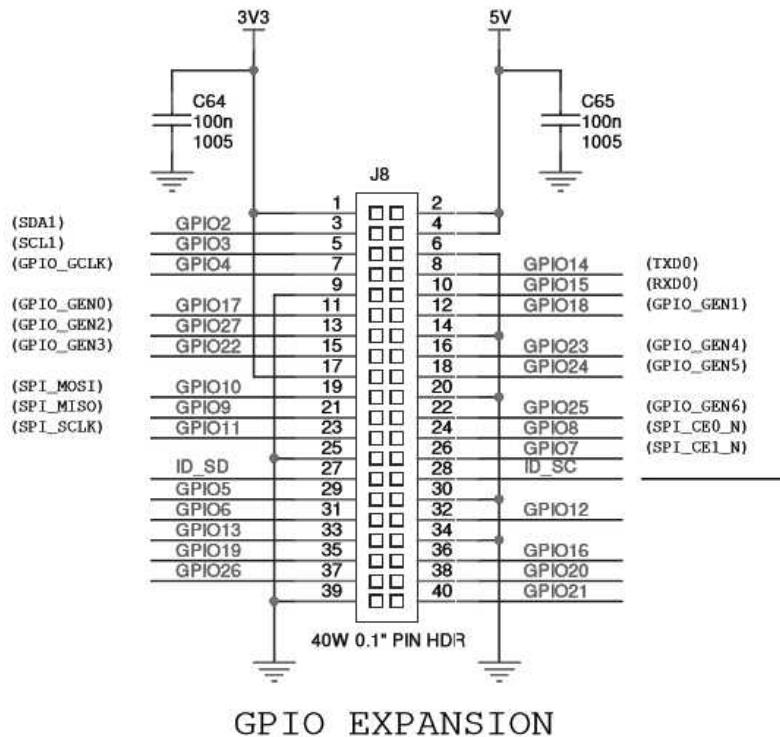


You can also get the same TWCK1, and TWD1 I2C signals from the ISC J18 header, as shown in the following figure. Having the same signals on two different connectors simplifies the connection to several PCF8574 boards. You can also get 3V3 and GND signals from this J18 connector.



LAB 6.1 Hardware Description for the BCM2837 Processor

For the BCM2837 processor, you will use the GPIO expansion connector to obtain the I2C signals. Open the Raspberry-Pi-3B-V1.2-Schematics to see the connector. The GPIO2 and GPIO3 pins will be used to get the SDA1 and SCL1 signals.



GPIO EXPANSION

LAB 6.1 Device Tree for the i.MX7D Processor

From the MCIMX7D-SABRE mikroBUS™ socket, you see that MKBUS_I2C_SCL pin connects to the I2C3_SCL pad of the i.MX7D processor, and the MKBUS_I2C_SDA pin to the I2C3_SDA pad. You have to configure these pads as I2C signals. To look for the macro that assigns the required I2C functionality open the imx7d-pinfunc.h file under linux/arch/arm/boot/dts/ directory and find the next macros:

```
#define MX7D_PAD_I2C3_SDA_I2C3_SDA 0x015C 0x03CC 0x05E8 0x0 0x2
#define MX7D_PAD_I2C3_SCL_I2C3_SCL 0x0158 0x03C8 0x05E4 0x0 0x2
```

Now, you can modify the device tree file imx7d-sdb.dts adding the ioexp@38 and the ioexp@39 sub-nodes inside the i2c3 controller master node. The i2c3 controller is enabled writing "okay" to the status property. The clock-frequency property is set to 100 KHz and the pinctrl-0 property of the master node points to the pinctrl_i2c3 pin configuration node, where the I2C3_SDA and I2C3_SCL pads are multiplexed as I2C signals. In the sub-nodes devices declaration the reg property provides the I2C addresses of the two PCF8574 I/O expanders connected to the I2C bus.

```
&i2c3 {  
    clock-frequency = <100000>;  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_i2c3>;  
    status = "okay";  
  
    ioexp@38 {  
        compatible = "arrow,ioexp";  
        reg = <0x38>;  
    };  
    ioexp@39 {  
        compatible = "arrow,ioexp";  
        reg = <0x39>;  
    };  
  
    sii902x: sii902x@39 {  
        compatible = "SiI,sii902x";  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_sii902x>;  
        interrupt-parent = <&gpio2>;  
        interrupts = <13 IRQ_TYPE_EDGE_FALLING>;  
        mode_str ="1280x720M@60";  
        bits-per-pixel = <16>;  
        reg = <0x39>;  
        status = "okay";  
    };  
  
    [...]  
};
```

See below the pinctrl_i2c3 pin configuration node, where I2C controller pads are multiplexed as I2C signals:

```
pinctrl_i2c3: i2c3grp {  
    fsl,pins = <  
        MX7D_PAD_I2C3_SDA__I2C3_SDA 0x4000007f  
        MX7D_PAD_I2C3_SCL__I2C3_SCL 0x4000007f  
    >;  
};
```

LAB 6.1 Device Tree for the SAMA5D2 Processor

Open the SAMA5D2B-XULT board schematic and look for the J22 connector. You see that EXP_TWCK1_PD5 pin connects to the PD5 pad of the SAMA5D2 processor, and the EXP_TWD1_PD4 pin to the PD4 pad. You have to configure the PD5, and PD4 pads as I2C signals. To look for the macro that assigns the required I2C functionality go to the sama5d2-pinfuc.h file under linux/arch/arm/boot/dts/ directory and find the next macros below. See below in bold the needed macros to configure the PD4 and PD5 pads as I2C signals:

```
#define PIN_PD4 100
#define PIN_PD4_GPIO PINMUX_PIN(PIN_PD4, 0, 0)
#define PIN_PD4_TWD1 PINMUX_PIN(PIN_PD4, 1, 2)
#define PIN_PD4_URXD2 PINMUX_PIN(PIN_PD4, 2, 1)
#define PIN_PD4_GCOL PINMUX_PIN(PIN_PD4, 4, 2)
#define PIN_PD4_ISC_D10 PINMUX_PIN(PIN_PD4, 5, 2)
#define PIN_PD4_NCS0 PINMUX_PIN(PIN_PD4, 6, 2)
#define PIN_PD5 101
#define PIN_PD5_GPIO PINMUX_PIN(PIN_PD5, 0, 0)
#define PIN_PD5_TWCK1 PINMUX_PIN(PIN_PD5, 1, 2)
#define PIN_PD5_UTXD2 PINMUX_PIN(PIN_PD5, 2, 1)
#define PIN_PD5_GRX2 PINMUX_PIN(PIN_PD5, 4, 2)
#define PIN_PD5_ISC_D9 PINMUX_PIN(PIN_PD5, 5, 2)
#define PIN_PD5_NCS1 PINMUX_PIN(PIN_PD5, 6, 2)
```

In the processor's data-sheet, you can see that the PD4 and PD5 pads can be used for several functionalities. These pads functions are shown in the table below:

J6	H6	-	VDDANA	GPIO_AD	PD4	I/O	PTC_X1	-	A	TWD1	I/O	2	PIO, I, PU, ST
									B	URXD2	I	1	
									D	GCOL	I	2	
									E	ISC_D10	I	2	
									F	NCS0	O	2	
									A	TWCK1	I/O	2	
J4	H1	-	VDDANA	GPIO_AD	PD5	I/O	PTC_X2	-	B	UTXD2	O	1	PIO, I, PU, ST
									D	GRX2	I	2	
									E	ISC_D9	I	2	
									F	NCS1	O	2	
									A	TWCK1	I/O	2	
									B	UTXD2	O	1	

In the macro for the PD4 pin, the last two numbers set the function of the pad and the IO Set of the signal. For example, the PIN_PD4_TWD1 has function number 1 (A) and TWD1 signal correspond to the IO Set 2. The PIN_PD5_TWCK1 has function number 1 (A) and TWCK1 signal correspond to the IO Set 2.

Note: I/Os for each peripheral are grouped into IO sets, listed in the column "IO Set" in the pinout tables. For all peripherals, it is mandatory to use I/Os that belong to the same IO set. The timings are not guaranteed when IOs from different IO sets are mixed.

Open and modify the device tree file at91-sama5d2_xplained_common.dtsi adding the ioexp@38 and the ioexp@39 sub-nodes inside the i2c1 controller master node. The i2c1 controller is enabled writing "okay" to the status property. The pinctrl-0 property of the master node points to the pinctrl_i2c1_default pin configuration node, where the PD4 and PD5 pads are multiplexed as I2C signals. In the sub-nodes devices declaration the reg property provides the I2C addresses of the two PCF8574 I/O expanders connected to the I2C bus.

```
i2c1: i2c@fc028000 {
    dmas = <0>, <0>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c1_default>;
    status = "okay";

    [...]

    ioexp@38 {
        compatible = "arrow,ioexp";
        reg = <0x38>;
    };
    ioexp@39 {
        compatible = "arrow,ioexp";
        reg = <0x39>;
    };
    [...]

    at24@54 {
        compatible = "atmel,24c02";
        reg = <0x54>;
        pagesize = <16>;
    };
}
```

See below the pinctrl_i2c1_default pin configuration node, where I2C controller pads are multiplexed as I2C signals:

```
pinctrl_i2c1_default: i2c1_default {
    pinmux = <PIN_PD4__TWD1>,
              <PIN_PD5__TWCK1>;
    bias-disable;
};
```

LAB 6.1 Device Tree for the BCM2837 Processor

Open Raspberry-Pi-3B-V1.2-Schematics and find the GPIO EXPANSION connector. You see that GPIO2 pin connects to the GPIO2 pad of the BCM2837 processor, and the GPIO3 pin to the GPIO3 pad.

Open and modify the device tree file `bcm2710-rpi-3-b.dts` adding the `ioexp@38` and the `ioexp@39` sub-nodes inside the `i2c1` controller master node. The `i2c1` controller is enabled writing "okay" to the `status` property. The `pinctrl-0` property of the master node points to the `i2c1_pins` pin configuration node, where the GPIO2 and GPIO3 pads are multiplexed as I2C signals. In the sub-nodes devices declaration the `reg` property provides the I2C addresses of the two PCF8574 I/O expanders connected to the I2C bus.

```
&i2c1 {  
    pinctrl-names = "default";  
    pinctrl-0 = <&i2c1_pins>;  
    clock-frequency = <1000000>;  
    status = "okay";  
  
    [...]  
  
    ioexp@38 {  
        compatible = "arrow,ioexp";  
        reg = <0x38>;  
    };  
  
    ioexp@39 {  
        compatible = "arrow,ioexp";  
        reg = <0x39>;  
    };  
};
```

See below the `i2c1_pins` pin configuration node, where 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` file under `Documentation/devicetree/bindings/pinctrl/` directory.

Open the BCM2835 ARM Peripherals guide and look for 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 need to create a private structure that will store the I2C I/O device specific information. In this driver, the first field of the private structure is a struct i2c_client structure used to handle the I2C device. The second field of the private structure is a struct miscdevice structure. The misc subsystem will automatically handle the open() function for you. Inside the automatically created open() function, it will tie your created struct miscdevice to the private struct ioexp_dev for the file that's being opened. In this way in your write/read kernel callback functions you can recover the struct miscdevice structure, which will allow you to get access to the struct i2c_client that is included in the private struct ioexp_dev structure. Once you get the struct i2c_client structure you can read/write each I2C specific device using the SMBus functions. The last field of the private structure is a char array that will hold the name of the I2C I/O device.

```
struct ioexp_dev {
    struct i2c_client * client;
    struct miscdevice ioexp_miscdevice;
    char name[8]; /* ioexpXX */
};
```

3. Create a struct file_operations structure to define which driver's 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 allocates the private structure with the devm_kzalloc() function. Initialize each misc device and register it with the kernel using the misc_register() function. The i2c_set_clientdata() function is attaching each allocated private structure to the struct i2c_client one, which will allow you to access your private data structure in other parts of the driver, for example, you will recover the private structure in each remove() function call (called two times, once per each device attached to the bus) 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 new structure representing device */  
    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 device/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 misc device */  
    return misc_register(&ioexp->ioexp_miscdevice);  
  
    return 0;  
}
```

5. Create the ioexp_write_file() kernel callback function, that gets called whenever an 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 struct ioexp_dev structure. However, as the struct miscdevice structure is accessible through file->private_data, and is a member of the struct ioexp_dev structure, you can use the container_of() macro to compute the address of your private structure and recover the struct i2c_client 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 and you will write it to the I2C ioexp device using the i2c_smbus_write_byte() SMBus function. You will also write an ioexp_read_file() kernel callback function, that reads the ioexp device input and sends the value to user space. See below an extract 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 struct i2c_device_id structures:

```
static const struct i2c_device_id i2c_ids[] = {
    { .name = "ioexp", },
    { }
};
MODULE_DEVICE_TABLE(i2c, i2c_ids);
```

8. Add a struct 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. Build the modified device tree, and load it to the target processor.

See in the next **Listing 6-1** the "I2C I/O expander device" driver source code (io_imx_expander.c) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (io_sam_expander.c) and BCM2837 (io_rpi_expander.c) drivers can be downloaded from the GitHub repository of this book.

Listing 6-1: io_imx_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_misdevice);

    /* store IO expander input to expval int variable */
    expval = i2c_smbus_read_byte(ioexp->client);
    if (expval < 0)
        return -EFAULT;

    /*
     * converts expval int value into a char string
     * For instance 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;
}

/* Writing 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 two times */
static int ioexp_probe(struct i2c_client * client,
                      const struct i2c_device_id * id)
```

```
{  
    static int counter = 0;  
  
    struct ioexp_dev * ioexp;  
  
    /* Allocate new 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 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@arrownetworks.com>");
MODULE_DESCRIPTION("This is a driver that controls several I2C IO expanders");
```

io_imx_expander.ko Demonstration

```
root@imx7dsabresd:~# insmod io_imx_expander.ko /* load module, probe() is called
twice */
root@imx7dsabresd:~# ls -l /dev/ioexp* /* find ioexp00 and ioexp01 devices */
root@imx7dsabresd:~# echo 0 > /dev/ioexp00 /* set all the outputs to 0 */
root@imx7dsabresd:~# echo 255 > /dev/ioexp01 /* set all the outputs to 1 */
root@imx7dsabresd:~# rmmod io_imx_expander.ko /* remove module, remove() is called
twice */
```

The Sysfs Filesystem

Sysfs is a virtual file system 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 to user space. Various programs use sysfs: udev, mdev, lsusb, lspci.

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 the list of buses.
- `/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 kernel device tree, which is a hierarchy of struct 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 (cpu) 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 consists 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, it allows sysfs representation. The struct kobject structure is defined 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;  
  
    [...]  
}
```

For every kobject that is registered with the system, a directory is created for it in sysfs. That directory is created as a subdirectory of the kobject's parent, expressing internal object hierarchies to user space. Inside the struct kobject there is a pointer **Ktype** to the struct kobj_type structure. The struct Kobj_type controls what happens when the kobject is created and deleted; it is defined as:

```
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 struct Kobj_type also contains attributes that can be exported to sysfs directories (directories are created with kobject_create_and_add() function) as files (files are created with sysfs_create_file()). These attributes files can be "read/written" via "show/store" methods using the struct sysfs_ops structure that is pointed via the sysfs_ops pointer variable (included in the struct kobj_type structure).

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

Attributes can be exported for kobjects in the form of regular files in the filesystem. Sysfs forwards file I/O operations to methods defined for the attributes, providing a means to read and write kernel attributes.

An attribute definition is simply:

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

int sysfs_create_file(struct kobject * kobj, const struct attribute * attr);
void sysfs_remove_file(struct kobject * kobj, const struct attribute * attr);
```

A bare attribute contains no means to read or write the value of the attribute. 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 driver model defines 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);
};

int device_create_file(struct device *, const struct device_attribute *);
void device_remove_file(struct device *, const struct device_attribute *);
```

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 the device using the next 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);
```

You can add/remove a group of "sysfs attribute files" for the device 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);
```

For example:

You have two structures of type struct 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 the group of sysfs entries 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);
```

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 for 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.

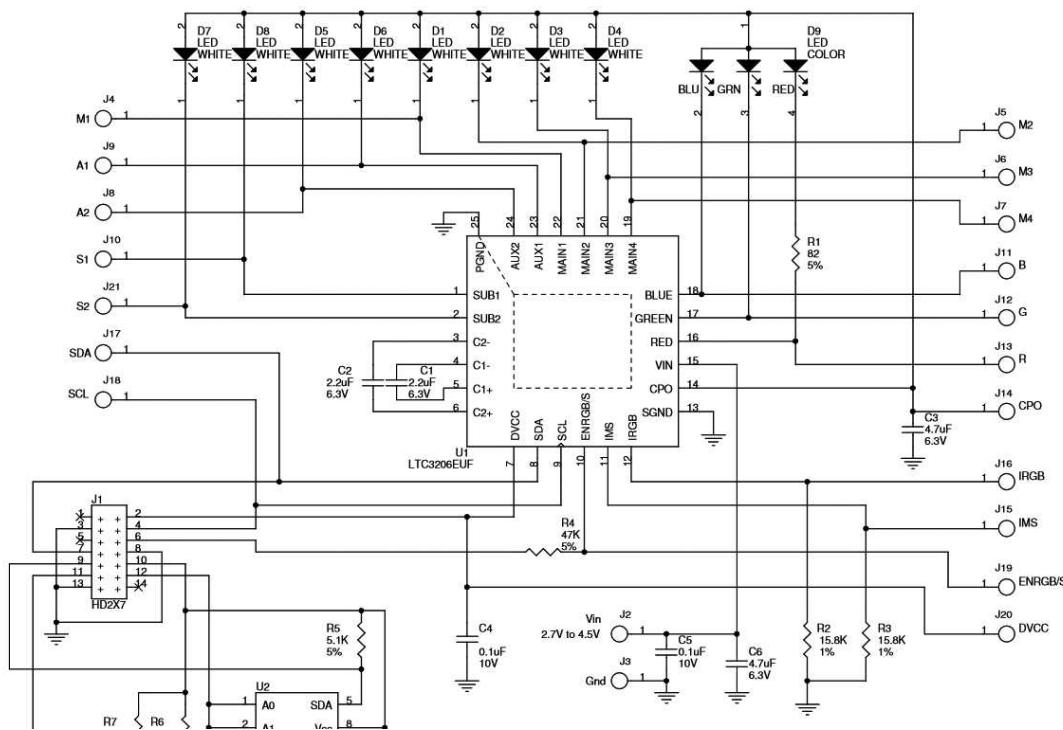
If bit A2 is set to 1 (SUB display control), then ENRGB/S will have no effect on the RGB display. Likewise, if bit A2 is set to 0 (RGB display control), then ENRGB/S will have no effect on the SUB display.

If the ENRGB/S pin is not used, it should be connected to DVCC. It should not be grounded or left floating.

In the Pag.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 used processor. Connect the processor's SDA signal to the pin 7 of the J1 connector and the processor's SCL signal to the pin 4 of the J1 connector. Connect 3.3V between the processor's board and the J20 DVCC connector. Do not forget to connect GND between DC749A and the processor's board. If you do not want to enable the ENRGB/S pin connect it to DVCC.

The DC749A board schematic is shown below:



LAB 6.2 Hardware Description for the i.MX7D Processor

In this lab, you will use the I2C pins of the MCIMX7D-SABRE mikroBUS™ to connect to the DC749A - Demo Board.

Go to the pag.20 of the MCIMX7D-SABRE schematic to see the MikroBUS connector and look for the SDA and the SCL pins. Connect the MikroBUS SDA pin to the pin 7 (SDA) of the DC749A J1 connector and the MikroBUS SCL pin to the pin 4 (SCL) of the DC749A J1 connector. Connect the 3.3V MikroBUS pin to the DC749A Vin J2 pin and to the DC749A J20 DVCC connector. Connect the MikroBUS MKBUS_INT pin to the pin 6 (ENRGB/S) of the DC749A J1 connector. Do not forget to connect GND between the two boards.

LAB 6.2 Hardware Description for the SAMA5D2 Processor

Open the SAMA5D2B-XULT board schematic and look for the J22 and J17 connectors. Connect the pin 8 (EXP_TWCK1_PD5) of the J22 connector to the pin 4 (SCL) of the DC749A J1 connector and the pin 7 (EXP_TWD1_PD4) of the J22 connector to the pin 7 (SDA) of the DC749A J1 connector. Connect the pin 30 (ISC_D11/EXP_PB25) of the J17 connector to the pin 6 (ENRGB/S) of the DC749A J1 connector. Connect 3.3V from processor's board to the DC749A Vin J2 pin and to the DC749A J20 DVCC connector. Do not forget to connect GND between the two boards.

LAB 6.2 Hardware Description for the BCM2837 Processor

For the BCM2837 processor, you will use the GPIO expansion connector to obtain the I2C signals. Go to the Raspberry-Pi-3B-V1.2-Schematics to see the connector. 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 to the pin 6 (ENRGB/S) of the DC749A J1 connector. Connect 3.3V from processor's board to the Vin J2 pin and to the DC749A J20 DVCC connector. Do not forget to connect GND between the two boards.

LAB 6.2 Device Tree for the i.MX7D Processor

From the MCIMX7D-SABRE mikroBUS™ socket, you can see that MKBUS_I2C_SCL pin connects to the I2C3_SCL pad of the i.MX7D processor, and the MKBUS_I2C_SDA pin to the I2C3_SDA pad. You have to configure these pads as I2C signals. To look for the macro that assigns the required I2C functionality go to the imx7d-pinfunc.h file under arch/arm/boot/dts/ directory and find the next macros:

```
#define MX7D_PAD_I2C3_SDA_I2C3_SDA      0x015C 0x03CC 0x05E8 0x0 0x2
#define MX7D_PAD_I2C3_SCL_I2C3_SCL      0x0158 0x03C8 0x05E4 0x0 0x2
```

The MKBUS_INT pin connects to the SAI1_TX_SYNC processor's pad. To look for the macro that assigns the required GPIO functionality go to the imx7d-pinfunc.h file under arch/arm/boot/dts/ directory and find the next macro:

```
#define MX7D_PAD_SAI1_TX_SYNC__GPIO6_I014      0x0208 0x0478 0x0000 0x5 0x0
```

Now, you can modify the device tree file imx7d-sdb.dts adding the ltc3206@1b sub-node inside the i2c3 controller master node. The pinctrl-0 property of the ltc3206 node points to the pinctrl_cs pin configuration node, where the SAI1_TX_SYNC pad is multiplexed as a GPIO signal. The gpios property will make the GPIO pin 14 of the GPIO6 port available to the driver so that you can set 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 LTC3206 I2C address. Inside the ltc3206 node there are five sub-nodes representing the different display devices. Each of these five nodes have a label property so that the driver can identify them and create devices with the provided label names:

```
&i2c3 {
    clock-frequency = <100000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c3>;
    status = "okay";

ltc3206: ltc3206@1b {
    compatible = "arrow,ltc3206";
    reg = <0x1b>;
    pinctrl-0 = <&pinctrl_cs>;
    gpios = <&gpio6 14 GPIO_ACTIVE_LOW>

    led1r {
        label = "red";
    };

    led1b {
        label = "blue";
    };

    led1g {
        label = "green";
    };

    ledmain {
        label = "main";
    };

    ledsub {
        label = "sub";
    };
}
```

```
};

sii902x: sii902x@39 {
    compatible = "SII,sii902x";
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_sii902x>;
    interrupt-parent = <&gpio2>;
    interrupts = <13 IRQ_TYPE_EDGE_FALLING>;
    mode_str ="1280x720M@60";
    bits-per-pixel = <16>;
    reg = <0x39>;
    status = "okay";
};

[...]

};
```

See below the pinctrl_cs pin configuration node located inside the iomuxc node, where the SAI1_TX_SYNC pad is multiplexed as a GPIO signal:

```
pinctrl_cs: cs_gpiogrp {
    fsl,pins = <
        MX7D_PAD_SAI1_TX_SYNC__GPIO6_IO14    0x2
    >;
};

};
```

LAB 6.2 Device Tree for the SAMA5D2 Processor

Open the SAMA5D2B-XULT board schematic and look for the J22 connector. You see that EXP_TWCK1_PD5 pin connects to the PD5 pad of the SAMA5D2 processor, and EXP_TWD1_PD4 pin to the PD4 pad. You have to configure the PD5, and PD4 pads as I2C signals. To look for the macro that assigns the required I2C functionality go to the sama5d2-pinfunc.h file under arch/arm/boot/dts/ directory and find the next macros below:

```
#define PIN_PD4__TWD1          PINMUX_PIN(PIN_PD4, 1, 2)
#define PIN_PD5__TWCK1          PINMUX_PIN(PIN_PD5, 1, 2)
```

Find now the J17 connector. You see that the ISC_D11/EXP_PB25 pin connects to the PB25 pad of the SAMA5D2 processor. You have to configure this pin as a GPIO signal. To look for the macro that assigns the required GPIO functionality go to the sama5d2-pinfunc.h file under arch/arm/boot/dts/ directory and find the next macro:

```
#define PIN_PB25__GPIO          PINMUX_PIN(PIN_PB25, 0, 0)
```

Open and modify the device tree file at91-sama5d2_xplained_common.dtsi adding the ltc3206@1b sub-node inside the i2c1 controller master node. The pinctrl-0 property of the ltc3206 node points to the pinctrl_cs_default pin configuration node, where the PB25 pad is multiplexed as a GPIO signal.

The PB25 pad is also being multiplexed as a GPIO for the isc node. This node is included in the at91-sama5d2_xplained_ov7670.dtsi file under the arch/arm/boot/dts/ folder. Comment out the following line inside the at91-sama5d2_xplained_common.dtsi file to avoid this "mux" conflict:

```
//#include "at91-sama5d2_xplained_ov7670.dtsi"
```

The gpios property will make the GPIO pin 25 of the PIOB port available to the driver so that you can set the pin direction to output and drive the physical line level from 0 to 1 to control the ENRGB/S pin. To set up this GPIO pin in the gpios property, you will add I/O lines from the PIOA GPIO pin 0 to the PIOB GPIO pin 25 setting 32+25 = 57 in the gpios value. The reg property provides the LTC3206 I2C address. Inside the ltc3206 node there are five sub-nodes representing the different display devices. Each of these five nodes have a label property so that the driver can identify and create devices with the provided label names:

```
i2c1: i2c@fc028000 {
    dmas = <0>, <0>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c1_default>;
    status = "okay";

    [...]

ltc3206: ltc3206@1b {
    compatible = "arrow,ltc3206";
    reg = <0x1b>;
    pinctrl-0 = <&pinctrl_cs_default>;
    gpios = <&pioA 57 GPIO_ACTIVE_LOW>

    led1r {
        label = "red";
    };

    led1b {
        label = "blue";
    };

    led1g {
        label = "green";
    };

    ledmain {
        label = "main";
    };
}
```

```
    };
    ledsub {
        label = "sub";
    };
};

[...]

at24@54 {
    compatible = "atmel,24c02";
    reg = <0x54>;
    pagesize = <16>;
};
};
```

See below the pinctrl_cs_default pin configuration node, where the PB25 pad is multiplexed as a GPIO signal:

```
pinctrl_cs_default: cs_gpio_default {
    pinmux = <PIN_PB25_GPIO>;
    bias-disable;
};
```

LAB 6.2 Device Tree for the BCM2837 Processor

Open Raspberry-Pi-3B-V1.2-Schematics and find the GPIO EXPANSION connector. You can see that the GPIO2 pin connects to the GPIO2 pad of the BCM2837 processor, and GPIO3 pin to the GPIO3 pad. These pads are multiplexed as I2C signals using the ALT0 mode. You are going to connect the GPIO EXPANSION GPIO23 pin to the pin 6 (ENRGB/S) of the DC749A J1 connector, so the GPIO23 pad must be multiplexed as a GPIO signal.

Open and modify the device tree file `bcm2710-rpi-3-b.dts` adding the `ltc3206@1b` sub-node inside the `i2c1` controller master node. The `pinctrl-0` property of the `ltc3206` node points to the `cs_pins` pin configuration node, where the GPIO23 pad 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 LTC3206 I2C address. Inside the `ltc3206` node there are five sub-nodes representing the different display devices. Each of the five nodes have a `label` property so that the driver can identify and create devices with the provided 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 pad 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

An Unified Device Properties API has been defined to provide a format compatible with existing device tree schemas. The purpose for this was to allow for the reuse of the existing schemas and encourage the development of firmware agnostic device drivers. It is now possible to pass device configuration information from ACPI in addition to DT. In order to support this, it is needed to convert the driver to use the unified device property accessors instead of DT specific.

New generic routines are provided for retrieving properties from device description objects in the platform firmware in case there are no struct device objects for them (either those objects have not been created yet or they do not exist at all).

The following functions in bold are provided in analogy with the corresponding functions for the struct device structure added previously:

```
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 struct fwnode_handle structure defined in include/linux/fwnode.h that 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 new 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 device 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 that will store the specific info for each of the five led devices. The first field is the brightness variable, that will hold values ranging from "0" to "15". The second field is a struct led_classdev variable, 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 updated and is exposed to all the led devices; this 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 info accessible to all the led devices. The first field of the private structure is the num_leds variable, that will hold the number of declared DT led devices. 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 struct gpio_desc structure that will allow you to control the ENRGB/S pin, and the last field is a pointer to a struct 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. See below an extract of the probe() routine with the main lines of code marked in bold. These are the main points to set up the driver within the probe() function:

- Declare a pointer to a struct fwnode_handle structure and a pointer to the global private struct led_priv structure.
- Get the number of led devices with the device_get_child_node_count() function.
- Allocate the global private structure with devm_kzalloc() and store the pointer to your client device on it (private->client = client). The i2c_set_clientdata() function is attaching your allocated 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 DT gpios property fields 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 for each led child node allocating a private structure struct led_device for each one using the devm_kzalloc() function and initializing the struct led_classdev field included in each allocated private structure. The fwnode_property_read_string() function reads each led node label property and stores it in the cdev->name field of each struct 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 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 sysfs brightness 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 private struct led_device structure associated to each device is recovered using the container_of() function. Depending of the cdev->name value of the recovered struct led_device different masks are applied to the char array command values, then the updated values are stored in the global struct led_priv structure. Finally, you will send the updated command values to the LTC3206 device 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 struct device_attribute with the respective names 'rgb' and 'sub' and 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 sysfs rgb_select() and sub_select() functions that will be called each time the user application writes "on" or "off" to the sysfs "rgb" and "sub" entries. Inside these functions you are going to recover the struct i2_client structure using the to_i2c_client() function and after that the i2c_get_clientdata() function will recover the global struct led_priv structure. The i2c_get_clientdata() function takes the previously recovered struct i2_client structure as a parameter. Once you have retrieved the global private structure you can update the bit A2 of the command[0] using the mask EN_CS_SHIFT and after that you will send the new command values to the LTC3206 device 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" using the gpiod_set_value() function, that takes as a parameter the gpio descriptor stored in your global struct led_priv 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 struct i2c_device_id structures:

```
static const struct i2c_device_id ltc3206_id[] = {
    { "ltc3206", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, ltc3206_id);
```

11. Add a struct 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. Build the modified device tree, and load it to the target processor.

See in the next **Listing 6-2** the "I2C multidisplay LED" driver source code (`ltc3206_imx_led_class.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`ltc3206_sam_led_class.c`) and BCM2837 (`ltc3206_rpi_led_class.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 6-2: `ltc3206_imx_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 a led_device struct for each 5 led device */
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 led_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 for each led device
 * when writing the brightness file under each device
 * the command parameters are kept in the led_priv struct
 * that is pointed inside each led_device struct
 */
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 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_imx_led_class.ko Demonstration

Connect the ENRGB/S pin to DVCC

```
root@imx7dsabresd:~# insmod ltc3206_imx_led_class.ko /* load the module, probe() function is called 5 times, and the LED BLUE is ON */
root@imx7dsabresd:~# ls -l /sys/class/leds /* find all the devices under leds class */
root@imx7dsabresd:~# echo 10 > /sys/class/leds/red/brightness /* switch on the LED RED with value 10 of brightness */
root@imx7dsabresd:~# echo 15 > /sys/class/leds/red/brightness /* set maximum brightness for LED RED */
root@imx7dsabresd:~# echo 0 > /sys/class/leds/red/brightness /* switch off the LED RED */
root@imx7dsabresd:~# echo 10 > /sys/class/leds/blue/brightness /* switch on the LED BLUE with value 10 of brightness */
root@imx7dsabresd:~# echo 15 > /sys/class/leds/blue/brightness /* set maximum brightness for LED BLUE */
root@imx7dsabresd:~# echo 0 > /sys/class/leds/blue/brightness /* switch off the LED BLUE */
root@imx7dsabresd:~# echo 10 > /sys/class/leds/green/brightness /* switch on the LED GREEN with value 10 of brightness */
root@imx7dsabresd:~# echo 15 > /sys/class/leds/green/brightness /* set maximum brightness for LED GREEN */
root@imx7dsabresd:~# echo 0 > /sys/class/leds/green/brightness /* switch off the LED GREEN */
root@imx7dsabresd:~# echo 10 > /sys/class/leds/main/brightness /* switch on the display MAIN with value 10 of brightness */
root@imx7dsabresd:~# echo 15 > /sys/class/leds/main/brightness /* set maximum brightness for MAIN display*/
root@imx7dsabresd:~# echo 0 > /sys/class/leds/main/brightness /* switch off the MAIN display */
root@imx7dsabresd:~# echo 10 > /sys/class/leds/sub/brightness /* switch on the SUB display with value 10 of brightness */
root@imx7dsabresd:~# echo 15 > /sys/class/leds/sub/brightness /* set maximum brightness for SUB display */
root@imx7dsabresd:~# echo 0 > /sys/class/leds/sub/brightness /* switch off the SUB display */

"Mix RED, GREEN, BLUE colors"

root@imx7dsabresd:~# echo 15 > /sys/class/leds/red/brightness
root@imx7dsabresd:~# echo 15 > /sys/class/leds/blue/brightness
root@imx7dsabresd:~# echo 15 > /sys/class/leds/green/brightness

root@imx7dsabresd:~# rmmod ltc3206_imx_led_class.ko /* remove the module, remove() function is called 5 times */

"Switch off the board's supply and connect the ENRGB/S pin to the Mikrobus INT pin. Switch on the board's supply booting the target processor"
```

```
root@imx7dsabresd:~# insmod ltc3206_imx_led_class.ko /* load the module, probe()
function is called 5 times, and the LED BLUE is OFF */
root@imx7dsabresd:~# echo 10 > /sys/class/leds/sub/brightness /* switch on the SUB
display with value 10 of brightness, the SUB display is ON */
root@imx7dsabresd:~# echo 10 > /sys/class/leds/red/brightness /* switch on the LED
RED with value 10 of brightness, the LED RED is OFF */
root@imx7dsabresd:~# echo off > /sys/class/i2c-dev/i2c-2/device/2-001b/display_cs/
sub /* switch OFF the SUB display and switch on the LED RED */

root@imx7dsabresd:~# echo off > /sys/class/i2c-dev/i2c-2/device/2-001b/display_cs/
rgb /* switch OFF the RGB LED and switch on the SUB display */

root@imx7dsabresd:~# rmmod ltc3206_imx_led_class.ko /* remove the module, remove()
function is called 5 times */
```

<http://www.rejoiceblog.com/>

Handling Interrupts in Device Drivers

An IRQ is an interrupt request from a device. They can come in over a pin or over a packet. Several devices may be connected to the same pin, thus sharing an IRQ.

In Linux, the **IRQ number** is an enumeration of the possible 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 current design of 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.

The number of **interrupt controllers** registered as unique **irqchips** is growing: for example subdrivers of different kinds such as GPIO controllers avoid re-implementing identical callback mechanisms such as the IRQ core system by modelling their interrupt handlers as irqchips cascading interrupt controllers. Here the interrupt number loses all correspondence to hardware interrupt numbers. Whereas in the past, IRQ numbers could be chosen so that they matched the hardware IRQ line into the root interrupt controller (for example, the component actually firing the interrupt line to the CPU) 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 — this one is architecture dependent — registers an **irq_chip** structure to the kernel. This structure contains a bunch of pointers to the basic routines that are needed to manage IRQs. For instance, this includes routines to enable and disable interrupts at the interrupt controller level, as well as interrupt acknowledgment stub.

```
/*
 * 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);

[...]
};
```

The chip level hardware descriptor structure struct irq_chip 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, masking 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 a struct irq_desc structure, which is the structure that represents an IRQ. A list of IRQ descriptors is maintained in an array indexed by the IRQ number (The interrupt is referenced by an 'unsigned int' numeric value, which selects the corresponding interrupt description structure in the descriptor structures array), called the IRQ descriptor table. The struct irq_desc contains a pointer to the struct irq_domain (included in the struct irq_data structure). 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.

```
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;
#ifndef 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         lastUnhandled; /* Aging timer for unhandled count */
    unsigned int         irqsUnhandled;
    atomic_t              threadsHandled;
    int                  threadsHandledLast;
    raw_spinlock_t        lock;
    struct cpumask        *percpuEnabled;
    const struct cpumask  *percpuAffinity;
#ifndef CONFIG_SMP
    const struct cpumask  *affinityHint;
    struct irq_affinity_notify *affinityNotify;
#endif
#ifndef CONFIG_GENERIC_PENDING_IRQ
    cpumask_var_t        pendingMask;

```

[...]

```
}
```

Inside each struct irq_desc there is an instance of struct irq_data (in bold in the struct irq_desc above), that contains low-level information that is relevant for interrupt management, such as Linux IRQ number, hwirq number, interrupt translation domain (struct irq_domain), and a pointer to interrupt controller operations (struct 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;
#ifndef 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 from the point of view of the hardware when a system has a single interrupt controller (IC). However, a mapping is needed as soon as two ICs, for example, two irq_chip, are available (for example, GIC and GPIO IC). To solve this problem, the Linux kernel came up with the notion of an **IRQ domain**, which is a well-defined translation interface between hardware IRQ numbers and the one used internally in the kernel.

The struct 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.

```
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;
#ifndef 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 struct irq_domain structure on success. The driver must provide to the chosen allocator function a struct 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. You can see described another map methods at Documentation/IRQ-domain.txt.

```
/***
 * 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 hwirq and IRQ numbers. The struct irq_domain is filled with the IRQ mapping by calling irq_create_mapping(), which accepts the irq_domain 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 to the `gpiod_to_irq()` function to get the virtual Linux IRQ number associated with the GPIO pin of a GPIO interrupt controller. Another possibility is to do the mapping for each hwirq in advance (inside the `probe()` function) as shown in the code below:

```
for (j = 0; j < gpiochip->chip.ngpio; j++) {
    irq = irq_create_mapping(
        gpiochip->irq_domain, j);
}
```

If you do the mapping inside the `probe()` function, you can recover the mapped Linux IRQ number inside the `gpio_chip.to_irq()` callback function, which calls the `irq_find_mapping()` function, as you can see for example in the code below extracted from the SAMA5D2 PIO4 controller driver located in `drivers/pinctrl/pinctrl-at91-pio4.c`:

```
static int atmel_gpio_to_irq(struct gpio_chip *chip, unsigned offset)
{
    struct atmel_pioctrl *atmel_pioctrl = gpiochip_get_data(chip);
```

```
    return irq_find_mapping(atmel_pioctrl->irq_domain, offset);
}

static struct gpio_chip atmel_gpio_chip = {
    .direction_input      = atmel_gpio_direction_input,
    .get                  = atmel_gpio_get,
    .direction_output     = atmel_gpio_direction_output,
    .set                  = atmel_gpio_set,
    .to_irq               = atmel_gpio_to_irq,
    .base                 = 0,
};
```

If a mapping for the hwirq doesn't already exist then `irq_create_mapping()` will allocate a new Linux `irq_desc`, 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()` is created a mapping between a Linux IRQ and a hwirq number. The mapping is done inside `.map()` 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 using for example `request_irq()` or `request_threaded_irq()` for the "GPIO device" driver that caused the interrupt. This GPIO device can also be a GPIO controller (see for example the `gpio-max732x.c` driver located under `drivers/gpio/`).

If the GPIO controller driver is not implementing the `.map()` function, then `irq_set_chip_and_handler()` can be called within the `probe()` function, as you can see in the code below extracted from the SAMA5D2 PIO4 controller driver located in `drivers/pinctrl/pinctrl-at91-pio4.c`. Each input pin of the GPIO controller is mapped into the GPIO controller's `irq_domain` and is provided a wrapper handle function `handle_simple_irq` that will call to the IRQ handler function of each GPIO device driver that is claiming the Linux IRQ number from a specific input pin of the GPIO controller.

```
for (i = 0; i < atmel_pioctrl->npins; i++) {
    int irq = irq_create_mapping(atmel_pioctrl->irq_domain, i);

    irq_set_chip_and_handler(irq, &atmel_gpio_irq_chip,
                           handle_simple_irq);
    irq_set_chip_data(irq, atmel_pioctrl);
    dev_dbg(dev,
            "atmel gpio irq domain: hwirq: %d, linux irq: %d\n",
            i, irq);
}
```

The GPIO irqchips usually fall in 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. This means the GPIO irqchip is registered using `irq_set_chained_handler()` or the corresponding, and the **GPIO irqchip handler** will be called immediately from the parent irqchip (for example, the Advanced Interrupt Controller (AIC) in the SAMA5D2 processor or the Generic Interrupt Controller (GIC) in the i.MX7D family) while holding the IRQs disabled. The GPIO irqchip will then end up calling something like this sequence in its interrupt handler:

```
static void atmel_gpio_irq_handler()
{
    chained_irq_enter(...);
    generic_handle_irq(...);
    chained_irq_exit(...);
```

The SoC's GPIO controller driver calls `generic_handle_irq()` to run the handler of each particular GPIO device driver (for example, see the first lab driver included in this chapter, or the `gpio-max732x.c` driver located under `linux/drivers/gpio/`).

In the code below, extracted from `drivers/pinctrl/pinctrl-at91-pio4.c`, you can see the creation of the SAMA5D2 GPIO banks interrupt handlers using the `irq_set_chained_handler()` function. It will be created a handler per GPIO bank interrupt. Inside each handler `atmel_gpio_irq_handler()` there is a call to the wrapper `generic_handle_irq()`, which in turn calls the interrupt handler of each of the GPIO device driver that is requesting this specific GPIO controller interrupt pin using the `request_irq()` function.

```
/* There is one controller but each bank has its own irq line. */
for (i = 0; i < atmel_pioctrl->nbanks; i++) {
    res = platform_get_resource(pdev, IORESOURCE_IRQ, i);
    if (!res) {
        dev_err(dev, "missing irq resource for group %c\n",
                'A' + i);
        return -EINVAL;
    }
    atmel_pioctrl->irqs[i] = res->start;
    irq_set_chained_handler(res->start, atmel_gpio_irq_handler);
    irq_set_handler_data(res->start, atmel_pioctrl);
    dev_dbg(dev, "bank %i: irq=%pr\n", i, res);
}

static void atmel_gpio_irq_handler(struct irq_desc *desc)
{
    unsigned int irq = irq_desc_get_irq(desc);
    struct atmel_pioctrl *atmel_pioctrl = irq_desc_get_handler_data(desc);
    struct irq_chip *chip = irq_desc_get_chip(desc);
```

```
unsigned long isr;
int n, bank = -1;

/* Find from which bank is the irq received. */
for (n = 0; n < atmel_pioctrl->nbanks; n++) {
    if (atmel_pioctrl->irqs[n] == irq) {
        bank = n;
        break;
    }
}

if (bank < 0) {
    dev_err(atmel_pioctrl->dev,
            "no bank associated to irq %u\n", irq);
    return;
}

chained_irq_enter(chip, desc);

for (;;) {
    isr = (unsigned long)atmel_gpio_read(atmel_pioctrl, bank,
                                         ATTEL_PIO_ISR);
    isr &= (unsigned long)atmel_gpio_read(atmel_pioctrl, bank,
                                         ATTEL_PIO_IMR);
    if (!isr)
        break;

    for_each_set_bit(n, &isr, BITS_PER_LONG)
        generic_handle_irq(gpio_to_irq(bank *
                                         ATTEL_PIO_NPINS_PER_BANK + n));
}

chained_irq_exit(chip, desc);
}
```

See the DT SAMA5D2 PIO4 Controller description at Documentation/devicetree/bindings/pinctrl/atmel,at91-pio4-pinctrl.txt. The platform_get_resource(pdev, IORESOURCE_IRQ, i) line of code will retrieve from DT each GPIO bank hwirq number (18, 68, 69, 70) corresponding to the interrupt pins which are sourced to its parent interrupt controller (AIC for the SAMA5D2). These hwirq numbers are entered as parameters (res->start) to the irq_set_chained_handler(res->start, atmel_gpio_irq_handler) together with the GPIO controller handler. The code below is extracted from atmel,at91-pio4-pinctrl.txt file:

```
pioA: pinctrl@fc038000 {
    compatible = "atmel,sama5d2-pinctrl";
    reg = <0xfc038000 0x600>;
    interrupts = <18 IRQ_TYPE_LEVEL_HIGH 7>;
```

```
        <68 IRQ_TYPE_LEVEL_HIGH 7>,
        <69 IRQ_TYPE_LEVEL_HIGH 7>,
        <70 IRQ_TYPE_LEVEL_HIGH 7>;
interrupt-controller;
#interrupt-cells = <2>;
gpio-controller;
#gpio-cells = <2>;
clocks = <&pioA_clk>;

pinctrl_i2c0_default: i2c0_default {
    pinmux = <PIN_PD21_TWD0>,
              <PIN_PD22_TWCK0>;
    bias-disable;
};

pinctrl_led_gpio_default: led_gpio_default {
    pinmux = <PIN_PB0>,
              <PIN_PB5>;
    bias-pull-up;
};
[...]
};
```

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 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 then mask the parent IRQ line until the interrupt is handled by the driver.

To help out in handling the set-up 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()` function adds an irqchip to a gpiochip. This function does:

- Set the `gpiochip.to_irq` field to `gpiochip_to_irq`
- Allocates an `irq_domain` to the `gpiochip` using `irq_domain_add_simple()`
- Create mapping from 0 to `gpiochip.ngpio` using `irq_create_mapping()`

The `gpiochip_set_chained_irqchip()` function sets up a chained irq handler for a `gpio_chip` from a parent IRQ (`client->irq`) and passes the `struct gpio_chip` structure as handler data.

Open the driver drivers/gpio/gpio-max732x.c and see the max732x_irq_setup() function that is called inside the driver's probe() function as an example of the usage of the gpiolib irq helpers:

```
static int max732x_irq_setup(struct max732x_chip *chip,
                           const struct i2c_device_id *id)
{
    struct i2c_client *client = chip->client;
    struct max732x_platform_data *pdata = dev_get_platdata(&client->dev);
    int has_irq = max732x_features[id->driver_data] >> 32;
    int irq_base = 0;
    int ret;

    if (((pdata && pdata->irq_base) || client->irq)
        && has_irq != INT_NONE) {
        if (pdata)
            irq_base = pdata->irq_base;
        chip->irq_features = has_irq;
        mutex_init(&chip->irq_lock);

        devm_request_threaded_irq(&client->dev, client->irq,
                                 NULL, max732x_irq_handler, IRQF_ONESHOT |
                                 IRQF_TRIGGER_FALLING | IRQF_SHARED,
                                 dev_name(&client->dev), chip);

        gpiochip_irqchip_add(&chip->gpio_chip,
                            &max732x_irq_chip,
                            irq_base,
                            handle_simple_irq,
                            IRQ_TYPE_NONE);
    }

    gpiochip_set_chained_irqchip(&chip->gpio_chip,
                                &max732x_irq_chip,
                                client->irq,
                                NULL);
}

return 0;
}
```

The devm_request_threaded_irq() function inside max732x_irq_setup() will take as a parameter the driver's interrupt handler max732x_irq_handler. Inside this handler are checked the pending GPIO interrupts by reading the pending variable value, then is returned the position of the first bit set in the 32 bits variable; 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 each

GPIO device driver that is requesting this specific MAX732x GPIO controller pin using the `request_irq()` function.

The parameter of the `handle_nested_irq()` function is the Linux IRQ number previously returned using the `irq_find_mapping()` function, which in turn has the hwirq of the input pin as a parameter (level variable).

After calling `handle_nested_irq()`, the pending interrupt is cleared by doing `pending &= ~(1 << level)`, and the same process is repeated until all the pending interrupts are being managed.

```
static irqreturn_t max732x_irq_handler(int irq, void *devid)
{
    struct max732x_chip *chip = devid;
    uint8_t pending;
    uint8_t level;

    pending = max732x_irq_pending(chip);

    if (!pending)
        return IRQ_HANDLED;

    do {
        level = __ffs(pending);
        handle_nested_irq(irq_find_mapping(chip->gpio_chip.irq.domain,
                                           level));

        pending &= ~(1 << level);
    } while (pending);

    return IRQ_HANDLED;
}
```

Device Tree Interrupt Handling

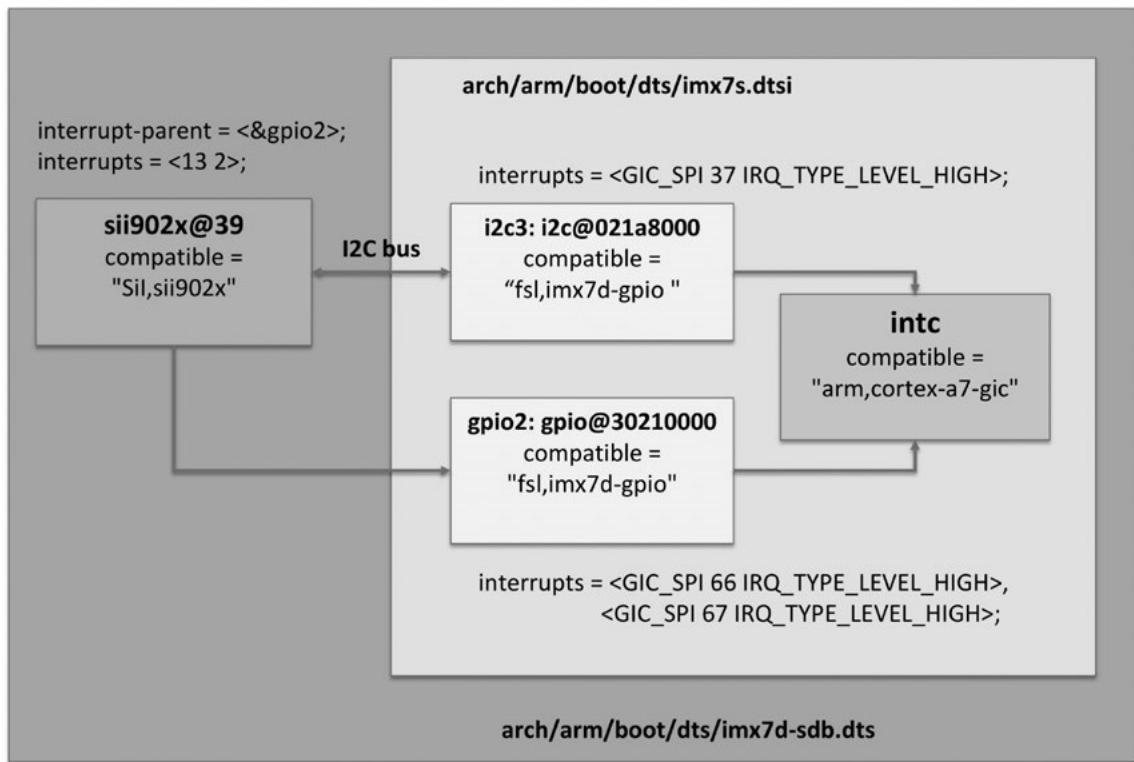
Unlike address range translation, which follows the natural structure of the tree, interrupt signals can originate from and terminate on any device in a machine. Unlike device addressing, which is naturally expressed in the device tree, interrupt signals are expressed as links between nodes independent 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** 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, `cortex-a7-gic`

interrupt controller indicates three cells for the ecspi1 and gpio1 controllers. The gpio1 controller is also an interrupt parent controller for GPIOx signals and indicates two cells for the slave nodes interrupts property. It is similar to the #address-cells and #size-cells properties.

3. The **interrupt-parent** 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 from 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.

The interrupt links between several related DT nodes are shown in the following figure. This is a real example you can check looking at i.MX7D SABRE board DT source files:



In the previous image, you can see that the "sii902x slave node" has the "gpio2 controller node" as its "interrupt parent" and the "i2c3 controller node" as its "parent node" (sii902x device is attached to the i2c3 controller bus). In the sii902x node the GPIO interrupts property has two fields (the first

field is the gpio2 controller hwirq value the device is requesting and the second value is one of the IRQ bindings described in the irq.h file under include/dt-bindings/interrupt-controller/ that were indicated by the #interrupt-cells property of the gpio2 interrupt controller node.

The "intc node" is the "parent interrupt controller node" for the "gpio2 controller node" and the "i2c3 controller node". In the gpio2 controller and i2c3 controller interrupts properties there are three fields that were indicated by the #interrupt-cells property of the ARM Generic Interrupt Controller intc node. See the i2c3, gpio2 and intc nodes in the imx7s.dtsi file under arch/arm/boot/dts/ folder.

In the i2c3, node, the value of the second field of the interrupts property is 37, that matches with the hwirq number of the i2c3 controller output interrupt signal (see Table 7-1. ARM Domain Interrupt Summary on page 1217 of the IMX7DQRM) that is being collected by the the Global Interrupt Controller (GIC).

```
i2c3: i2c@30a40000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx7d-i2c", "fsl,imx21-i2c";
    reg = <0x30a40000 0x10000>;
    interrupts = <GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clks IMX7D_I2C3_ROOT_CLK>;
    status = "disabled";
};

gpio3: gpio@30220000 {
    compatible = "fsl,imx7d-gpio", "fsl,imx35-gpio";
    reg = <0x30220000 0x1000>;
    interrupts = <GIC_SPI 68 IRQ_TYPE_LEVEL_HIGH>,
                 <GIC_SPI 69 IRQ_TYPE_LEVEL_HIGH>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    gpio-ranges = <&iomuxc 0 45 29>;
};

intc: interrupt-controller@31001000 {
    compatible = "arm,cortex-a7-gic";
    interrupts = <GIC_PPI 9 (GIC_CPU_MASK_SIMPLE(4) | IRQ_TYPE_LEVEL_HIGH)>;
    #interrupt-cells = <3>;
    interrupt-controller;
    reg = <0x31001000 0x1000>,
          <0x31002000 0x2000>,
          <0x31004000 0x2000>,
          <0x31006000 0x2000>;
};
```

Look for the sii902x node in the imx7d-sdb.dts file:

```
&i2c3 {
    clock-frequency = <100000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c3>;
    status = "okay";

    ltc2607@72 {
        compatible = "arrow,ltc2607";
        reg = <0x72>;
    };
    ltc2607@73 {
        compatible = "arrow,ltc2607";
        reg = <0x73>;
    };

    ioexp@38 {
        compatible = "arrow,ioexp";
        reg = <0x38>;
    };
    ioexp@39 {
        compatible = "arrow,ioexp";
        reg = <0x39>;
    };

    sii902x: sii902x@39 {
        compatible = "SiI,sii902x";
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_sii902x>;
        interrupt-parent = <&gpio2>;
        interrupts = <13 IRQ_TYPE_EDGE_FALLING>;
        mode_str ="1280x720M@60";
        bits-per-pixel = <16>;
        reg = <0x39>;
        status = "okay";
    };
    [...]
};
```

Requesting Interrupts in Linux Device Drivers

Interrupts are scheduled by the CPU and run asynchronously. The kernel could be in any state when interrupt occurs, so the interrupt context can not access the user buffers, and can not 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 using the `request_irq()` and `free_irq()` functions. To perform this task is recommended to use the managed API for automatic freeing at device or module release time:

```
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 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 to the `struct device` structure, the Linux IRQ number (`irq`), a handler that will be called when the interrupt is generated (`handler`), flags that will instruct the kernel about the desired behaviour (`irqflags`), the name of the device using this interrupt (`devname`), and a pointer 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 role of an **interrupt handler** is to give feedback to its device about 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 sent to `request_irq()` when the interrupt was requested. The interrupt handling routine must return a value with a type of `typedef irqreturn_t`. For the kernel 4.9 version

used in this book, 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, `void *dev_id`, is a sort of client data; this argument is passed to `devm_request_irq()` function, and this same pointer is then 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, indeed, need attention, it should return `IRQ_HANDLED`. If the driver detects that it was not your hardware that caused the interrupt, it will do nothing and return `IRQ_NONE`, allowing the kernel to call the next possible 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 the platform driver.

LAB 7.1 Hardware Description for the i.MX7D Processor

Open the MCIMX7D-SABRE board schematic and find the button USR_BT1 in pag.21. This button will be used to generate the interrupt.

LAB 7.1 Hardware Description for the SAMA5D2 Processor

Open the SAMA5D2B-XULT board schematic and find the button BP1 in pag.11. This button will be used to generate the interrupt.

LAB 7.1 Hardware Description for the BCM2837 Processor

For the BCM2837 processor, you will use the button of the **MikroElektronika Button R click** board. See the board at <https://www.mikroe.com/button-r-click>. You can download the schematic from that link or from the GitHub repository of this book. Connect the GPIO23 pin of the GPIO expansion connector to the INT pin of the Button R click board.

LAB 7.1 Device Tree for the i.MX7D Processor

Open the MCIMX7D-SABRE schematic and find the button USR_BT1. This button is connected to the SD2_WP pad of the i.MX7D processor. To look for the macro that assigns the required GPIO functionality go to the imx7d-pinfunc.h file under arch/arm/boot/dts/ directory and find the next macro:

```
#define MX7D_PAD_SD2_WP__GPIO5_IO10 0x01B0 0x0420 0x0000 0x5 0x0
```

You need a sixth integer to include in the fsl,pins property, that corresponds to the configuration for the PAD control register. This number defines the low-level physical settings of the pin. The chosen value 0x32 will enable the internal pull-up of the pin. When the button is pressed, the GPIO input value is set to GND, generating an interrupt if IRQF_TRIGGER_FALLING flag was passed to the request_irq() function.

Now, you can modify the device tree file imx7d-sdb.dts adding the next code in bold:

```
/ {
    model = "Freescale i.MX7 SabreSD Board";
    compatible = "fsl,imx7d-sdb", "fsl,imx7d";

    memory {
```

```
    reg = <0x80000000 0x80000000>;
};

[...]

int_key{
    compatible = "arrow,intkey";

    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_key_gpio>

    label = "PB_USER";
    gpios = <&gpio5 10 GPIO_ACTIVE_LOW>;
    interrupt-parent = <&gpio5>;
    interrupts = <10 IRQ_TYPE_EDGE_FALLING>;
};

[...]

&iomuxc {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_hog_1>

    imx7d-sdb {

        pinctrl_hog_1: hoggrp-1 {
            fsl,pins = <
                MX7D_PAD_EPDC_BDR0__GPIO2_IO28      0x59
            >;
        };
        [...]
        pinctrl_key_gpio: key_gpiogrp {
            fsl,pins = <
                MX7D_PAD_SD2_WP__GPIO5_IO10      0x32
            >;
        };
        [...]
    };
};

}
```

LAB 7.1 Device Tree for the SAMA5D2 Processor

Open the SAMA5D2B-XULT board schematic and find the button BP1. This button is connected to the PB9 pad of the SAMA5D2 processor. You have to configure in the DT the PB9 pad as a GPIO signal. To look for the macro that assigns the required GPIO functionality go to the sama5d2-pinfunc.h file under arch/arm/boot/dts/ directory and find the next macro:

```
#define PIN_PB9__GPIO      PINMUX_PIN(PIN_PB9, 0, 0)
```

The chosen DT value for the pad setting will enable an internal pull-up on the pin. When the button is pressed, the GPIO input value is set to GND, generating an interrupt if IRQF_TRIGGER_FALLING flag was passed to the request_irq() function.

Now, you can modify the device tree file at91-sama5d2_xplained_common.dtsi adding the next code in bold. Disable the gpio_keys node to avoid "mux" conflict with the PB9 pad:

```
pinctrl@fc038000 {  
  
    pinctrl_adc_default: adc_default {  
        pinmux = <PIN_PD23__GPIO>;  
        bias-disable;  
    };  
  
    [...]  
  
    pinctrl_key_gpio_default: key_gpio_default {  
        pinmux = <PIN_PB9__GPIO>;  
        bias-pull-up;  
    };  
  
    [...]  
}  
  
/  
    model = "Atmel SAMA5D2 Xplained";  
    compatible = "atmel,sama5d2-xplained", "atmel,sama5d2", "atmel,sama5";  
  
    chosen {  
        stdout-path = "serial0:115200n8";  
    };  
  
    [...]  
  
    int_key {  
        compatible = "arrow,intkey";  
        pinctrl-names = "default";  
    };
```

```
pinctrl-0 = <&pinctrl_key_gpio_default>;
gpios = <&pioA 41 GPIO_ACTIVE_LOW>;
interrupt-parent = <&pioA>;
interrupts = <41 IRQ_TYPE_EDGE_FALLING>;
};

[...]
};
```

LAB 7.1 Device Tree for the BCM2837 Processor

For the BCM2837 processor, 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 IRQF_TRIGGER_FALLING flag was passed to the request_irq() function. Open and modify the device tree file `bcm2710-rpi-3-b.dts` adding the next code in bold:

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

&gpio {
    sdhost_pins: sdhost_pins {
        brcm,pins = <48 49 50 51 52 53>;
        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";
    };
};
```

```
expgpio: expgpio {
    compatible = "brcm,bcm2835-expgpio";
    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 DT int_key node using the devm_gpiod_get() function, then the Linux IRQ number corresponding to the given GPIO is returned using the function gpiod_to_irq(), that 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 DT int_key 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 struct device structure, the Linux IRQ number (irq), a handler that will be called when the interrupt is generated

(hello_keys_isr), a flag that will instruct the kernel about the desired interrupt behaviour (IRQF_TRIGGER_FALLING), the name of the device using this interrupt (HELLO_KEYS_NAME), and a pointer that can be configured at any value. In this driver, dev_id will point to your struct 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 struct device structure, that 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 struct 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. Build the modified device tree, and load it to the target processor.

See in the next **Listing 7-1** the "button interrupt device" driver source code (`int_imx_key.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`int_sam_key.c`) and BCM2837 (`int_rpi_key.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 7-1: `int_imx_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_misctype);
    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_imx_key.ko Demonstration

```
root@imx7dsabresd:~# insmod int_imx_key.ko /* load module */
"Press the FUNC2 button to generate interrupts"

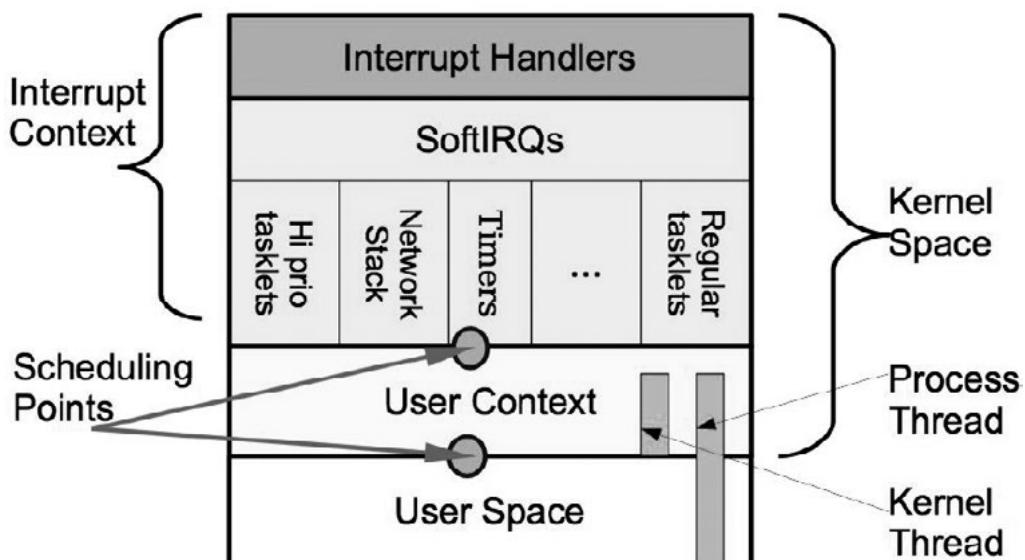
root@imx7dsabresd:~# cat /proc/interrupts /* check the linux IRQ number (220) and
hwirq number (10) for the gpio-mxc controller */
root@imx7dsabresd:~# rmmod int_imx_key.ko /* remove module */
```

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 can not call blocking functions.

Deferred work is a class of kernel facilities that allows one to schedule code to be executed at a later time. 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**. To summarize, the main types of deferred work are kernel threads and softirqs. Work queues and bottom-half of threaded irqs are implemented on top of kernel threads that are able to block and tasklets and timers 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

Using deferred work you can perform the minimum required work in the interrupt handler and schedule an asynchronous action from the interrupt handler to run at a later time and execute the rest of the operations. This deferred work used in interrupts is also known as bottom-half, since its purpose is to execute the rest of the actions 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. It should complete as quickly as possible since all interrupts are disabled. It 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 irqs in process context.

Softirqs

Softirqs are a form of bottom half processing that run in interrupt context, and are suitable for long running, non-blocking handlers. They are executed once all interrupt handlers have completed, before the kernel resumes scheduling processes, so sleeping is not allowed. A softirq is executed as early as possible but remains interruptible so that the execution can be preempted by any top half. Softirqs can not be used by device drivers, they are reserved for various kernel subsystems. Because of this there is a fixed number of softirqs defined at compile time. For the current kernel version the following types are defined:

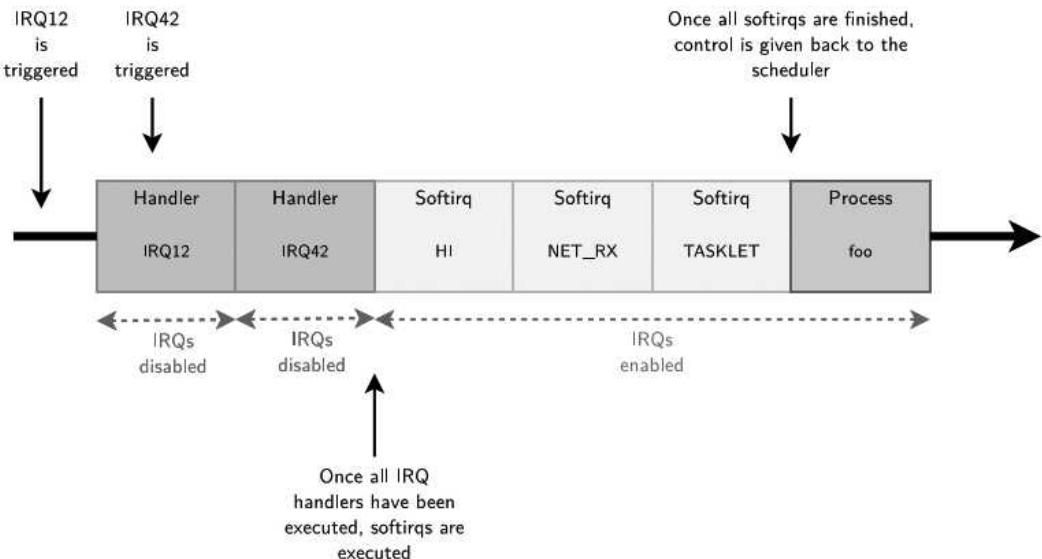
```
enum {
    HI_SOFTIRQ = 0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    IRQ_POLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ,
    RCU_SOFTIRQ,
    NR_SOFTIRQS
};
```

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 can not 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 special form of deferred work that runs in interrupt context, just like softirqs. The main difference between softirqs and tasklets is that tasklets can be allocated dynamically and thus they can be used by device drivers. A tasklet is represented by struct tasklet 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 following:

```
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 tasklet execution with:

```
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

A particular type of deferred work, very often used, are timers. They are defined by the struct timer_list structure. They run in interrupt context and are implemented on top of softirqs. 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 structure and associates 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 timers is **jiffies**. The absolute value of a jiffie is dependent on the platform and it can be found using the `HZ` macro that 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 a LED every second using a timer deferred work. You can change the blinking period from user space using the sysfs `period` entry. You can test the driver using your Raspberry Pi 3 Model B board and the Color click™ accessory board with the HW configuration used in 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 + 0x200000

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_misctype);
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

There are situations where the device driver handling the interrupts can't read the device's registers 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, in the interruption, you must plan a work-in-process action (work queue, kernel thread) to access the device's registers. 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 will implement critical operations while the `thread_fn` handler runs in process context and implements the rest of the operations.

```
/**  
 * 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;  
    }  
}
```

Note that the first IRQ handler can be set to NULL so that the default primary handler will be executed. This default handler does nothing more than to return IRQ_WAKE_THREAD to wake up the associated kernel thread, which will execute the thread_fn handler. Hence, it is possible to move the execution of interrupt handlers entirely to process context, which is widely used in the domain of real-time systems. In such case the IRQF_ONESHOT argument must be passed, otherwise the request will fail. If you do not specify the IRQF_ONESHOT argument in such a

situation, the interrupt would be reenabled after return of the top half, resulting in stack overflows for level interrupts since the issuing device has still the interrupt line asserted. Hence, the kernel rejects such requests and throws an error message. Note that `thread_fn` runs competing for CPU along with other processes on the runqueue, but it can run for longer duration without hurting the overall system responsiveness, as it does not run in high-priority by default unlike SoftIRQs. When the `thread_fn` completes, the associated kthread would take itself out of the runqueue and remain in blocked state until woken up again by the hard-IRQ function.

SoftIRQs run at a high-priority with scheduler preemption being disabled, not relinquishing CPU to processes/threads until they complete. Thus, if functions registered in softIRQs fail to finish within a jiffy (1 to 10 milliseconds based on `CONFIG_HZ` of the kernel), this can severely impact the responsiveness of the kernel, as processes/threads would be starved for CPU. If the scheduler could not be invoked periodically while there are processes in the run-queue, possibly due to softIRQ flooding, any new softIRQs raised by the Hard-IRQ handlers would be delegated to run in process context via `ksoftirqd` thread, thus making softIRQs compete for their CPU share along with other processes and threads on the runqueue. In general, softIRQs are preferred for bottom-half processing that could finish consistently in few 100 microseconds (well within a jiffy).

Threaded IRQ handlers are preferred for bottom-half processing that would spill over half a jiffy consistently (for example, more than 500 microseconds if `CONFIG_HZ` is set to 1000). When you use the `request_threaded_irq()` function passing two functions as arguments, this would create a dedicated kthread similar to `[irq-x/device_intr]` where the number `x` represents the irq number associated with this thread, and `device_intr` would be the name provided as a string argument to the `request_threaded_irq()` function. One of the advantages of using threaded IRQs in multi-core/SMP architectures is that you can set affinity of hard-IRQ for one CPU, while setting affinity of the associated kthread to another CPU, thus allowing top half and bottom half to run in parallel. They are in general, more flexible than softIRQs, though interrupt processing might be delayed due to scheduler preemption if the runqueue is overloaded with lots of CPU hungry processes/threads.

Workqueues

Workqueues are used to schedule actions to run in process context. The base unit with which they work is called **work** and is queued in a **workqueue**. All work items are dequeued and executed in process context. A kernel thread called **worker**, picks up the items from the workqueue and executes the functions associated with those items. If there is no work, the worker will idle, but wake up as soon as new items arrive in the workqueue. Workqueues are also suitable for long running and lengthy tasks, since the system scheduler can guarantee forward progress for other threads. Workqueues can be used inside and outside the context of interrupt handling.

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.

The new Concurrency Managed Workqueue (cmwq) design differentiates between the user-facing workqueues that subsystems and drivers queue work items on and the backend mechanism, which manages worker-pools and processes the queued work items. There are two worker-pools, one for normal work items and the other for high priority ones, for each possible CPU and some extra worker-pools to serve work items queued on unbound workqueues - the number of these backing pools is dynamic.

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.

When a work item is queued to a workqueue, the target worker-pool is determined according to the queue parameters and workqueue attributes and appended on the shared worklist of the worker-pool. For example, unless specifically overridden, a work item of a bound workqueue will be queued on the worklist of either normal or high priority worker-pool that is associated to the CPU the issuer is running on.

Each worker-pool bound to an actual CPU implements concurrency management by hooking into the scheduler. The worker-pool is notified whenever an active worker wakes up or sleeps and keeps track of the number of the currently runnable workers. Generally, work items are not expected to hog a CPU and consume many cycles. That means maintaining just enough concurrency to prevent work processing from stalling should be optimal. As long as there are one or more runnable workers on the CPU, the worker-pool doesn't start execution of a new work, but, when the last running worker goes to sleep, it immediately schedules a new worker so that the CPU doesn't sit idle while there are pending work items. This allows using a minimal number of workers without losing execution bandwidth.

The workqueue API offers two types of functions 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 work:

- structure work_struct - it schedules a task to run at a later time
- struct 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 struct work_struct, but has _delayed in the functions 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 sequence 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 into the workqueue through schedule_work() and schedule_delayed_work(). This function enqueues the given work item on the local CPU workqueue, but does not guarantee its execution on 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():

```
void flush_scheduled_work(void);
```

Finally, the following functions can be used to schedule work items on a particular processor (`schedule_delayed_work_on()`), or on all processors (`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 *));
```

A usual sequence to initialize and schedule a work item is the following:

```
void my_work_handler(struct work_struct *work);
struct work_struct my_work;
INIT_WORK(&my_work, my_work_handler);
schedule_work(&my_work);
```

And for waiting for termination of a work item:

```
flush_scheduled_work();
```

As you can see, the `my_work_handler()` function receives the task as the parameter. To be able to access the module's private data, you can use `container_of()`:

```
struct my_device_data {
    struct work_struct my_work;
    [...]
};

void my_work_handler(struct work_struct *work)
{
    struct my_device_data * my_data;
    my_data = container_of(work, struct my_device_data, my_work);
    [...]
}
```

Scheduling work items with the functions above will run the handler in the context of a thread kernel called `events/x`, where `x` is the processor number. The kernel will initialize a kernel thread (or a pool of workers) for each processor 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 items handlers can cause delays for other queue users. For this reason there are functions for creating additional queues.

A workqueue is represented by the `struct workqueue_struct` structure. A new workqueue can be created with these 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 item to finish call `flush_workqueue()`:

```
void flush_workqueue(struct worksqueue_struct * queue);
```

To destroy the workqueue call `destroy_workqueue()`:

```
void destroy_workqueue(structure workqueue_struct *queue);
```

The next sequence declares and initializes an additional workqueue, 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);
```

And the next lines of code show how to remove the workqueue:

```
flush_workqueue(my_workqueue);
destroy_workqueue(my_workqueue);
```

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 that is running read the value of the counter, and before incrementing is preempted for the other kthread; the second kthread also read the value of the counter and in this occasion increment it; now the second kthread is preempted for the first one that increments the counter. The value of the counter has only been incremented by one unit after been 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 starting running on SMP machines, they became one of the major issues in kernel design and implementation.

Preemption can have the same effect, even if there is only one CPU, by preempting one task during the critical region, we have exactly the same race condition. In this case the thread which preempts might run the critical region itself.

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** (include/asm/spinlock.h), 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 can be used anywhere.

The second type is a **mutex** (include/linux/mutex.h); 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 in non-interrupt (process) execution context in the kernel. In this case spin_lock_irqsave and the spin_unlock_irqrestore variants have to be used to protect the critical section. This has the effect of disabling interrupts on the executing CPU. You can see in the steps below what could happen if you just used spin_lock in the process context:

1. Process context kernel code acquires the spinlock using spin_lock.
2. While the spinlock is held, an interrupt comes in on the same CPU and executes.
3. Interrupt Service Routing (ISR) tries to acquire the spinlock, and spins continuously waiting for it. Process context is blocked in the CPU and there is never a chance to run again and free the spinlock.

To prevent this, the process context code needs call spin_lock_irqsave, which has the effect of disabling interrupts on that particular CPU along with the regular disabling of preemption on the executing CPU.

The ISR can still just call `spin_lock` instead of `spin_lock_irqsave` because interrupts are disabled anyway during ISR on the executing CPU. Often times people use `spin_lock_irqsave` in an ISR, that's not necessary.

Also note that during the executing of the critical section protected by `spin_lock_irqsave`, the interrupts are only disabled on the executing CPU. Another interrupt that needs to access to the shared spinlock can come in on a different CPU and the ISR will be executed there, but that will not trigger the hard lock condition, because the process-context code is not blocked and can finish executing the locked critical section and release the spinlock while the ISR spins on the spinlock on a different CPU waiting for it. The process context does get a chance to finish and free the spinlock causing no hard lock up.

In the lab 7.3, you will see the use of shared spinlocks to protect between interrupt and process context.

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 (`include/linux/mutex.h`) to protect it. This is the most trivial case: you initialize the mutex. Then you can call `mutex_lock_interruptible()` to grab the mutex, and `mutex_unlock()` to release it. There is also a `mutex_lock()`, which should be avoided, because it will not return if a signal is received.

Sleeping in the Kernel

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. So any sleeping that happens while holding semaphores should be short, and you should convince yourself that, by holding the semaphore, you are not blocking the process that will eventually wake you up. 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 managed by means of a "wait queue head," a structure of type `wait_queue_head_t`, which is defined in `linux/wait.h`. A wait queue head can be defined and initialized statically with:

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

or dynamically as follows:

```
wait_queue_head_t my_queue;  
init_waitqueue_head(&my_queue);
```

When a process sleeps, it does so in expectation that some condition will become true in the future. As noted before, any process that sleeps must check to be sure that the condition it was waiting for is really true when it wakes up again.

The simplest way of sleeping in the Linux kernel is a macro called `wait_event` (with a few variants); it combines handling the details of sleeping with a check on the condition a process is waiting for. The forms of `wait_event` are:

```
wait_event(queue, condition)  
wait_event_interruptible(queue, condition)  
wait_event_timeout(queue, condition, timeout)  
wait_event_interruptible_timeout(queue, condition, timeout)
```

In all of the above forms, `queue` is the wait queue head to use. The `condition` is an arbitrary boolean expression that is 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.

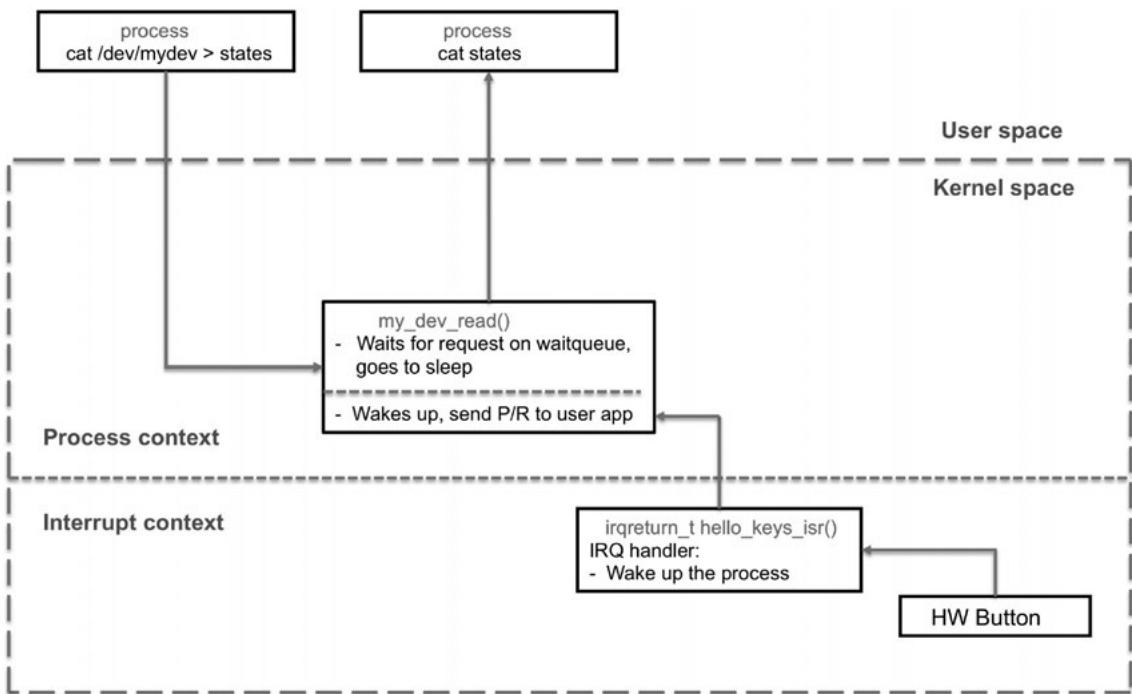
The other half of the picture, of course, is waking up. Some other thread of execution (a different process, or an interrupt handler) has to perform the wakeup for you, since your process is, of course, asleep. The basic function that wakes up sleeping processes is called `wake_up`. It comes in several forms (but only two of them will be reviewed now):

```
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 the next lab 7.2, you are going to develop a kernel module that causes a process to sleep and then wakes it up via an interrupt. You can see the driver's behavior 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 for all the different processors.

LAB 7.2 Device Tree for the i.MX7D Processor

Open the MCIMX7D-SABRE schematic and find the button USR_BT1. This button is connected to the SD2_WP pad of the i.MX7D processor. To look for the macro that assigns the required GPIO functionality go to the imx7d-pinfuc.h file under arch/arm/boot/dts/ directory and find the next macro:

```
#define MX7D_PAD_SD2_WP__GPIO5_I010      0x01B0 0x0420 0x0000 0x5 0x0
```

You need a sixth integer that corresponds to the configuration for the PAD control register. This number defines the low-level physical settings of the pin. The chosen value will enable an internal pull-up on the pin. 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 the IRQ_TYPE_EDGE_BOTH value in the interrupts property.

Now, you can modify the device tree file imx7d-sdb.dts adding the next code in bold:

```
/ {
    model = "Freescale i.MX7 SabreSD Board";
    compatible = "fsl,imx7d-sdb", "fsl,imx7d";

    memory {
        reg = <0x80000000 0x80000000>;
    };

    [...]

    int_key_wait {
        compatible = "arrow,intkeywait";

        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_key_gpio>

        label = "PB_USER";
        gpios = <&gpio5 10 GPIO_ACTIVE_LOW>;
        interrupt-parent = <&gpio5>;
        interrupts = <10 IRQ_TYPE_EDGE_BOTH>;
    };

    [...]

    &iomuxc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hog_1>;
        imx7d-sdb {

            pinctrl_hog_1: hoggrp-1 {
```

```
        fsl,pins = <  
                    MX7D_PAD_EPDC_BDR0__GPIO2_IO28      0x99  
                >;  
};  
[...]  
  
pinctrl_key_gpio: key_gpiogrp {  
        fsl,pins = <  
                    MX7D_PAD_SD2_WP__GPIO5_IO10      0x32  
                >;  
};  
[...]  
};  
};
```

LAB 7.2 Device Tree for the SAMA5D2 Processor

Open the SAMA5D2B-XULT board schematic and find the button BP1. This button is connected to the PB9 pad of the SAMA5D2 processor. You have to configure in the DT the PB9 pad as a GPIO signal. To look for the macro that assigns the required GPIO functionality go to the sama5d2-pinfunc.h file under arch/arm/boot/dts/ directory and find the next macro:

```
#define PIN_PB9_GPIO      PINMUX_PIN(PIN_PB9, 0, 0)
```

The chosen DT value for the pad setting will enable an internal pull-up on the pin. 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 the IRQ_TYPE_EDGE_BOTH value in the interrupts property.

Now, you can modify the device tree file `at91-sama5d2_xplained_common.dtsi` adding the next code in bold. Disable the `gpio_keys` node to avoid "mux" conflict with the PB9 pad:

```
pinctrl@fc038000 {  
  
    pinctrl_adc_default: adc_default {  
        pinmux = <PIN_PD23__GPIO>;  
        bias-disable;  
    };  
  
    [...]  
  
    pinctrl_key_gpio_default: key_gpio_default {  
        pinmux = <PIN_PB9__GPIO>;  
        bias-pull-up;  
    };
```

```
[...]  
}  
  
/ {  
    model = "Atmel SAMA5D2 Xplained";  
    compatible = "atmel,sama5d2-xplained", "atmel,sama5d2", "atmel,sama5";  
  
    chosen {  
        stdout-path = "serial0:115200n8";  
    };  
  
    [...]  
  
    int_key_wait {  
        compatible = "arrow,intkeywait";  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_key_gpio_default>;  
        gpios = <&pioA 41 GPIO_ACTIVE_LOW>;  
        interrupt-parent = <&pioA>;  
        interrupts = <41 IRQ_TYPE_EDGE_BOTH>;  
    };  
  
    [...]  
};
```

LAB 7.2 Device Tree for the BCM2837 Processor

For the BCM2837 processor, 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 the **IRQ_TYPE_EDGE_BOTH** value in the **interrupts** property.

Open and modify the device tree file `bcm2710-rpi-3-b.dts` adding the next code in bold:

```
/ {  
    model = "Raspberry Pi 3 Model B";  
};  
  
&gpio {  
    sdhost_pins: sdhost_pins {  
        brcm,pins = <48 49 50 51 52 53>;  
        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";
    };
};

expgpio: expgpio {
    compatible = "brcm,bcm2835-expgpio";
    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 struct gpio_desc structure associated with your button GPIO. In this driver, you will handle a char device, so a struct miscdevice will be created, initialized and added to your device specific private 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 holds your 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 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 struct device, the interrupt number, a handler that will be called when the interrupt is generated (hello_keys_isr), a flag that will instruct the kernel about the desired interrupt behaviour (IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING), the name of the device using this interrupt (HELLO_KEYS_NAME), and a pointer that in this driver will point 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 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_misdevice.name = "mydev";
priv->int_misdevice.minor = MISC_DYNAMIC_MINOR;
priv->int_misdevice.fops = &my_dev_fops;

ret_val = misc_register(&priv->int_misdevice);
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 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 using the wake_up_interruptible() function, that 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 my_dev_read() kernel function that gets called whenever an user space read operation occurs on the character device file. You will recover the private structure 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 before waking 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 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 struct 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. Build the modified device tree, and load it to the target processor.

See in the next **Listing 7-2** the "sleeping device" driver source code (int_imx_key_wait.c) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (int_sam_key_wait.c) and BCM2837 (int_rpi_key_wait.c) drivers can be downloaded from the GitHub repository of this book.

Listing 7-2: int_imx_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 new structure representing device */
    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 virtual int 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_misctype.name = "mydev";
priv->int_misctype.minor = MISC_DYNAMIC_MINOR;
priv->int_misctype.fops = &my_dev_fops;

ret_val = misc_register(&priv->int_misctype);
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_misctype);
    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_imx_key_wait.ko Demonstration

```
root@imx7dsabresd:~# insmod int_imx_key_wait.ko /* load module */  
root@imx7dsabresd:~# cat /proc/interrupts /* check the linux IRQ number (220) and  
hwirq number (10) for the gpio-mxc controller */  
root@imx7dsabresd:~# cat /dev/mydev > states /* sleep the process */  
"Press and release the FUNC2 button several times"  
  
root@imx7dsabresd:~# cat states /* check all the times you pressed and released the  
button */  
root@imx7dsabresd:~# rmmod int_imx_key_wait.ko /* remove module */
```

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 thread kernel, 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` ; can contain sequences `%d` , `%s` etc., which will be replaced according to the standard `printf` syntax.

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 `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 the last driver of this chapter, you will work with many of the concepts learned during the previous chapters as well as in the current chapter to develop the driver. You will create a new class called `Keyled`. Several led devices will be created under the `Keyled` class, and also several sysfs entries will be created under each led device. You will control each led device by writing from user space to the sysfs entries under each led device registered to the `Keyled` class. In this driver, you will not initialize a `struct cdev` structure for each device (adding as an argument a `file_operations` structure), and then add it to kernel space, so you will not see led devices under `/dev` that can be controlled using system calls; the led devices will be controlled writing to the sysfs entries under `/sys/class/Keyled/<led_device>/` directory.

The blinking value period of each led device will be incremented or decremented via interrupts using two buttons. A kernel thread will manage the led blinking, toggling the output value of the GPIO connected to the led using the `msleep()` function.

LAB 7.3 Hardware Description for the i.MX7D Processor

You will use three pins of the i.MX7D to control each LED. These pins must be multiplexed as GPIOs in the DT. Go to the pag.20 of the MCIMX7D-SABRE schematic to see the MikroBUS connector. You will use the MOSI pin to control the green LED, the SCK pin to control the blue LED and the MKBUS_PWM pin to control the red LED.

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

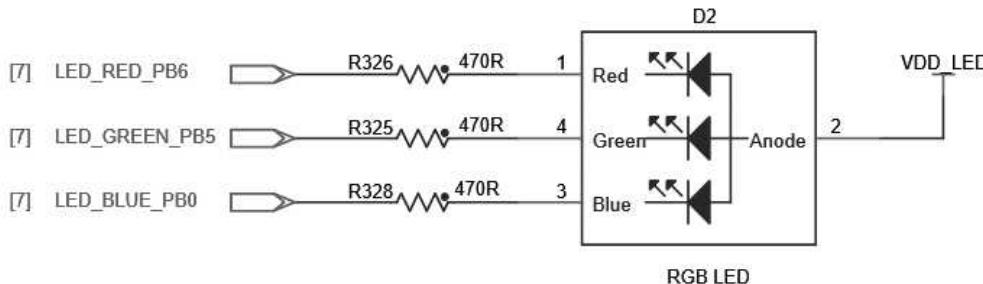
Connect the MCIMX7D-SABRE mikroBUS™ PWM pin to the Color click™ RD pin, MOSI pin to GR, and SCK to BL. Supply VCC = +5V from MCIMX7D-SABRE board to the Color click™ accessory board and connect GND between both boards.

Open the MCIMX7D-SABRE board schematic and find the buttons USR_BT0 and USR_BT1 in pag.21. These buttons will be used to generate the interrupts.

LAB 7.3 Hardware Description for the SAMA5D2 Processor

You will use three pins of the SAMA5D2 to control each LED. These pins must be multiplexed as GPIOs in the DT.

The SAMA5D2B-XULT board integrates an RGB LED. Go to the pag.11 of the SAMA5D2B-XULT schematic to see the RGB LED:

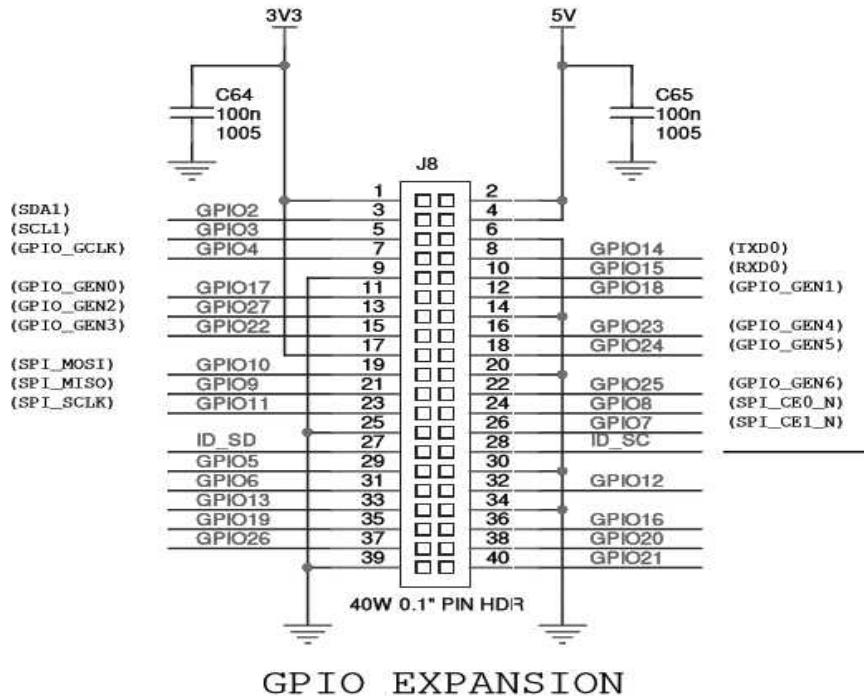


Open the SAMA5D2B-XULT board schematic and find the button BP1 in pag.11. This button will be used to generate one of the interrupts. The second interrupt will be generated using the button of the **MikroElektronika Button R click** board. See the board at <https://www.mikroe.com/button-r-click>. You can download the schematic from that link or from the GitHub repository of this book. Connect the PB25 pad of the SAMA5D2 processor that is available to the pin 30 of the J17 connector (pag. 15 of the SAMA5D2B-XULT board schematic) to the INT pin of the Button R click board.

LAB 7.3 Hardware Description for the BCM2837 Processor

You will use three pins of the BCM2837 to control each LED. These pins must be multiplexed as GPIOs in the DT.

To obtain the GPIOs, you will use the GPIO expansion connector. Go to the Raspberry-Pi-3B-V1.2-Schematics to see the connector:



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

Connect the GPIO EXPANSION 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 schematic from that link or from the GitHub repository of this book. Connect the GPIO23 pin and the GPIO24 pin of the GPIO expansion connector to the INT pin of each Button R click board.

LAB 7.3 Device Tree for the i.MX7D Processor

From the MCIMX7D-SABRE mikroBUS™ socket, you can see that MOSI pin connects to the SAI2_TXC pad of the i.MX7D processor, the SCK pin to the SAI2_RXD pad and the PWM pin to the GPIO1_IO02 pad. You have to configure the SAI2_TXC, SAI2_RXD and GPIO1_IO02 pads as GPIO signals. To look for the macros that assign the required GPIO functionality go to the imx7d-pinfuc.h file under arch/arm/boot/dts/ and find the next macros:

```
#define MX7D_PAD_SAI2_TX_BCLK__GPIO6_IO20      0x0220 0x0490 0x0000 0x5 0x0
#define MX7D_PAD_SAI2_RX_DATA__GPIO6_IO21      0x0224 0x0494 0x0000 0x5 0x0
```

Go now to the imx7d-pinfuc-lpsr.h file under arch/arm/boot/dts/ and find the next macro:

```
#define MX7D_PAD_GPIO1_IO02__GPIO1_IO2 0x0008 0x0038 0x0000 0x0 0x0
```

The USR_BT1 button is connected to the SD2_WP pad of the i.MX7D processor and the USR_BT0 is connected to the SD2_RESET pad of the i.MX7D processor. To look for the macros that assign the required GPIO functionality go to the imx7d-pinfuc.h file under arch/arm/boot/dts/ directory and find the next macros:

```
#define MX7D_PAD_SD2_WP__GPIO5_IO10      0x01B0 0x0420 0x0000 0x5 0x0
#define MX7D_PAD_SD2_RESET_B__GPIO5_IO11 0x01B4 0x0424 0x0000 0x5 0x0
```

Now, you can modify the device tree file imx7d-sdb.dts adding the next code in bold:

```
/ {
    model = "Freescale i.MX7 SabreSD Board";
    compatible = "fsl,imx7d-sdb", "fsl,imx7d";

    memory {
        reg = <0x80000000 0x80000000>;
    };

    [...]

    ledpwm {
        compatible = "arrow,ledpwm";

        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_keys_gpio &pinctrl_gpio_leds &pinctrl_gpio_led>

        bp1 {
```

```
label = "KEY_1";
gpios = <&gpio5 10 GPIO_ACTIVE_LOW>;
trigger = "falling";
};

bp2 {
    label = "KEY_2";
    gpios = <&gpio5 11 GPIO_ACTIVE_LOW>;
    trigger = "falling";
};

ledred {
    label = "led";
    colour = "red";
    gpios = <&gpio1 2 GPIO_ACTIVE_LOW>;
};

ledgreen {
    label = "led";
    colour = "green";
    gpios = <&gpio6 20 GPIO_ACTIVE_LOW>;
};

ledblue {
    label = "led";
    colour = "blue";
    gpios = <&gpio6 21 GPIO_ACTIVE_LOW>;
};

}

[...]

&iomuxc {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_hog_1>

    imx7d-sdb {

        pinctrl_hog_1: hoggrp-1 {
            fsl,pins = <
                MX7D_PAD_EPDC_BDR0__GPIO2_I028      0x59
            >;
        };

        [...]

        pinctrl_keys_gpio: keys_gpiogrp {
            fsl,pins = <
```

You need to be careful not to configure the same pad twice in the device tree. IOMUX configurations are set by the drivers in the order the kernel probes the configured device. If the same pad is configured differently by two drivers, the settings associated with the last-probed

driver will apply. If you check the `ecspi3` node in the device tree file `imx7d-sdb.dts` you can see that the pin configuration defined on the `pinctrl-0` property assigns the "default" name and points to the `pinctrl_ecspi3` and `pinctrl_ecspi3_cs` pin function nodes:

```
pinctrl_ecspi3_cs: ecspi3_cs_grp {  
    fsl,pins = <  
        MX7D_PAD_SD2_CD_B__GPIO5_I09 0x80000000  
        MX7D_PAD_SAI2_TX_DATA__GPIO6_I022 0x2  
    >;  
};  
  
pinctrl_ecspi3: ecspi3grp {  
    fsl,pins = <  
        MX7D_PAD_SAI2_TX_SYNC__ECSPI3_MISO 0x2  
        MX7D_PAD_SAI2_TX_BCLK__ECSPI3_MOSI 0x2  
        MX7D_PAD_SAI2_RX_DATA__ECSPI3_SCLK 0x2  
    >;  
};
```

You can see that the `SAI2_TX_BCLK` and `SAI2_RX_DATA` pads are configured twice by two different drivers. You can comment out the entire definition for `ecspi3` or disable it by changing status to "disabled". See the code below if you choose the second option:

```
&ecspi3 {  
    fsl,spi-num-chipselects = <1>;  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_ecspi3 &pinctrl_ecspi3_cs>;  
    cs-gpios = <&gpio5 9 GPIO_ACTIVE_HIGH>, <&gpio6 22 0>;  
    status = "disabled";  
  
    [...]  
}
```

LAB 7.3 Device Tree for the SAMA5D2 Processor

From the SAMA5D2B-XULT board, you can see that `LED_RED_PB6` pin connects to the `PB6` pad of the SAMA5D2 processor, the `LED_GREEN_PB5` pin to the `PB5` pad and the `LED_BLUE_PB0` pin to the `PB0` pad. You have to configure the `PB6`, `PB5` and `PB0` pads as GPIO signals. To look for the macro that assigns the required functionality (GPIO) go to the `sama5d2-pinfuc.h` file under `arch/arm/boot/dts/` and find the next macros:

```
#define PIN_PB6_GPIO      PINMUX_PIN(PIN_PB6, 0, 0)  
#define PIN_PB5_GPIO      PINMUX_PIN(PIN_PB5, 0, 0)  
#define PIN_PB0_GPIO      PINMUX_PIN(PIN_PB0, 0, 0)
```

For the buttons, you have to configure the PB9 and PB25 pads of the SAMA5D2 processor as GPIO signals. To look for the macros that assign the required functionality (GPIO) go to the sama5d2-pinfunc.h file under linux/arch/arm/boot/dts/ directory and find the next macros:

```
#define PIN_PB9__GPIO      PINMUX_PIN(PIN_PB9, 0, 0)
#define PIN_PB25__GPIO      PINMUX_PIN(PIN_PB25, 0, 0)
```

The PB25 pad is also being multiplexed as a GPIO by the isc node. This node is included in the at91-sama5d2_xplained_ov7670.dtsi file under the arch/arm/boot/dts/ folder. Comment out the following line inside the at91-sama5d2_xplained_common.dtsi file to avoid this "mux" conflict:

```
//#include "at91-sama5d2_xplained_ov7670.dtsi"
```

Now, you can modify the device tree file at91-sama5d2_xplained_common.dtsi adding the next code in bold. Disable the gpio_keys node to avoid "mux" conflict with the PB9 pad:

```
pinctrl@fc038000 {
    pinctrl_adc_default: adc_default {
        pinmux = <PIN_PD23__GPIO>;
        bias-disable;
    };

    [...]

    pinctrl_ledkey_gpio_default: ledkey_gpio_default {
        key {
            pinmux = <PIN_PB25__GPIO>,
            <PIN_PB9__GPIO>;
            bias-pull-up;
        };

        led {
            pinmux = <PIN_PB0__GPIO>,
            <PIN_PB5__GPIO>,
            <PIN_PB6__GPIO>;
            bias-pull-up;
        };
    };

    [...]
}

/ {
    model = "Atmel SAMA5D2 Xplained";
    compatible = "atmel,sama5d2-xplained", "atmel,sama5d2", "atmel,sama5";
    chosen {
```

```
    stdio-path = "serial0:115200n8";
};

[...]

ledpwm {
    compatible = "arrow,ledpwm";

    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ledkey_gpio_default>;

    bp1 {
        label = "PB_KEY";
        gpios = <&pioA 41 GPIO_ACTIVE_LOW>;
        trigger = "falling";
    };

    bp2 {
        label = "MIKROBUS_KEY";
        gpios = <&pioA 57 GPIO_ACTIVE_LOW>;
        trigger = "falling";
    };

    ledred {
        label = "led";
        colour = "red";
        gpios = <&pioA 38 GPIO_ACTIVE_LOW>;
    };

    ledgreen {
        label = "led";
        colour = "green";
        gpios = <&pioA 37 GPIO_ACTIVE_LOW>;
    };

    ledblue {
        label = "led";
        colour = "blue";
        gpios = <&pioA 32 GPIO_ACTIVE_LOW>;
    };
};

[...]
};
```

You can see that the "gpio-leds" driver is configuring the same LEDs. Disable it by changing status to "disabled".

```
leds {  
    compatible = "gpio-leds";  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_led_gpio_default>;  
    status = "disabled";  
  
    red {  
        label = "red";  
        gpios = <&pioA 38 GPIO_ACTIVE_LOW>;  
    };  
  
    green {  
        label = "green";  
        gpios = <&pioA 37 GPIO_ACTIVE_LOW>;  
    };  
  
    blue {  
        label = "blue";  
        gpios = <&pioA 32 GPIO_ACTIVE_LOW>;  
        linux,default-trigger = "heartbeat";  
    };  
};
```

LAB 7.3 Device Tree for the BCM2837 Processor

From the Raspberry Pi 3 Model B board, you have to configure the pads GPIO27, GPIO22, GPIO26, GPIO23 and GPIO24 as GPIO signals.

Modify the device tree file `bcm2710-rpi-3-b.dts` adding the next code in bold:

```
/ {  
    model = "Raspberry Pi 3 Model B";  
};  
  
&gpio {  
    sdhost_pins: sdhost_pins {  
        brcm,pins = <48 49 50 51 52 53>;  
        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";
    };

    expgpio: expgpio {
        compatible = "brcm,bcm2835-expgpio";
        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 info for each of the three led devices. The first field holds the name of each device. The second field ledd holds the gpio descriptor of each specific pin connected to one of the three LEDs. The last field is a pointer to a private struct keyled_priv structure that will hold the global data used for all the led devices. This struct keyled_priv field 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 private struct keyed_priv that will store global info accessible to all the led devices. The first field of the private structure is the num_leds variable, that will hold 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 user and interrupt context tasks. The task pointer variable will point to the struct task_struct returned by the kthread_run() function. The led_class field will point to the struct class returned by the class_create() function; the struct class structure is 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 private struct led_device structures.

```
struct keyed_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. See below an extract of the probe() routine with the main lines of code marked in bold. These are the main points to set up the driver within the probe() function:
 - Declare a pointer to a struct fwnode_handle (struct fwnode_handle *child) and a pointer to the global private structure (struct keyed_priv *priv).
 - Get the number of LEDs and interrupt devices using the device_get_child_node_count() function. You should get five devices returned.
 - Allocate the private structures with devm_kzalloc(). You will allocate space for the global structure and three pointers (you have to set the number of pointers to do the allocation of an array of pointers declared inside a structure) to struct led_device structures (see the sizeof_keyed_priv() function).
 - Allocate three device numbers with alloc_chrdev_region() and create the Keyled class with class_create().
 - Initialize a spinlock with spin_lock_init(); the spinlock will be used to protect access to the shared period variable between interrupt and user contexts tasks. You will

use `spin_lock_irqsave()` in user context and `spin_lock()` inside the ISR when using SMP architectures (i.MX7D and BCM2837). For uniprocessor architectures (SAMA5D2) 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 adquired 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 (see the `led_device_register()` function, which performs this task). You will get the GPIO descriptor of each LED pin declared inside each **DT led node** using the `devm_get_gpiod_from_child()` function, then you will set the direction to output with `gpiod_direction_output()`. The GPIO descriptor of each INT pin declared inside each **DT key node** is obtained using the `devm_get_gpiod_from_child()` function, then the direction of the pin is set to input with `gpiod_direction_input()`. The Linux IRQ numbers are obtained using `gpiod_to_irq()` and both interrupts are allocated 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 keyled_priv *priv;

    count = device_get_child_node_count(dev);

    priv = devm_kzalloc(dev, sizeof_keyled_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
         * used to set the minor number of each device
         * increased to the end of the iteration
         */
        new_led = led_device_register(colour_name,
                                      priv->num_leds,
                                      dev,
                                      priv->led_devt,
                                      priv->led_class);

        new_led->ledd = devm_get_gpiod_from_child(dev, NULL,
                                                 child);

        /* 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, "KEY_1") == 0) {
        keyd = devm_get_gpiod_from_child(dev, NULL, child);
        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, "KEY_2") == 0) {

    keyd = devm_get_gpiod_from_child(dev, NULL, child);
    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;
}
}

dev_info(dev, "i am out of the device tree\n");

/* 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 next 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, that 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 specific struct `led_device` structure is recovered using the `dev_get_drvdata()` function. In the `led_device_register()` function called within `probe()` was previously done the setting between each led device and its `led_device` structure 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 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 driver execution).
- The `blink_on_led_store()` function will receive only 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 performing the LED blinking with a specific period. If there is already a kthread running the function will be exited.
- The `blink_off_led_store()` function will receive only 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 inside the global private structure. See below the ISR that increases the period:

```
static irqreturn_t KEY_ISR1(int irq, void *data)
{
    struct keyled_priv *priv = data;
    priv->period = priv->period + 10;
    if ((priv->period < 10) || (priv->period > 10000))
        priv->period = 10;
    return IRQ_HANDLED;
}
```

8. Write the function thread. Inside the function, you will recover the struct 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 be invoked periodically blinking ON and OFF the specific LED using gpiod_set_value() and the msleep() delay function:

```
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;
        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;
};
```

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 struct 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. Build the modified device tree, and load it to the target processor.

See in the next **Listing 7-3** the "keyled class" driver source code (keyled_imx_class.c) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (keyled_sam_class.c) and BCM2837 (keyled_rpi_class.c) drivers can be downloaded from the GitHub repository of this book.

Listing 7-3: keyled_imx_class.c

```
#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>

#define LED_NAME_LEN      32
#define INT_NUMBER        2
static const char *HELLO_KEYS_NAME1 = "KEY1";
static const char *HELLO_KEYS_NAME2 = "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);

/* blinking ON the specific LED 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 the blinking of any led */
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 local 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 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 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;

    /* First 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's 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 __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;

    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(keyed_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 this 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 keyled class
     * priv->num_leds is 0 for the first iteration
     * used to set the minor number of each device
     * increased to the end of the 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_get_gpiod_from_child(dev, NULL, child);
    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,"KEY_1") == 0) {
    keyd = devm_get_gpiod_from_child(dev, NULL, child);
    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");
}
else if (strcmp(label_name,"KEY_2") == 0) {

    keyd = devm_get_gpiod_from_child(dev, NULL, child);
    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 __exit my_remove(struct platform_device *pdev)
{
    int i;
    struct led_device *led_count;
    struct keyled_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];
        gpiod_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 keyled_class driver that decreases \
    and increases the led flashing period");
```

keyled_imx_class.ko Demonstration

```
root@imx7dsabresd:~# insmod keyled_imx_class.ko /* load module */
root@imx7dsabresd:~# cat /proc/interrupts /* see the linux IRQ numbers (219 and 220)
and hwirq numbers (10 and 11) for the gpio-mxc controller */
root@imx7dsabresd:~# ls /sys/class/keyled/ /* see devices under keyled class */
root@imx7dsabresd:/sys/class/keyled/blue# ls /* see sysfs entries under one of the
devices */
root@imx7dsabresd:/sys/class/keyled/blue# echo on > set_led /* switch on blue led */
root@imx7dsabresd:/sys/class/keyled/red# echo on > set_led /* switch on red led and
switch off blue led */
root@imx7dsabresd:/sys/class/keyled/green# echo on > set_led /* switch on green led
and switch off red led */
root@imx7dsabresd:/sys/class/keyled/green# echo off > set_led /* switch off green
led */
root@imx7dsabresd:/sys/class/keyled/green# echo on > blink_on_led /* start blinking
the green led */
root@imx7dsabresd:/sys/class/keyled/green# echo off > blink_off_led /* stop blinking
the green led */
root@imx7dsabresd:/sys/class/keyled/red# echo on > blink_on_led /* start blinking
the red led */
root@imx7dsabresd:/sys/class/keyled/red# echo 100 > set_period /* change the
blinking period */

"Increase the blinking period pressing the FUNC2 key"

"Decrease the blinking period pressing the FUNC1 key"

root@imx7dsabresd:/sys/class/keyled/red# echo off > blink_off_led /* stop blinking
the red led */
root@imx7dsabresd:~# rmmod keyled_imx_class.ko /* remove the module */
```

<http://www.rejoiceblog.com/>

8

Allocating Memory with Linux Drivers

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. This virtual address space is separated from the actual physical map of memory in the system. You can write, compile, and link applications to run in the virtual memory space. Virtual addresses are those used by you, and the compiler and linker, when placing code in memory. 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.

Good page faults are expected behavior, and occur whenever a program allocates dynamic memory, runs a section of code or writes a section of data for the first time, or increases its stack

size. When the process attempts to access this new memory, the MMU declares a page fault, and Linux adds a fresh page of memory to the process's translation tables. The interrupted process is then resumed.

Bad faults occur when a process follows a NULL pointer, or tries to access memory that it doesn't own. Bad faults can also occur due to programming bugs in the kernel, in which case the handler will print an "oops" message before terminating the process.

Walking ARM MMU Translation Tables

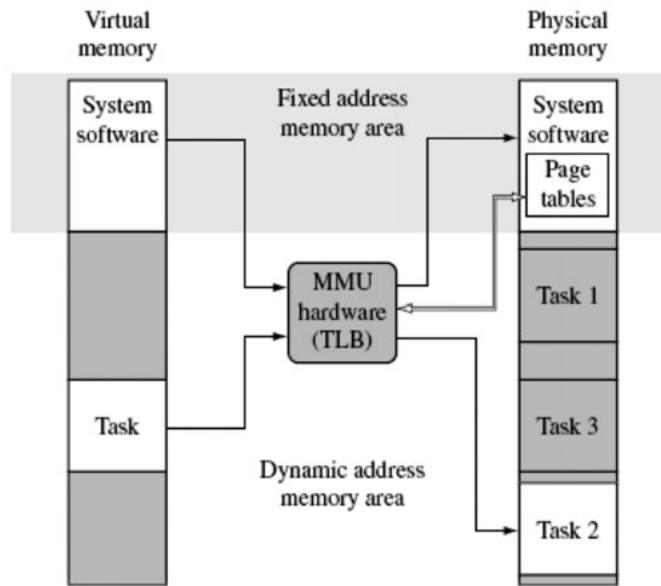
Addresses generated by the processor core are virtual addresses. The MMU essentially replaces the most significant bits of this virtual address with some other value, to generate the physical address (effectively defining a base address of a piece of memory). The lower bits are the same in both addresses (effectively defining an offset in physical memory from that base address). The translation is carried out automatically in hardware and is transparent to the application. In addition to address translation, the MMU controls memory access permissions, memory ordering, and cache policies for each region of memory.

A full translation table lookup is called a translation table walk and can have a significant cost in execution time. To support fine granularity of the VA to PA mapping, a single input address to output address translation can require multiple accesses to the translation tables, with each access giving finer granularity.

Translation Lookaside Buffers (TLBs) reduce the average cost of a memory access by caching the results of translation table walks. TLBs behave as caches of the translation table information.

The Translation Lookaside Buffer (TLB) is a cache of page translations within the MMU. During memory access, the MMU first checks whether the translation is cached in the TLB. If the requested translation is available, we have a TLB hit, and the TLB provides the translation of the physical address immediately. If the TLB does not have a valid translation for that address, we have a TLB miss and an external page table walk is required. This newly loaded translation can then be cached in the TLB for possible reuse.

The exact structure of the TLB differs between implementations of the ARM processors. There are one or more micro-TLBs, which are situated close to the instruction and data caches. Addresses with entries which hit in the micro-TLB require no additional memory look-up and no cycle penalty. However, the micro-TLB has only a small number of mappings (typically eight on the instruction side and eight on the data side). This is backed by a larger main TLB (typically 64 entries), but there may be some penalty associated with accesses which miss in the micro-TLB but which hit in the main TLB.



A translation table walk occurs as the result of a TLB miss, and starts with a read of the appropriate starting-level translation table. The result of that read determines whether additional translation table reads are required.

The SAMA5D2 Cortex-A5 processor supports the ARM v7 VMSA including the TrustZone security extension, and its ARM MMU supports entries in the translation tables, which can represent 1 Mbyte (section), 64 Kbytes (large page) or 4 Kbytes (small page) of virtual memory. The ARM MMU supports a multi-level page table architecture with two levels of translation tables: level 1 (L1) and level 2 (L2). The process in which the MMU accesses page tables to translate addresses is known as page table walking. Page table L1 will translate 1 Mbyte pages, whereas page table L2 will translate 4 Kbytes and 64 Kbytes pages.

When the processor generates a memory access, the MMU:

1. Performs a lookup for the requested virtual address and current ASID and security state in the relevant instruction or data micro TLB.
2. If there is a miss in the micro TLB, performs a lookup for the requested virtual address and current ASID and security state in the main TLB.
3. If there is a miss in main TLB, performs a hardware translation table walk.

The MMU can be configured to perform hardware translation table walks in cacheable regions by setting the IRGN bits in TTRB0 and TTRB1 register.

The L1 page table divides the full 4GB address space into 4096 equally sized 1MB sections. The L1 page table therefore contains 4096 entries, each entry being word sized. Each entry can be the base address of a level 2 page table or a page table entry for translating a 1MB section. If the page table entry is translating a 1MB section, it gives the base address of the 1MB page in physical memory.

The base address of the L1 translation table is known as the Translation Table Base Address and must be aligned to a 16KB boundary. The Translation table locations are defined by the Translation Table Base Registers (TTRB0 and TTRB1). The translation tables are split in two and two TTBR registers are used: TTBR0 points to the level 1 page table location for user space processes and TTBR1 to the level 1 page table for kernel space. During a context switch, only TTBR0 is changed and as a consequence the kernel space pages are always mapped in. There will be different level 1 and level 2 translation tables per process, the location of each level 1 translation table is updated in a context switch via TTBR0. The translation table pointed to by TTBR0 is used by the kernel when the topmost n bits of the virtual address (VA) are all set to 0. The translation table pointed to by TTBR1 is used when the topmost n bits of the VA are all set to 1. The value for n is defined by the Translation Table Base Control Register (TTBCR).

Each task will have its own unique set of translation tables residing in physical memory. Typically, much of the memory system is organized so that the virtual-to-physical address mapping is fixed, with translation table entries that never change. This typically is used to contain operating system code and data. Whenever an application is started, the operating system will allocate itself a set of translation table entries, which map both the actual code and data used by the application to physical memory. If the application requires mapping in code or extra data space (for example through a malloc() call), the kernel can subsequently modify these tables. When a task completes and the application is no longer running, the kernel can remove any associated translation table entries and re-use the space for a new application. Upon a task switch, the kernel switches translation tables to the new ones of the next thread to be run. The expected use of two different set of translation tables managed by TTBR0 and TTBR1 is to reduce the cost of OS context switches by enabling the OS, and each individual task or process, to have its own pagetable without consuming much memory. In this model, the virtual address space is divided into two regions:

- 0x0 -> 1<<(32-N) that TTBR0 controls.
- 1<<(32-N) -> 4GB that TTBR1 controls.

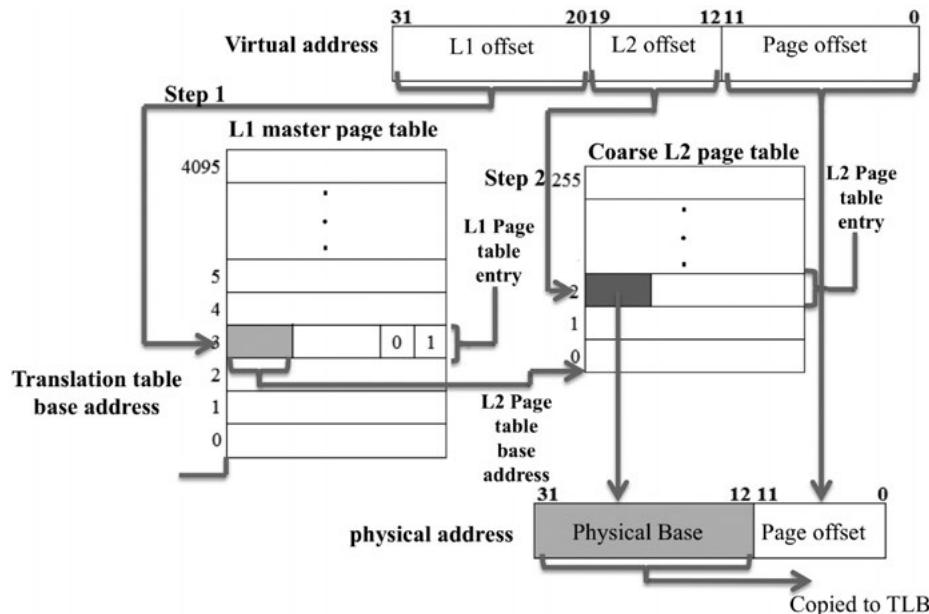
The value of N is set in the TTBCR. If N is zero, then TTBR0 is used for all addresses. If N is not zero, the OS and memory mapped IO are located in the upper part of the memory map, TTBR1, and the tasks or processes all occupy the same virtual address space in the lower part of the memory, TTBR0. This allows one to have a design where the operating system and memory-

mapped I/O are located in the upper part of the address space and managed by the translation table in TTBR1 and user processes are in the lower part of memory and managed by the translation table pointed by TTBR0. On a context switch, the operating system has to change TTBR0 to point to the first-level table location for the new process. The L1 offset of the virtual address will set the index of the new L1 translation table pointed by the TTBR0 register. The TTBR1 register will still contain the memory map for the operating system and memory-mapped I/O.

To locate the relevant entry in the L1 translation table, the top 12 bits ($2^{12} = 4096$) of the virtual address are summed to the Translation Table Base Address (TTBR0 or TTBR1), obtaining the index to one of the 4096 words within the L1 translation table. The upper 20 bits of the L1 translation table entry (Small Page Base Address) select the base address of the L2 translation table. If the L1 entry is translating a 1MB section, it gives the base address of the 1MB page in physical memory using the 16 upper bits of the L1 translation table entry (Large Page Base Address); the lower 16 bits of the virtual address will set the index of the 1MB page in physical memory.

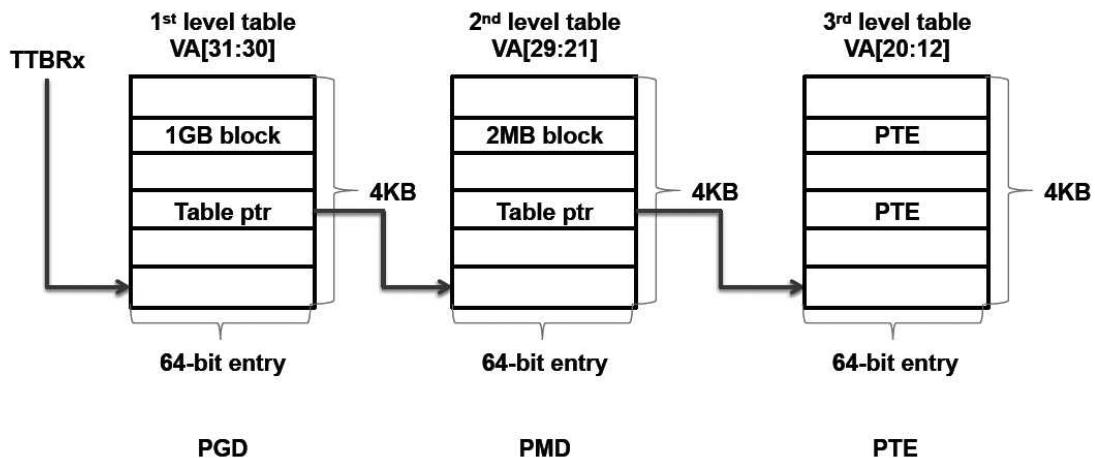
A L2 translation table divides the 1MB section further into either large pages of 64KB or small pages of 4KB size each. Each L2 page table entry is 4 bytes and besides the page base address also contains memory access bits and other information, such as whether data of this page should be cached or executed. There are 256 L2 page table entries per table, thus a L2 page table consumes 1KB of memory. The Bits [19:12] of the virtual address are used to index within the 256 entries in the L2 translation table. The upper 20 bits of the L2 translation table entry select the base address of the 4KB small page in physical memory; the lower 12 bits of the virtual address will set the index of the 4KB page in physical memory.

The next figure shows a general view of the address translation:



Processors that implement the ARMv7-A Large Physical Address Extension (LPAE), like the NXP i.MX7D expand the range of accessible physical addresses from 4GB to 1024GB, a terabyte, by translating 32-bit virtual memory addresses into 40-bit physical memory addresses. To do this they use the Long-descriptor format.

The translation table walking steps are similar to those previously described for processors that do not implement LPAE. Individual translation table entries are now 64-bit in size. With LPAE, there are 3 levels of page tables. Each level has 512 entries of 8 bytes each, occupying a 4K page. The first level table covers a range of 512GB, each entry representing 1GB. Since we are limited to 4GB input address range, only 4 entries in the PGD are used. The first level entries point to the second level translation table. The second label contains 512 entries, each entry representing 2MB and pointing to a third translation label. The third level contains 512 entries, each entry addressing a 4KB range.



The page table definitions have been separated into `pgtable-2level.h` and `pgtable-3level.h` files located under `arch/arm/include/asm/` directory.

Linux uses a four-level paging model:

- **Page Global Directory (PGD):** It is the first level (level 1) page table. Each entry's type is `pgd_t` in kernel (generally an unsigned long), and point on an entry in table at the second level. In kernel, the `struct task_struct` structure represents a process's description, which in turn has a member `mm` whose type is `struct mm_struct`, and that characterizes and represents the process's memory space. In the `struct mm_struct`, there is a processor-specific field `pgd`, which is a pointer on the first entry (entry 0) of the process's level-1 (PGD) page table.

```
struct task_struct {      :include/linux/sched.h
    ...
    struct mm_struct *mm
    ...
}
struct mm_struct{      :include/linux/mm_types.h
    ...
    pgd_t * pgd;
    ...
}
```

- **Page Upper Directory (PUD):** This exists only on architectures using four-level tables. It represents the second level of indirection.
- **Page Middle Directory (PMD):** This is the third indirection level, and exists only on architectures using four-level tables
- **Page Table (PTE):** It is an array of pte_t, where each entry points to the physical page. The SAMA5D2 MMU only supports a 2 level page table (PGD and PTE), whereas the i.MX7D MMU supports a 3 level page table (PGD, PMD and PTE). In arch/arm/mm/proc-v7.S:

```
#ifdef CONFIG_ARM_LPAE
#include "proc-v7-3level.S"
#else
#include "proc-v7-2level.S"
#endif
```

Let's look how is performed the switching of the translation tables during the context switch in a 32-bit ARM without LPAE, which uses 2-Level page table where Level-1: pgd is the page global directory and Level-2: pte is the page table entry.

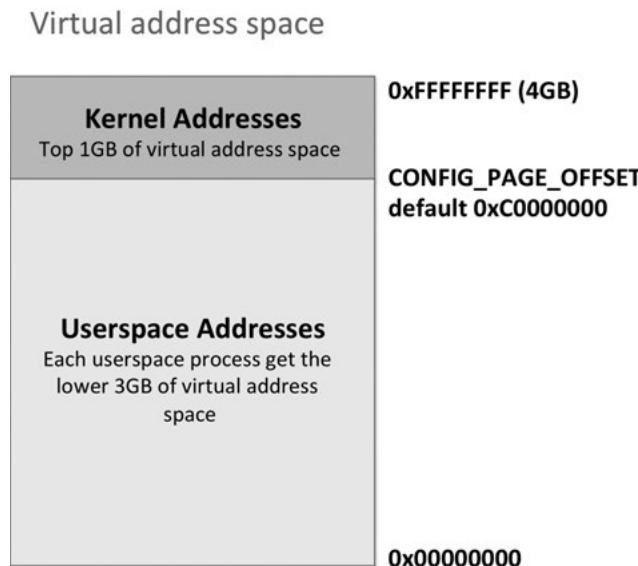
Whenever a context switch happens in Linux, the pgd base of the next process has to be stored in TTBR (note that this is not done while switching to kernel threads as the kernel threads doesn't have a mm struct of its own). As you have seen previously in this section ARM supports two page table trees simultaneously, using the hardware registers TTBR0 and TTBR1. A virtual address is mapped to a physical address by the CPU depending on settings in TTBCR. This control register has a field which sets a split point in the address space. Addresses below the cutoff value are mapped through the page tables pointed to by TTBR0, and addresses above the cutoff value are mapped through TTBR1. TTBR0 is unique per-process, and when a context switch occurs, the kernel sets TTBR0 to the current->mm.pgd for new process. TTBR1 is global for the whole system, and represents the page tables for the kernel. It is referenced in the global kernel variable swapper_pg_dir. Note that both of these addresses are virtual addresses. You can find the physical address of the first-level page table by using virt_to_phys() functions on these addresses.

For ARM, in particular, the second-level page table (PTE) layout is weird. The ARM hardware supports 1K tables at this level (256 entries of 4bytes each). However, the Linux kernel needs some bits that are not provided on some hardware (like DIRTY and YOUNG). These are synthesized by the ARM MMU code via permissions and faulting, effectively making them software-managed. They have to be stored outside the hardware tables, but still in a place where the kernel can get to them easily. In order to keep things aligned into units of 4K, the kernel therefore keeps 2 hardware second-level page tables and 2 parallel arrays (totalling 512 entries) on a single page. When a new second-level page table is created, it is created 512 entries at a time, with the hardware entries at the top of the table, and Linux software entries (synthesized flags and values) at the bottom.

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** - 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** - 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, this 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 using the macros: `__pa(x)` and `__va(x)`.
5. **Kernel Virtual Addresses** - kernel virtual 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, however this virtual memory is only continuous in virtual space, it may not be continuous in physical space. Memory returned by `ioremap()` will also be dynamically placed in this region. Machine specific static mappings are also located here, through `iotable_init()`.

User Process Virtual to Physical Memory Mapping

In Linux, kernel space is constantly present and maps the same physical memory in all processes. Kernel code and data are always addressable, ready to handle interrupts or system calls at any time. By contrast, the mapping for the user-mode portion of the address space changes whenever a process switch happens.

Every user space process, has its own virtual memory layout, containing four logical areas:

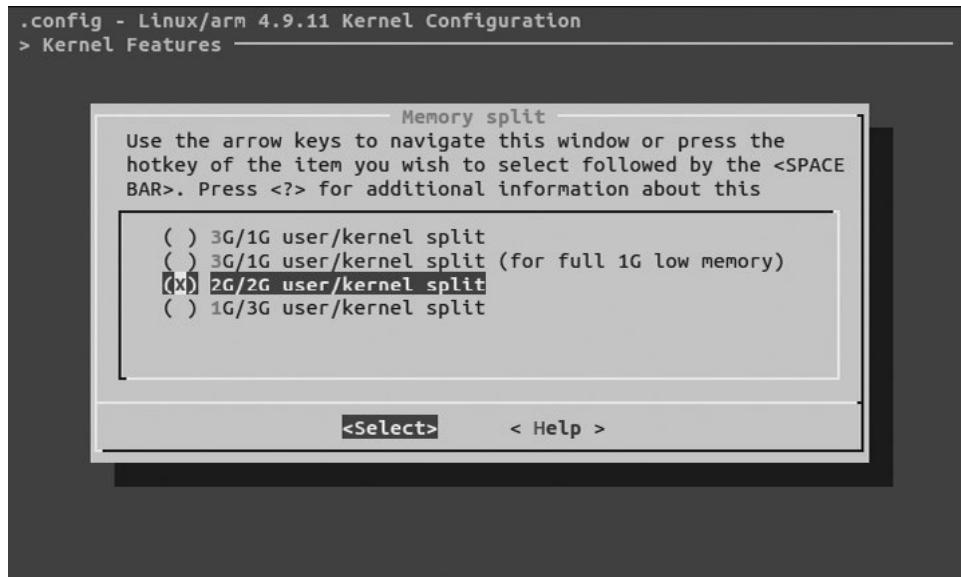
1. **Text segment** - program code, stores the binary image of the process (`./bin/app`).
2. **Data segment (data+bss+heap)** - various data structures created and initialized at the start of a process or while it is running (e.g., heap). The heap provides runtime memory allocation, like the stack, meant 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** - here the kernel maps the contents of files directly to memory. Any application can ask for such a mapping via the Linux `mmap()` system call (including dynamic libraries, for instance `/lib/libc.so`). It is also possible to create an anonymous memory mapping that does not correspond to any files, being used instead for program

data. In Linux, if you request a large block of memory via `malloc()`, the C library will create such an anonymous mapping instead of using heap memory.

4. **Stack segment** - starts near the end of 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. This simple design, possible because the data obeys strict LIFO order, means that no complex data structure is needed to track stack contents, a simple pointer to the top of the stack will do. Pushing and popping are thus very fast and deterministic. Also, the constant reuse of stack regions tends to keep active stack memory in the CPU caches, speeding up access. 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:



The kernel virtual address space is printed in the kernel message buffer during boot:

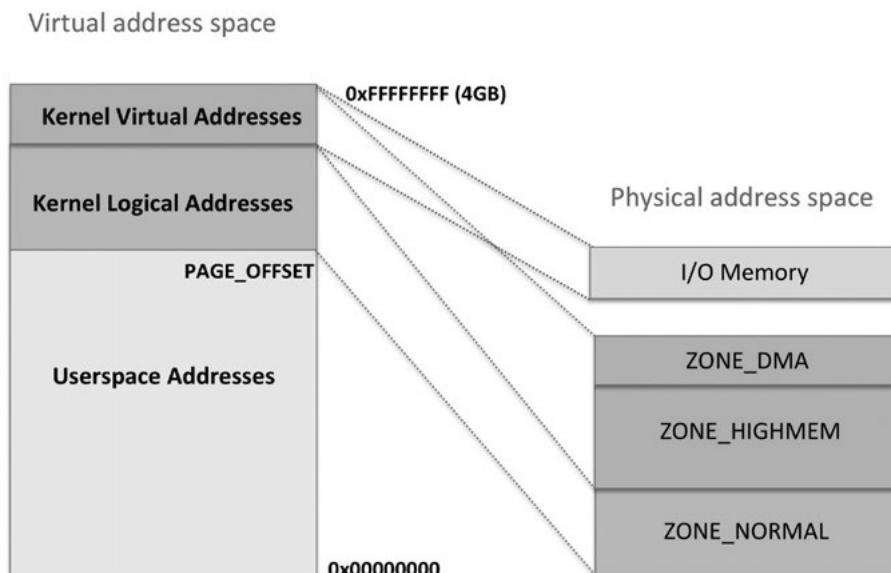
```
Virtual kernel memory layout:
vector  : 0xfffff0000 - 0xfffff1000  ( 4 kB)
fixmap  : 0xfffc00000 - 0xfffff0000  (3072 kB)
vmalloc : 0xc0800000 - 0xff800000  (1008 MB)
```

```
lowmem  : 0x80000000 - 0xc0000000  (1024 MB)
pkmap   : 0x7fe00000 - 0x80000000  (  2 MB)
modules : 0x7f000000 - 0x7fe00000  ( 14 MB)
  .text : 0x80008000 - 0x80a00000  (10208 kB)
  .init : 0x80e00000 - 0x80f00000  (1024 kB)
  .data : 0x80f00000 - 0x80f886e0  ( 546 kB)
  .bss : 0x80f8a000 - 0x810005a0  ( 474 kB)
```

You can split the kernel physical memory into four zones:

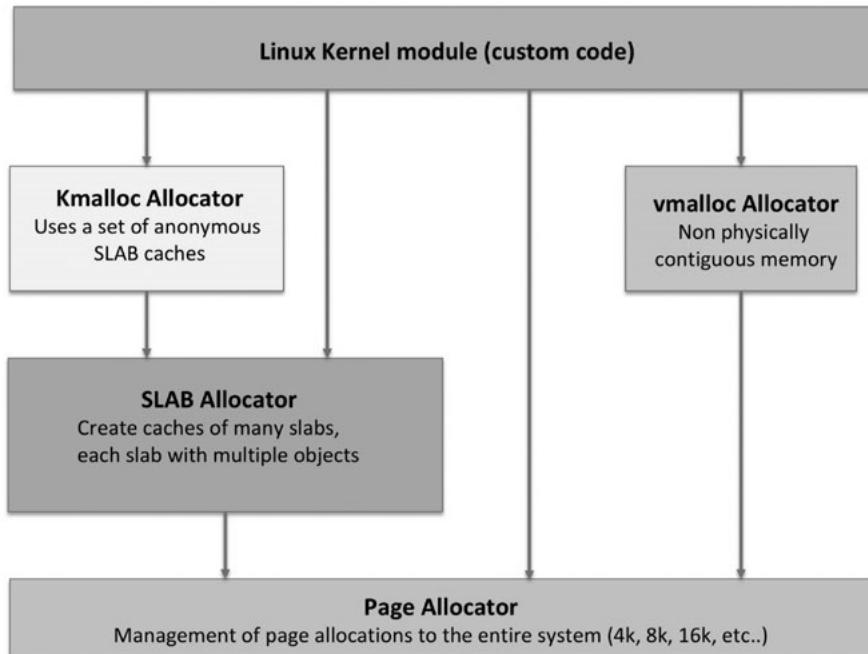
1. **ZONE_DMA** - mapped to the kernel virtual address space (HIGHMEM). In ARM 32-bit CPUs is mapped to kernel virtual addresses ranging from ffc00000 to ffffffff. The mapped virtual DMA memory region is returned by the `dma_alloc_xxx` functions.
2. **ZONE_NORMAL** - mapped to the kernel logical address space (LOWMEM). Used by the kernel for internal data structures as well as other system and user space allocations.
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 `vmalloc` function.
4. **Memory-Mapped I/O** - mapped to kernel virtual address space (HIGHMEM). Memory returned by `ioremap()` will be also 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 name Buddy Allocator is derived from the algorithm this subsystem uses to maintain free page lists. All physical pages in RAM are cataloged by the buddy allocator and grouped into lists. Each list represents clusters of 2^n pages, where n is incremented in each list. If no entries exist on the requested list, an entry from the next list up is broken into two separate clusters and is returned.

to the caller while the other is added to the next list down. When an allocation is returned to the buddy allocator, the reverse process happens. Note that the buddy allocator also manages memory zones, which define pools of memory that have different purposes.

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 */
```

The most common flags are:

- **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 usable for DMA transfers.

```
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 */
```

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 struct 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 on a doubly linked circular list called a "cache chain". In order to reduce fragmentation, the slabs are sorted in three groups:

- Full slabs with zero free objects.
- Partial slabs.
- Empty slabs with no allocated objects.

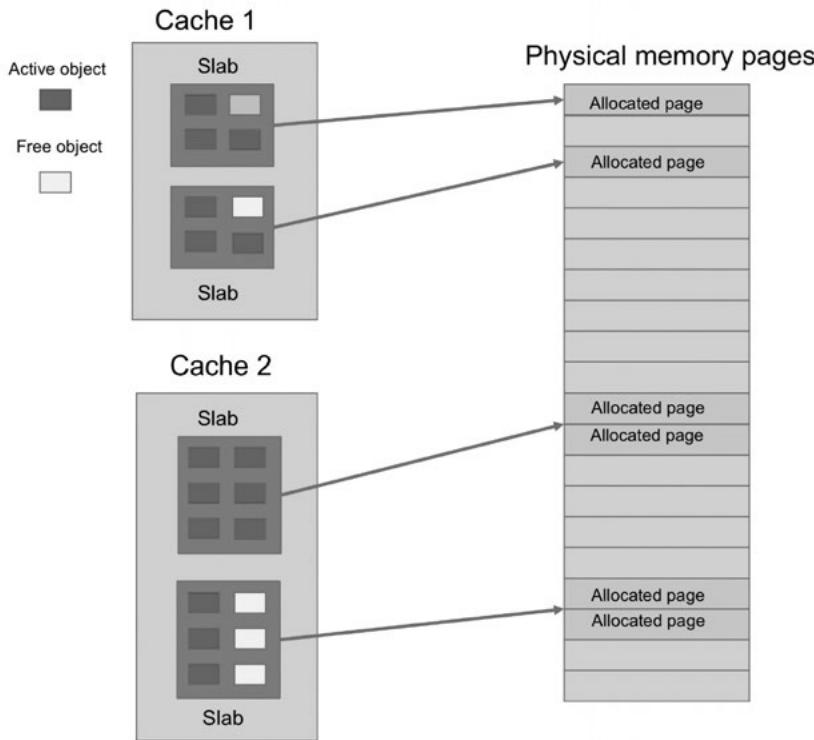
If partial slabs exist, then new allocations come from these slabs, otherwise from empty slabs or new slabs are allocated.

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:** 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:** 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:** the kernel object is the allocated and initialized instance of a C struct. 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*))
```

where:

- **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/slub.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 through the `kmem_cache_create()` function to create a cache of that type struct so that it can satisfy subsequent memory allocations (and releases). Based on the size of the struct, 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 creation of a new object of the same type is needed it makes a request to the SLAB allocator through the `kmem_cache_alloc()` function with the 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 through 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 `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 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 use the same variants as the page allocator.

To free a block of memory previously allocated with `kmalloc()` use the `kfree()` function:

```
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 kernel module lab, you will allocate in 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 driver's `read()` callback function. The reading starts from the first 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 inside 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. Modify the device tree files under arch/arm/boot/dts/ to include your DT driver's 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.

For the **MCIMX7D-SABRE** Board open the DT file imx7d-sdb.dts and add the linked_memory node below the memory node:

```
[...]
/
{
    model = "Freescale i.MX7 SabreSD Board";
    compatible = "fsl,imx7d-sdb", "fsl,imx7d";
    memory {
        reg = <0x80000000 0x80000000>;
    };
}
```

```
linked_memory {  
    compatible = "arrow,memory";  
};  
[...]
```

For the **SAMA5D2B-XULT** Board open the DT file `at91-sama5d2_xplained_common.dtsi` and add the `linked_memory` node below the `gpio_keys` node:

```
[...]  
gpio_keys {  
    compatible = "gpio-keys";  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_key_gpio_default>;  
    bp1 {  
        label = "PB_USER";  
        gpios = <&pioA 41 GPIO_ACTIVE_LOW>;  
        linux,code = <0x104>;  
    };  
};  
linked_memory {  
    compatible = "arrow,memory";  
};  
[...]
```

For the **Raspberry Pi 3 Model B** Board open the DT file `bcm2710-rpi-3-b.dts` and add the `linked_memory` node inside the `soc` node:

```
[...]  
&soc {  
    virtgpio: virtgpio {  
        compatible = "brcm,bcm2835-virtgpio";  
        gpio-controller;  
        #gpio-cells = <2>;  
        firmware = <&firmware>;  
        status = "okay";  
    };  
    expgpio: expgpio {  
        compatible = "brcm,bcm2835-expgpio";  
        gpio-controller;  
        #gpio-cells = <2>;  
        firmware = <&firmware>;  
        status = "okay";  
    };  
    linked_memory {  
        compatible = "arrow,memory";
```

```
};  
[...]
```

6. Build the modified device tree and load it to the target processor.

See in the next **Listing 8-1** the "linked list memory allocation" driver source code (`linkedlist_imx_platform.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`linkedlist_sam_platform.c`) and BCM2837 (`linkedlist_rpi_platform.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 8-1: `linkedlist_imx_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 newListe;  
  
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_imx_platform.ko Demonstration

```
root@imx7dsabresd:~# insmod linkedlist_imx_platform.ko /* load module */
root@imx7dsabresd:~# echo abcdefg > /dev/mydev /* write values to the nodes buffer */
*/
root@imx7dsabresd:~# cat /dev/mydev /* read the values and point to the first node
buffer */
root@imx7dsabresd:~# rmmod linkedlist_imx_platform.ko /* remove module */
```

<http://www.rejoiceblog.com/>

Linux DMA in Device Drivers

Direct Memory Access (DMA) is the hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need to involve the system processor. Use of this mechanism can greatly increase throughput to and from a device, because a great deal of computational overhead is eliminated.

The CPU manages DMA operations via a DMA controller unit. While the DMA transfer is in progress, the CPU can continue executing code. When the DMA transfer is completed, the DMA 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. DMA is a particular advantage in situations where the blocks to be transferred are large or the transfer is a repetitive operation that would consume a large portion of potentially useful CPU processing time.

Cache Coherency

On processors with a data cache an unwanted side effect of using DMA is the possibility that the contents of the cache are no longer coherent with respect to main memory, which can lead to data corruption problems. Imagine a CPU equipped with a cache and external memory that can be accessed directly by devices using DMA. When the CPU tries to access data X located in the main memory could happen that the current 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 main memory is not correct.

Some processors include a mechanism called bus snooping or cache snooping; the snooping hardware notices when an external DMA transfer refers to main memory using an address that matches data in the cache, and either flushes/invalidates the cache entry so that the DMA transfers

the correct data and the state of the cache entry is updated accordingly. 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 will 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 cache and main memory, but it does allow the application to use any arbitrary region of cached memory as a data buffer.

Linux kernel provides two struct `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 slave DMA API usage consists of following steps:

- 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

The details of these operations are:

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 `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 `struct dma_slave_config`. 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 `struct 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 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`: Perform 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. For this, see step 5, `dma_async_issue_pending()`.

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 channel is idle then the first transaction in 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 completion callback routine for notification, if set.

```
void dma_async_issue_pending(struct dma_chan *chan)
```

There are several kinds of addresses involved in the Linux DMA API, and it's important to understand the differences. As seen in previous 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 device resources like registers as physical addresses. These are the addresses in `/proc/iomem`. The physical address is not directly useful to a driver; it must use the `ioremap()` function to map the space and produce a virtual address.

I/O devices use a third kind of address: a **bus address**. If a device performs DMA to read or write system memory, the addresses used by the device are bus addresses. In many systems, bus addresses are identical to CPU physical addresses.

If the device supports DMA, the driver sets up a buffer 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 give 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 setup DMA on each individual physical page).

The **Contiguous Memory Allocator** (or CMA), has been developed to allow allocation of big, physically-contiguous memory blocks. Simple in principle, it has grown quite complicated, requiring cooperation between many subsystems. CMA is integrated with the DMA subsystem, and is accessible using `dma_alloc_coherent()` DMA API. In fact, device drivers should never need to call the CMA API directly, since instead of bus addresses and kernel mappings it operates on pages and page frame numbers (PFNs), and provides no mechanism for maintaining cache coherency.

The CMA has to be integrated with the DMA subsystem of a given architecture. This is performed in two steps:

1. CMA works by reserving memory early at boot time. This memory, called a CMA area or a CMA context, is later returned to the buddy allocator so that it can be used by regular applications. To do the reservation, one needs to call:

```
void dma_contiguous_reserve(phys_addr_t limit);
```

just after the low-level "memblock" allocator is initialized but prior to the buddy allocator setup. On ARM, for example, it is called in `arm_memblock_init()`. The amount of reserved memory depends on a few Kconfig options and a `cma` kernel parameter. The `dma_contiguous_reserve()` function will reserve memory and prepare it to be used with

CMA. On some architectures (eg. ARM) some architecture-specific work needs to be performed as well. To allow that, CMA will call the following function:

```
void dma_contiguous_early_fixup(phys_addr_t base, unsigned long size);
```

2. The second thing to do is to change the architecture's DMA implementation to use the whole machinery. To allocate CMA memory one uses:

```
struct page *dma_alloc_from_contiguous(struct device *dev, int count,
                                       unsigned int align);
```

Its first argument is a device that the allocation is performed on behalf of. The second specifies the number of pages (not bytes or order) to allocate. The third argument is the alignment expressed as a page order. The return value is the first of a sequence of count allocated pages.

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 using `dma_alloc_coherent()`. The kernel allocates a suitable buffer and sets the mapping for the driver. Can simultaneously be accessed by the CPU and device, so this has to be in a cache coherent memory area. Usually 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 will 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 phys to virt mapping for cpu.

If the architecture is non-coherent `dma_alloc_coherent()` will make the memory uncached, so that coherency is maintained. This 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 access it. The argument `dev` is a struct device pointer. This function may be called in an interrupt context with the `GFP_ATOMIC` flag. The argument `size` is the length of the region you want to allocate in bytes and `gfp` is a standard GFP flag. The `dma_alloc_coherent()` function returns two values: the virtual address `cpu_addr`, 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 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 needed using `dma_map_single()` and `dma_unmap_single()`. This is different from coherent mapping because the mappings deal with addresses that were chosen apriori, 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 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 struct 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 DMA bus address returned for the device.

To unmap the memory region:

```
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 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

You will now develop your first kernel DMA module. This driver will allocate two kernel buffers, 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 values 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>

/*
 * To enumerate peripheral types. Used for NXP SDMA controller.
 */
#include <linux/platform_data/dma-imx.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>
```

2. Create a private structure that will store the DMA device specific information. In this driver, you will handle a char device, so a struct miscdevice 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 struct 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 for synchronization of the two tasks, with code such as:

```
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 available almost all the time; if there is significant contention for the semaphore, performance suffers and the locking scheme needs to be reviewed. So semaphores have been heavily optimized for the available case. When used to communicate task completion in the way shown above, however, 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 clear intent of the code, but also more efficient code as both threads can continue until the result is actually needed.

3. Despite the generic DMA engine API, it could be needed to provide a custom data structure for the specific processor's DMA controller. In the i.MX7D processor the struct imx_dma_data is initialized and passed as an argument to the dma_request_channel() function within the probe() function:

```
static int __init my_probe(struct platform_device *pdev)
{
    [...]
    struct imx_dma_data m2m_dma_data = {0};
    [...]
    m2m_dma_data.peripheral_type = IMX_DMATYPE_MEMORY;
    m2m_dma_data.priority = DMA_PRIO_HIGH;

    dma_device->dma_m2m_chan = dma_request_channel(dma_m2m_mask,
                                                    dma_m2m_filter,
                                                    &m2m_dma_data);

    [...]
}
```

4. In the probe() function set up the capabilities for the channel that will be requested, initialize the struct imx_dma_data structure, allocate the wbuf and rbuf buffers, and request the DMA channel from the DMA engine 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 i.MX7D custom data structure
- 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 your driver's dma_m2m_filter callback function:

```
static bool dma_m2m_filter(struct dma_chan *chan, void *param)
{
    if (!imx_dma_is_general_purpose(chan))
        return false;
    chan->private = param;
    return true;
}
```

Once the channel is obtained, you have to configure it by filling the struct `dma_slave_config` structure with the proper values to do a DMA transaction. Most of the generic information that a slave DMA can use is included in this struct `dma_slave_config` structure. It allows the clients to specify DMA direction, DMA addresses, bus widths, DMA burst lengths, etc. If some DMA controllers have more parameters to be sent then they should try to embed struct `dma_slave_config` in their controller specific structure. That gives flexibility to pass more parameters, if required.

```
static int __init my_probe(struct platform_device *pdev)
{
    /* Create private structure */
    struct dma_private *dma_device;

    dma_cap_mask_t dma_m2m_mask;
    struct imx_dma_data m2m_dma_data = {0};
    struct dma_slave_config dma_m2m_config = {0};

    /* Allocate private structure */
    dma_device = devm_kzalloc(&pdev->dev,
                           sizeof(struct dma_private),
                           GFP_KERNEL);

    /* Create your char device */
    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;

    /* store the DMA device in your private struct */
    dma_device->dev = &pdev->dev;

    /* Allocate the DMA buffers */
    dma_device->wbuf = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
    dma_device->rbuf = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);

    /* Set up the channel capabilities */
    dma_cap_zero(dma_m2m_mask); /* Clear the mask */
    dma_cap_set(DMA_MEMCPY, dma_m2m_mask); /* Set the capability */

    /* Initialize custom DMA processor's controller structure */
    m2m_dma_data.peripheral_type = IMX_DMATYPE_MEMORY;
    m2m_dma_data.priority = DMA_PRIO_HIGH;

    /* Request the DMA channel */
    dma_device->dma_m2m_chan = dma_request_channel(dma_m2m_mask,
                                                   dma_m2m_filter,
                                                   &m2m_dma_data);
```

```
/* Set slave and controller specific parameters */
dma_m2m_config.direction = DMA_MEM_TO_MEM;
dma_m2m_config.dst_addr_width = DMA_SLAVE_BUSWIDTH_4_BYTES;
dmaengine_slave_config(dma_device->dma_m2m_chan, &dma_m2m_config);

retval = misc_register(&dma_device->dma_misc_device);
platform_set_drvdata(pdev, dma_device);

return 0;
}
```

5. Write the sdma_write() function to communicate with user space. This function gets the characters written to the the char device using copy_from_user() and store them in the wbuf buffer. The DMA addresses dma_src and dma_dst are obtained using the dma_map_single() function, that 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() 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 then be submitted using dmaengine_submit().

Finally, issue pending DMA requests and wait for callback notification (see dma_m2m_callback() function). 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;
    struct device *chan_dev;
    dma_cookie_t cookie;
    dma_addr_t dma_src;
    dma_addr_t dma_dst;

    /* Retrieve the private structure */
    dma_priv = container_of(file->private_data,
```

```
        struct dma_private,
        dma_misc_device);

/* Get the channel dev */
dma_dev = dma_priv->dma_m2m_chan->device;
chan_dev = dma_priv->dma_m2m_chan->device->dev;

/* Receive characters from user space and store in wbuf */
if(copy_from_user(dma_priv->wbuf, buf, count)){
        return -EFAULT;
}

/* Get DMA addresses */
dma_src = dma_map_single(chan_dev, dma_priv->wbuf,
                         SDMA_BUF_SIZE, DMA_TO_DEVICE);

dma_dst = dma_map_single(chan_dev, dma_priv->rbuf,
                         SDMA_BUF_SIZE, DMA_FROM_DEVICE);

/* Get a descriptor for the DMA transaction */
dma_m2m_desc = device_prep_dma_memcpy(dma_priv->dma_m2m_chan,
                                       dma_dst,
                                       dma_src,
                                       SDMA_BUF_SIZE, 0);

dev_info(dma_priv->dev, "successful descriptor obtained");

/* Add callback information */
dma_m2m_desc->callback = dma_m2m_callback;
dma_m2m_desc->callback_param = dma_priv;

/* Init the completion event */
init_completion(&dma_priv->dma_m2m_ok);

/* Add DMA operation to the pending queue */
cookie = dmaengine_submit(dma_m2m_desc);

/* Issue DMA transaction */
dma_async_issue_pending(dma_priv->dma_m2m_chan);

/* Wait for completion of the event */
wait_for_completion(&dma_priv->dma_m2m_ok);

/* check the status of the channel */
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);
```

```
/* Unmap after finishing the DMA transaction */
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);

/* Check the buffers (CPU access) after doing unmap */
if (*dma_priv->rbuf) != *(dma_priv->wbuf)) {
    dev_err(dma_priv->dev, "buffer copy failed!\n");
    return -EINVAL;
}

return count;
}
```

6. 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);
}
```

7. Modify the device tree files under arch/arm/boot/dts/ to include your DT driver's 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.

For the **MCIMX7D-SABRE** Board open the DT file imx7d-sdb.dts and add the `sdma_m2m` node below the `/` node:

```
[...]
/
{
    model = "Freescale i.MX7 SabreSD Board";
    compatible = "fsl,imx7d-sdb", "fsl,imx7d";
    memory {
        reg = <0x80000000 0x80000000>;
    };
    sdma_m2m {
        compatible = "arrow,sdma_m2m";
    };
}
[...]
```

For the **SAMA5D2B-XULT** Board open the DT file `at91-sama5d2_xplained_common.dtsi` and add the `sdma_m2m` node below the `gpio_keys` node:

```
[...]  
  gpio_keys {  
    compatible = "gpio-keys";  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_key_gpio_default>;  
    bp1 {  
      label = "PB_USER";  
      gpios = <&pioA 41 GPIO_ACTIVE_LOW>;  
      linux,code = <0x104>;  
    };  
  };  
  sdma_m2m {  
    compatible = "arrow,sdma_m2m";  
  };  
[...]
```

For the **Raspberry Pi 3 Model B** Board open the DT file `bcm2710-rpi-3-b.dts` and add the `sdma_m2m` node inside the `soc` node:

```
[...]  
&soc {  
  virtgpio: virtgpio {  
    compatible = "brcm,bcm2835-virtgpio";  
    gpio-controller;  
    #gpio-cells = <2>;  
    firmware = <&firmware>;  
    status = "okay";  
  };  
  expgpio: expgpio {  
    compatible = "brcm,bcm2835-expgpio";  
    gpio-controller;  
    #gpio-cells = <2>;  
    firmware = <&firmware>;  
    status = "okay";  
  };  
  sdma_m2m {  
    compatible = "arrow,sdma_m2m";  
  };  
[...]
```

8. Build the modified device tree and load it to the target processor.

See in the next **Listing 9-1** the "streaming DMA" driver source code (`sdma_imx_m2m.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`sdma_sam_m2m.c`) and BCM2837 (`sdma_rpi_m2m.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 9-1: `sdma_imx_m2m.c`

```
#include <linux/module.h>
#include <linux/uaccess.h>
#include <linux/dma-mapping.h>
#include <linux/fs.h>
#include <linux/platform_data/dma-imx.h>
#include <linux/dmaengine.h>
#include <linux/miscdevice.h>
#include <linux/platform_device.h>

/* private structure */
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;
};

/* set the buffer size */
#define SDMA_BUF_SIZE  (1024*63)

/* function to filter a specific DMA channel */
static bool dma_m2m_filter(struct dma_chan *chan, void *param)
{
    if (!imx_dma_is_general_purpose(chan))
        return false;
    chan->private = param;
    return true;
}

/* callback notification handling */
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);
}

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;
    struct device *chan_dev;
    dma_cookie_t cookie;
    dma_addr_t dma_src;
    dma_addr_t dma_dst;

    /* retrieve the private structure */
    dma_priv = container_of(file->private_data,
                           struct dma_private, dma_misc_device);

    /* get the channel dev */
    dma_dev = dma_priv->dma_m2m_chan->device;
    chan_dev = dma_priv->dma_m2m_chan->device->dev;

    /* Receive characters from user space and store in wbuf */
    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);

    /* get DMA addresses */
    dma_src = dma_map_single(chan_dev, dma_priv->wbuf,
                           SDMA_BUF_SIZE, DMA_TO_DEVICE);

    dev_info(dma_priv->dev, "dma_src map obtained");

    dma_dst = dma_map_single(chan_dev, dma_priv->rbuf,
                           SDMA_BUF_SIZE, DMA_FROM_DEVICE);

    dev_info(dma_priv->dev, "dma_dst map obtained");

    /* get a descriptor for the DMA transaction */
    dma_m2m_desc = dma_dev->device_prep_dma_memcpy(dma_priv->dma_m2m_chan,
                                                 dma_dst, dma_src,
                                                 SDMA_BUF_SIZE, 0);

    dev_info(dma_priv->dev, "successful descriptor obtained");

    /* add callback notification information */
```

```
dma_m2m_desc->callback = dma_m2m_callback;
dma_m2m_desc->callback_param = dma_priv;

/* init the completion event */
init_completion(&dma_priv->dma_m2m_ok);

/* add DMA operation to the pending queue */
cookie = dmaengine_submit(dma_m2m_desc);

if (dma_submit_error(cookie)){
    dev_err(dma_priv->dev, "Failed to submit DMA\n");
    return -EINVAL;
};

/* issue DMA transaction */
dma_async_issue_pending(dma_priv->dma_m2m_chan);

/* wait for completion of the event */
wait_for_completion(&dma_priv->dma_m2m_ok);

/* check the status of the channel */
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);

/* unmap after finishing the DMA transaction */
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);

/* check the buffers (CPU access) after doing unmap */
if (*(dma_priv->rbuf) != *(dma_priv->wbuf)) {
    dev_err(dma_priv->dev, "buffer copy failed!\n");
    return -EINVAL;
}

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);

return count;
}

struct file_operations dma_fops = {
    write: sdma_write,
};
```

```
static int __init my_probe(struct platform_device *pdev)
{
    int retval;

    /* create private structure */
    struct dma_private *dma_device;
    dma_cap_mask_t dma_m2m_mask;
    struct imx_dma_data m2m_dma_data = {0};
    struct dma_slave_config dma_m2m_config = {0};

    dev_info(&pdev->dev, "platform_probe enter\n");

    /* allocate private structure */
    dma_device = devm_kzalloc(&pdev->dev, sizeof(struct dma_private), GFP_KERNEL);

    /* create your char device */
    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;

    /* allocate wbuf and rbuf buffers */
    dma_device->wbuf = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
    if(!dma_device->wbuf) {
        dev_err(&pdev->dev, "error allocating wbuf !!\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;
    }

    /* set up the channel capabilities */
    dma_cap_zero(dma_m2m_mask); /* Clear the mask */
    dma_cap_set(DMA_MEMCPY, dma_m2m_mask); /* Set the capability */

    /* initialize custom DMA processor's controller structure */
    m2m_dma_data.peripheral_type = IMX_DMATYPE_MEMORY;
    m2m_dma_data.priority = DMA_PRIO_HIGH;

    /* request the DMA channel */
    dma_device->dma_m2m_chan = dma_request_channel(dma_m2m_mask,
                                                    dma_m2m_filter,
                                                    &m2m_dma_data);
    if (!dma_device->dma_m2m_chan) {
```

```
    dev_err(&pdev->dev,
            "Error opening the SDMA memory to memory channel\n");
    return -EINVAL;
}

/* set slave and controller specific parameters */
dma_m2m_config.direction = DMA_MEM_TO_MEM;
dma_m2m_config.dst_addr_width = DMA_SLAVE_BUSWIDTH_4_BYTES;
dmaengine_slave_config(dma_device->dma_m2m_chan, &dma_m2m_config);

retval = misc_register(&dma_device->dma_misc_device);
if (retval) return retval;

/*
 * attach the private structure to the pdev structure
 * to recover it in each remove() function call
 */
platform_set_drvdata(pdev, dma_device);

dev_info(&pdev->dev, "platform_probe exit\n");

return 0;
}

static int __exit 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_imx_m2m.ko Demonstration

```
root@imx7dsabresd:~# insmod sdma_imx_m2m.ko /* load module */
root@imx7dsabresd:~# echo abcdefg > /dev/sdma_test /* write values to the wbuf
buffer, start DMA transaction that copies values from wbuf to rbuf and compares both
buffers values */
root@imx7dsabresd:~# rmmod sdma_imx_m2m.ko /* remove module */
```

DMA Scatter/Gather Mappings

The allocated buffer can be fragmented in the physical memory and does not need to be allocated contiguously. These allocated physical memory blocks are mapped to a contiguous buffer in the calling process virtual address space, thus enabling easy access to the allocated physical memory blocks.

There are different ways to send contents of several buffers over DMA. They can be sent one at a time mapping each or with scatter/gather, can be sent them all at once (speed). Many devices can accept a scatterlist of array pointers and lengths. Scatterlist entries must be the size of a page (except ends).

Scatter/gather I/O allows the system to perform DMA I/O operations on buffers which are scattered throughout physical memory. Consider, for example, the case of a large (multi-page) buffer created in user space. The application sees a continuous range of virtual addresses, but the physical pages behind those addresses will almost certainly not be adjacent to each other. If that buffer is to be written to a device in a single I/O operation, one of two things must be done: (1) the data must be copied into a physically-contiguous buffer, or (2) the device must be able to work with a list of physical addresses and lengths, grabbing the right amount of data from each segment. Scatter/gather I/O, by eliminating the need to copy data into contiguous buffers, can greatly increase the efficiency of I/O operations while simultaneously getting around the problem generated by the creation of large, physically-contiguous buffers. In order to set up a scatterlist mapping, you should:

1. First, create a struct scatterlist structure to perform a scatterlist DMA transfer:

```
struct scatterlist *sg
```

The struct scatterlist structure is defined in `include/linux/scatterlist.h` as follows:

```
struct scatterlist {  
    unsigned long page_link;  
    unsigned int offset;  
    unsigned int length;  
    dma_addr_t dma_address;  
    unsigned int dma_length;  
};
```

2. Initialize a scatterlist array with:

```
void sg_init_table(struct scatterlist *sg, unsigned int nents)
```

Here, `sg` points to the allocated array, and `nents` is the number of allocated scatter/gather entries.

Set each entry in the allocated array `sg` to point at given data:

```
void sg_set_buf(struct scatterlist *sg, const void *buf, unsigned int buf_len)
```

Here, buf is the virtual address pointer of your allocated buffer and buflen is the length of your allocated buffer.

3. Map the scatterlist to get the DMA addresses. For each buffer in the input scatterlist, `dma_map_sg()` determines the proper bus address to give to the device:

```
int dma_map_sg(device, sg, nent, direction)
```

Returns the number of DMA buffers to send (\leq nent).

4. Once the transfer is complete, a scatter/gather mapping is unmapped with a call to `dma_unmap_sg()`:

```
void dma_unmap_sg(device, sg, nent, direction)
```

LAB 9.2: "scatter/gather DMA device" Module

In this second DMA lab, you are going to develop a kernel module that will send the contents from three "wbuf" buffers to three "rbuf" buffers at the same time using a scatter/gather DMA.

The driver will create four scatter lists, two of them with three buffer entries and another two with just one entry, then will allocate six buffers (wbuf, wbuf2, wbuf3 and rbuf, rbuf2, rbuf3) using the `kzalloc()` function. The driver will fill the wbuf buffers with some selected values. Next, it will receive characters from user space and store them in the `dma_src_coherent` buffer, which is allocated using the coherent allocation method.

After allocating all the buffers and storing the characters in the transmission side buffers, the driver will set up a first DMA transaction from wbuf's to rbuf's. Then, it will set up a second DMA transaction from `dma_src_coherent` to `dma_dst_coherent`.

After each DMA transaction, the driver will check that all the values in the source and destination buffers are identical.

It will be described now the main code sections that differ from the previous lab 9.1 driver:

1. Create the scatter list structure arrays and buffer pointer variables:

```
static dma_addr_t dma_dst;
static dma_addr_t dma_src;
static char *dma_dst_coherent;
static char *dma_src_coherent;
static unsigned int *wbuf, *wbuf2, *wbuf3;
static unsigned int *rbuf, *rbuf2, *rbuf3;
static struct scatterlist sg3[1],sg4[1];
static struct scatterlist sg[3],sg2[3];
```

2. Get the virtual buffer addresses inside the probe() function:

```
wbuf = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
wbuf2 = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
wbuf3 = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);

rbuf = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
rbuf2 = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
rbuf3 = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);

dma_dst_coherent = dma_alloc_coherent(&pdev->dev, SDMA_BUF_SIZE,
                                      &dma_dst, GFP_DMA);
dma_src_coherent = dma_alloc_coherent(&pdev->dev, SDMA_BUF_SIZE,
                                      &dma_src, GFP_DMA);
```

3. In sdma_write() fill in the wbuf, wbuf2, and wbuf3 with some selected values:

```
index1 = wbuf;
index2 = wbuf2;
index3 = wbuf3;

for (i=0; i<SDMA_BUF_SIZE/4; i++) {
    *(index1 + i) = 0x12345678;
}
for (i=0; i<SDMA_BUF_SIZE/4; i++) {
    *(index2 + i) = 0x87654321;
}
for (i=0; i<SDMA_BUF_SIZE/4; i++) {
    *(index3 + i) = 0xabcd012;
}
```

4. In sdma_write() initialize scatterlist arrays with sg_init_table(), and set each entry in the allocated arrays with sg_set_buf() to point at given data. You should provide the exact DMA direction if you know it. Map the scatterlist to get the DMA addresses using the dma_map_sg() function.

```
sg_init_table(sg, 3);
sg_set_buf(&sg[0], wbuf, SDMA_BUF_SIZE);
sg_set_buf(&sg[1], wbuf2, SDMA_BUF_SIZE);
sg_set_buf(&sg[2], wbuf3, SDMA_BUF_SIZE);
dma_map_sg(dma_dev->dev, sg, 3, DMA_TO_DEVICE);

sg_init_table(sg2, 3);
sg_set_buf(&sg2[0], rbuf, SDMA_BUF_SIZE);
sg_set_buf(&sg2[1], rbuf2, SDMA_BUF_SIZE);
sg_set_buf(&sg2[2], rbuf3, SDMA_BUF_SIZE);
dma_map_sg(dma_dev->dev, sg2, 3, DMA_FROM_DEVICE);

sg_init_table(sg3, 1);
sg_set_buf(sg3, dma_src_coherent, SDMA_BUF_SIZE);
```

```
dma_map_sg(dma_dev->dev, sg3, 1, DMA_TO_DEVICE);
sg_init_table(sg4, 1);
sg_set_buf(sg4, dma_dst_coherent, SDMA_BUF_SIZE);
dma_map_sg(dma_dev->dev, sg4, 1, DMA_FROM_DEVICE);
```

5. In sdma_write() get the channel descriptors, add DMA operations to the pending queue, issue pending DMA requests, set and wait for callback notifications. After each DMA transaction compare the values of the transmission and reception buffers.

```
/* Get the DMA descriptor for the first DMA transaction */
dma_m2m_desc = dma_dev->device_prep_dma_sg(dma_m2m_chan, sg2, 3, sg, 3, 0);

dma_m2m_desc->callback = dma_sg_callback;
dmaengine_submit(dma_m2m_desc);
dma_async_issue_pending(dma_m2m_chan);
wait_for_completion(&dma_m2m_ok);
dma_unmap_sg(dma_dev->dev, sg, 3, DMA_TO_DEVICE);
dma_unmap_sg(dma_dev->dev, sg2, 3, DMA_FROM_DEVICE);

/* compare values of the first transaction */
for (i=0; i<SDMA_BUF_SIZE/4; i++) {
    if (*(rbuf+i) != *(wbuf+i)) {
        pr_info("buffer 1 copy failed!\n");
        return -EINVAL;
    }
}
pr_info("buffer 1 copy passed!\n");

for (i=0; i<SDMA_BUF_SIZE/4; i++) {
    if (*(rbuf2+i) != *(wbuf2+i)) {
        pr_info("buffer 2 copy failed!\n");
        return -EINVAL;
    }
}
pr_info("buffer 2 copy passed!\n");

for (i=0; i<SDMA_BUF_SIZE/4; i++) {
    if (*(rbuf3+i) != *(wbuf3+i)) {
        pr_info("buffer 3 copy failed!\n");
        return -EINVAL;
    }
}
pr_info("buffer 3 copy passed!\n");

reinit_completion(&dma_m2m_ok);

/* Get the DMA descriptor for the second DMA transaction */
dma_m2m_desc = dma_dev->device_prep_dma_sg(dma_m2m_chan, sg4, 1, sg3, 1, 0);
dma_m2m_desc->callback = dma_m2m_callback;
dmaengine_submit(dma_m2m_desc);
```

```
dma_async_issue_pending(dma_m2m_chan);
wait_for_completion(&dma_m2m_ok);
dma_unmap_sg(dma_dev->dev, sg3, 1, DMA_TO_DEVICE);
dma_unmap_sg(dma_dev->dev, sg4, 1, DMA_FROM_DEVICE);

/* compare values of the first transaction */
if (*(dma_src_coherent) != *(dma_dst_coherent)) {
    pr_info("buffer copy failed!\n");
    return -EINVAL;
}
pr_info("buffer coherent sg copy passed!\n");
```

6. You will use the same DT node of the lab 9.1.

```
sdma_m2m {
    compatible = "arrow,sdma_m2m";
};
```

See in the next **Listing 9-2** the "scatter/gather DMA device" driver source code (sdma_imx_m2m.c) for the i.MX7D processor.

Note: This driver has only been implemented in the i.MX7D processor.

Listing 9-2: sdma_imx_sg_m2m.c

```
#include <linux/module.h>
#include <linux/uaccess.h>
#include <linux/dma-mapping.h>
#include <linux/fs.h>
#include <linux/platform_data/dma-imx.h>
#include <linux/dmaengine.h>
#include <linux/miscdevice.h>
#include <linux/platform_device.h>

static dma_addr_t dma_dst;
static dma_addr_t dma_src;
static char *dma_dst_coherent;
static char *dma_src_coherent;
static unsigned int *wbuf, *wbuf2, *wbuf3;
static unsigned int *rbuf, *rbuf2, *rbuf3;

static struct dma_chan *dma_m2m_chan;

static struct completion dma_m2m_ok;

static struct scatterlist sg3[1],sg4[1];
static struct scatterlist sg[3],sg2[3];
```

```
#define SDMA_BUF_SIZE  (63*1024)

static bool dma_m2m_filter(struct dma_chan *chan, void *param)
{
    if (!imx_dma_is_general_purpose(chan))
        return false;
    chan->private = param;
    return true;
}

static void dma_sg_callback(void *data)
{
    pr_info("%s\n finished SG DMA transaction\n",__func__);
    complete(&dma_m2m_ok);
}

static void dma_m2m_callback(void *data)
{
    pr_info("%s\n finished DMA coherent transaction\n",__func__);
    complete(&dma_m2m_ok);
}

static ssize_t sdma_write(struct file * filp, const char __user * buf,
                         size_t count, loff_t * offset)
{
    unsigned int *index1, *index2, *index3, i;
    struct dma_async_tx_descriptor *dma_m2m_desc;
    struct dma_device *dma_dev;
    dma_dev = dma_m2m_chan->device;

    pr_info("sdma_write is called.\n");

    index1 = wbuf;
    index2 = wbuf2;
    index3 = wbuf3;

    for (i=0; i<SDMA_BUF_SIZE/4; i++) {
        *(index1 + i) = 0x12345678;
    }

    for (i=0; i<SDMA_BUF_SIZE/4; i++) {
        *(index2 + i) = 0x87654321;
    }

    for (i=0; i<SDMA_BUF_SIZE/4; i++) {
        *(index3 + i) = 0xabcd012;
    }
}
```

```
init_completion(&dma_m2m_ok);

if(copy_from_user(dma_src_coherent, buf, count)){
    return -EFAULT;
}

pr_info ("The string is %s\n", dma_src_coherent);

sg_init_table(sg, 3);
sg_set_buf(&sg[0], wbuf, SDMA_BUF_SIZE);
sg_set_buf(&sg[1], wbuf2, SDMA_BUF_SIZE);
sg_set_buf(&sg[2], wbuf3, SDMA_BUF_SIZE);
dma_map_sg(dma_dev->dev, sg, 3, DMA_TO_DEVICE);

sg_init_table(sg2, 3);
sg_set_buf(&sg2[0], rbuf, SDMA_BUF_SIZE);
sg_set_buf(&sg2[1], rbuf2, SDMA_BUF_SIZE);
sg_set_buf(&sg2[2], rbuf3, SDMA_BUF_SIZE);
dma_map_sg(dma_dev->dev, sg2, 3, DMA_FROM_DEVICE);

sg_init_table(sg3, 1);
sg_set_buf(sg3, dma_src_coherent, SDMA_BUF_SIZE);
dma_map_sg(dma_dev->dev, sg3, 1, DMA_TO_DEVICE);
sg_init_table(sg4, 1);
sg_set_buf(sg4, dma_dst_coherent, SDMA_BUF_SIZE);
dma_map_sg(dma_dev->dev, sg4, 1, DMA_FROM_DEVICE);

dma_m2m_desc = dma_dev->device_prep_dma_sg(dma_m2m_chan,
                                             sg2, 3,
                                             sg, 3, 0);

dma_m2m_desc->callback = dma_sg_callback;
dmaengine_submit(dma_m2m_desc);
dma_async_issue_pending(dma_m2m_chan);
wait_for_completion(&dma_m2m_ok);
dma_unmap_sg(dma_dev->dev, sg, 3, DMA_TO_DEVICE);
dma_unmap_sg(dma_dev->dev, sg2, 3, DMA_FROM_DEVICE);

for (i=0; i<SDMA_BUF_SIZE/4; i++) {
    if (*(rbuf+i) != *(wbuf+i)) {
        pr_info("buffer 1 copy failed!\n");
        return -EINVAL;
    }
}
pr_info("buffer 1 copy passed!\n");

for (i=0; i<SDMA_BUF_SIZE/4; i++) {
    if (*(rbuf2+i) != *(wbuf2+i)) {
```

```
        pr_info("buffer 2 copy failed!\n");
        return -EINVAL;
    }
}
pr_info("buffer 2 copy passed!\n");

for (i=0; i<SDMA_BUF_SIZE/4; i++) {
    if (*(rbuf3+i) != *(wbuf3+i)) {
        pr_info("buffer 3 copy failed!\n");
        return -EINVAL;
    }
}
pr_info("buffer 3 copy passed!\n");

reinit_completion(&dma_m2m_ok);

dma_m2m_desc = dma_dev->device_prep_dma_sg(dma_m2m_chan,
                                              sg4, 1,
                                              sg3, 1, 0);

dma_m2m_desc->callback = dma_m2m_callback;
dmaengine_submit(dma_m2m_desc);
dma_async_issue_pending(dma_m2m_chan);
wait_for_completion(&dma_m2m_ok);
dma_unmap_sg(dma_dev->dev, sg3, 1, DMA_TO_DEVICE);
dma_unmap_sg(dma_dev->dev, sg4, 1, DMA_FROM_DEVICE);

if (*(dma_src_coherent) != *(dma_dst_coherent)) {
    pr_info("buffer copy failed!\n");
    return -EINVAL;
}
pr_info("buffer coherent sg copy passed!\n");
pr_info("dma_src_coherent is %s\n", dma_src_coherent);
pr_info("dma_dst_coherent is %s\n", dma_dst_coherent);

return count;
}

struct file_operations dma_fops = {
    write: sdma_write,
};

static struct miscdevice dma_miscdevice = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "sdma_test",
    .fops = &dma_fops,
};
```

```
static int __init my_probe(struct platform_device *pdev)
{
    int retval;
    dma_cap_mask_t dma_m2m_mask;
    struct imx_dma_data m2m_dma_data = {0};
    struct dma_slave_config dma_m2m_config = {0};

    pr_info("platform_probe enter\n");
    retval = misc_register(&dma_miscdevice);
    if (retval) return retval;

    pr_info("mydev: got minor %i\n",dma_miscdevice.minor);

    wbuf = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
    if(!wbuf) {
        pr_info("error wbuf !!!!!!!\n");
        return -ENOMEM;
    }

    wbuf2 = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
    if(!wbuf2) {
        pr_info("error wbuf !!!!!!!\n");
        return -ENOMEM;
    }

    wbuf3 = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
    if(!wbuf3) {
        pr_info("error wbuf2 !!!!!!!\n");
        return -ENOMEM;
    }

    rbuf = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
    if(!rbuf) {
        pr_info("error rbuf !!!!!!!\n");
        return -ENOMEM;
    }

    rbuf2 = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
    if(!rbuf2) {
        pr_info("error rbuf2 !!!!!!!\n");
        return -ENOMEM;
    }

    rbuf3 = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
    if(!rbuf3) {
        pr_info("error rbuf2 !!!!!!!\n");
        return -ENOMEM;
    }
}
```

```
dma_dst_coherent = dma_alloc_coherent(&pdev->dev, SDMA_BUF_SIZE,
                                         &dma_dst, GFP_DMA);
if (dma_dst_coherent == NULL) {
    pr_err("dma_alloc_coherent failed\n");
    return -ENOMEM;
}

dma_src_coherent = dma_alloc_coherent(&pdev->dev, SDMA_BUF_SIZE,
                                         &dma_src, GFP_DMA);
if (dma_src_coherent == NULL) {
    dma_free_coherent(&pdev->dev, SDMA_BUF_SIZE,
                      dma_dst_coherent, dma_dst);
    pr_err("dma_alloc_coherent failed\n");
    return -ENOMEM;
}

dma_cap_zero(dma_m2m_mask);
dma_cap_set(DMA_MEMCPY, dma_m2m_mask);
m2m_dma_data.peripheral_type = IMX_DMATYPE_MEMORY;
m2m_dma_data.priority = DMA_PRIO_HIGH;

dma_m2m_chan = dma_request_channel(dma_m2m_mask,
                                    dma_m2m_filter,
                                    &m2m_dma_data);
if (!dma_m2m_chan) {
    pr_err("Error opening the SDMA memory to memory channel\n");
    return -EINVAL;
}

dma_m2m_config.direction = DMA_MEM_TO_MEM;
dma_m2m_config.dst_addr_width = DMA_SLAVE_BUSWIDTH_4_BYTES;
dmaengine_slave_config(dma_m2m_chan, &dma_m2m_config);

return 0;
}

static int __exit my_remove(struct platform_device *pdev)
{
    misc_deregister(&dma_misdevice);
    dma_release_channel(dma_m2m_chan);
    dma_free_coherent(&pdev->dev, SDMA_BUF_SIZE,
                      dma_dst_coherent, dma_dst);
    dma_free_coherent(&pdev->dev, SDMA_BUF_SIZE,
                      dma_src_coherent, dma_src);
    pr_info("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 __init demo_init(void)
{
    int ret_val;

    ret_val = platform_driver_register(&my_platform_driver);
    if (ret_val !=0)
    {
        pr_err("platform value returned %d\n", ret_val);
        return ret_val;
    }

    return 0;
}

static void __exit demo_exit(void)
{
    platform_driver_unregister(&my_platform_driver);
}

module_init(demo_init);
module_exit(demo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a SDMA scatter/gather memory to memory driver");
```

sdma_imx_sg_m2m.ko Demonstration

```
root@imx7dsabresd:~# insmod sdma_imx_sg_m2m.ko /* load module */
root@imx7dsabresd:~# echo abcdefg > /dev/sdma_test /* write selected values to the
dbuf buffers, store characters written to the terminal into the dma_src_coherent
buffer. Then, start a sg DMA transaction that copies values from sg wbuf(s) to sg
rbuf(s). After the transaction compare the buffer values; start a second sg DMA
transaction from coherent dma_src_coherent buffer to dma_dst_coherent buffer, and
after the transaction compare the buffer values */
root@imx7dsabresd:~# rmmod sdma_imx_sg_m2m.ko /* remove module */
```

DMA from User Space

Linux provides frameworks that allows user space to interface with kernel space for most types of devices (except DMA). User space DMA is defined as the ability to access buffers for DMA transfers and control DMA transfers from an user space application.

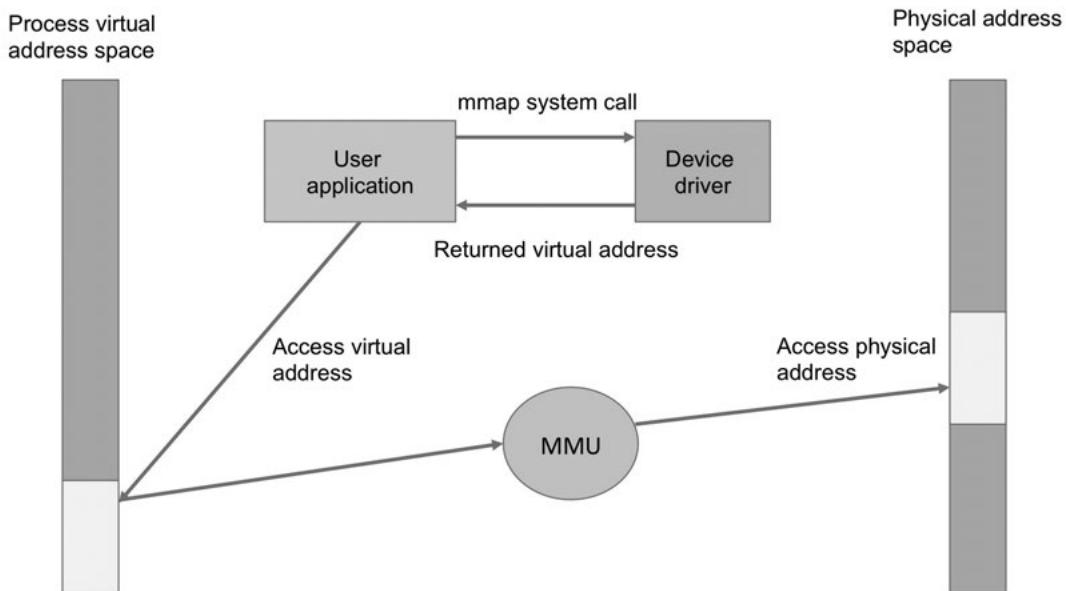
For larger buffers copying data with `copy_to_user()` and `copy_from_user()` is inefficient and in the case of DMA it defeats the purpose of using DMA to move the data. Mapping a kernel space allocated memory buffer into user space removes the need to copy data.

A process can allocate memory on its heap at runtime using `malloc()`. This mapping is taken care of by the kernel, however a process can also manipulate its memory map in a explicit way using the `mmap()` function.

The `mmap()` file operation allows memory of the device driver to be mapped into the address space of the user space process. When an user space process calls `mmap()` to map device memory into its address space, the system responds by creating a new VMA to represent that mapping. A driver that supports `mmap` (and, thus, that implements the `mmap` method) needs to help that process by completing the initialization of that VMA.

The Linux frame buffer and the Video 4 Linux, version 2 (V4L2) are two examples of drivers that use the `mmap()` function to map kernel buffers to user space.

See in the next figure how the memory is mapped from kernel to user space using `mmap()`:



You can see below the main points to implement `mmap()` in the user space:

- The call to `mmap()` requires an address and size for the memory being mapped into user space.
- The application passes zero for the address to map, since it doesn't know the address of the buffer allocated in the kernel driver.
- The size cannot be zero or `mmap()` will return an error.

```
void * mmap(  
    void *start, /* Often 0, preferred starting address */  
    size_t length, /* Length of the mapped area */  
    int prot, /* Permissions: read, write, execute */  
    int flags, /* Options: shared mapping, private copy... */  
    int fd, /* Open file descriptor */  
    off_t offset  
)
```

The `mmap()` function returns a pointer to a process virtual address. If you translate this virtual address to the physical space you can see that it matches with the physical memory space that was allocated by the kernel. This process virtual address is different from the kernel virtual address.

returned by kzalloc(), but both virtual addresses share the same physical address. Once the memory is mapped user space can read and write to it.

You can see below the main points to implement mmap() in kernel space:

- Implement a mmap() file operation and add it to the driver's file operations:

```
int (*mmap)(  
    struct file *, /* Open file structure */  
    struct vm_area_struct * /* Kernel VMA structure */  
)
```

- Initialize the mapping. This can be done in most cases with the remap_pfn_range() function, which takes care of most of the job. Only one argument has to be created, as all others come in the VMA structure. The third argument of the remap_pfn_range() is the page frame number, which is based on the physical address.

```
#include <linux/mm.h>  
  
int remap_pfn_range(  
    struct vm_area_struct *, /* VMA struct */  
    unsigned long virt_addr, /* Starting user virtual address */  
    unsigned long pfn, /* pfn of the starting physical address, use dma_map_  
    single() to get it from pointer to the virtual address of the allocated  
    buffer */  
    unsigned long size, /* Mapping size */  
    pgprot_t prot /* Page permissions */  
)
```

LAB 9.3: "DMA from user space" Module

In this lab, the sdma_m2m.c driver will be used as a starting point to develop your new DMA driver. These are the main points of the new driver:

- You will use driver's callback sdma_ioctl() function instead of sdma_write() to manage the DMA transaction.
- The sdma_mmap() callback function is added to the driver to do the mapping of the kernel buffer.
- The process virtual address will be returned to user space using the mmap() system call. Any text can be written from the user application to the returned virtual memory buffer. After that, the ioctl() system call manages the DMA transaction sending the written text from the dma_src buffer to the dma_dst buffer without any CPU intervention.

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

1. In the sdma_open() file operation obtain the dma_src DMA address using dma_map_single() that takes as a parameter the previously allocated wbuf kernel virtual address:

```
dma_priv->dma_src = dma_map_single(dma_priv->dev, dma_priv->wbuf,  
SDMA_BUF_SIZE, DMA_TO_DEVICE)
```

2. Replace the sdma_write() file operation with the sdma_ioctl() one.
3. Add a struct file_operations structure including the sdma_mmap() function:

```
struct file_operations dma_fops = {  
    .owner          = THIS_MODULE,  
    .open           = sdma_open,  
    .unlocked_ioctl = sdma_ioctl,  
    .mmap           = sdma_mmap,  
};
```

4. Create the mmap() function. The third parameter of remap_pfn_range() is the page frame number shifting the DMA physical address dma_src that you obtained using dma_map_single():

```
static int sdma_mmap(struct file *file, struct vm_area_struct *vma) {  
    struct dma_private *dma_priv;  
    dma_priv = container_of(file->private_data,  
                           struct dma_private,  
                           dma_misc_device);  
  
    if(remap_pfn_range(vma, vma->vm_start, dma_priv->dma_src >> PAGE_SHIFT,  
                      vma->vm_end - vma->vm_start, vma->vm_page_prot))  
        return -EAGAIN;  
    return 0;  
}
```

5. In the sdma.c user application, you will get the process virtual address (mapped from DMA dma_src address) using mmap() call, and will send some text to it using a char pointer. You will copy the text to the DMA dma_src buffer using the strcpy() function.

```
char *virtaddr;  
char *phrase = "Arrow web: www.arrow.com\n";  
virtaddr = (char *)mmap(0, SDMA_BUF_SIZE, PROT_READ | PROT_WRITE,  
                      MAP_SHARED, my_dev, 0);  
strcpy(virtaddr, phrase);
```

6. The ioctl() function called in user space (sdma.c) will have its corresponding sdma_ioctl() callback function in kernel space, which is in charge of executing the DMA transaction from dma_src to dma_dst.

7. Create the sdma.c application in my_apps project. Modify the application Makefile to build and deploy the sdma application.
8. You will use for this driver the same DT node of the lab 9.1 and lab 9.2.

```
sdma_m2m {  
    compatible = "arrow,sdma_m2m";  
};
```

See in the next **Listing 9-3** the "DMA from user space" driver source code (sdma_imx_mmap.c) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (sdma_sam_mmap.c) driver can be downloaded from the GitHub repository of this book. This driver has not been implemented in the BCM2837 processor.

Listing 9-3: sdma_imx_mmap.c

```
#include <linux/module.h>  
#include <linux/slab.h>  
#include <linux/uaccess.h>  
#include <linux/dma-mapping.h>  
#include <linux/fs.h>  
#include <linux/platform_data/dma-imx.h>  
#include <linux/dmaengine.h>  
#include <linux/miscdevice.h>  
#include <linux/platform_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;  
    dma_addr_t dma_src;  
    dma_addr_t dma_dst;  
};  
  
#define SDMA_BUF_SIZE (1024*63)  
  
static bool dma_m2m_filter(struct dma_chan *chan, void *param)  
{  
    if (!imx_dma_is_general_purpose(chan))  
        return false;  
    chan->private = param;
```

```
    return true;
}

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);
}

static int sdma_open(struct inode * inode, struct file * file)
{
    struct dma_private *dma_priv;
    dma_priv = container_of(file->private_data,
                           struct dma_private, dma_misc_device);

    dma_priv->wbuf = kzalloc(SDMA_BUF_SIZE, GFP_DMA);
    if(!dma_priv->wbuf) {
        dev_err(dma_priv->dev, "error allocating wbuf !!!\n");
        return -ENOMEM;
    }

    dma_priv->rbuf = kzalloc(SDMA_BUF_SIZE, GFP_DMA);
    if(!dma_priv->rbuf) {
        dev_err(dma_priv->dev, "error allocating rbuf !!!\n");
        return -ENOMEM;
    }

    dma_priv->dma_src = dma_map_single(dma_priv->dev, dma_priv->wbuf,
                                         SDMA_BUF_SIZE, DMA_TO_DEVICE);

    return 0;
}

static long sdma_ioctl(struct file *file,
                      unsigned int cmd,
                      unsigned long arg)
{
    struct dma_async_tx_descriptor *dma_m2m_desc;
    struct dma_device *dma_dev;
    struct dma_private *dma_priv;
    dma_cookie_t cookie;

    dma_priv = container_of(file->private_data,
                           struct dma_private,
                           dma_misc_device);

    dma_dev = dma_priv->dma_m2m_chan->device;
```

```
dma_priv->dma_src = dma_map_single(dma_priv->dev, dma_priv->wbuf,
                                     SDMA_BUF_SIZE, DMA_TO_DEVICE);
dma_priv->dma_dst = dma_map_single(dma_priv->dev, dma_priv->rbuf,
                                     SDMA_BUF_SIZE, DMA_TO_DEVICE);

dma_m2m_desc = dma_dev->device_prep_dma_memcpy(dma_priv->dma_m2m_chan,
                                                dma_priv->dma_dst,
                                                dma_priv->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);

dma_unmap_single(dma_priv->dev, dma_priv->dma_src,
                  SDMA_BUF_SIZE, DMA_TO_DEVICE);
dma_unmap_single(dma_priv->dev, dma_priv->dma_dst,
                  SDMA_BUF_SIZE, DMA_TO_DEVICE);

if (*(dma_priv->rbuf) != *(dma_priv->wbuf)) {
    dev_err(dma_priv->dev, "buffer copy failed!\n");
    return -EINVAL;
}

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);

kfree(dma_priv->wbuf);
kfree(dma_priv->rbuf);

return 0;
}
```

```
static int sdma_mmap(struct file *file, struct vm_area_struct *vma) {
    struct dma_private *dma_priv;

    dma_priv = container_of(file->private_data,
                           struct dma_private, dma_misc_device);

    if(remap_pfn_range(vma, vma->vm_start, dma_priv->dma_src >> PAGE_SHIFT,
                       vma->vm_end - vma->vm_start, vma->vm_page_prot))
        return -EAGAIN;

    return 0;
}

struct file_operations dma_fops = {
    .owner          = THIS_MODULE,
    .open           = sdma_open,
    .unlocked_ioctl = sdma_ioctl,
    .mmap           = sdma_mmap,
};

static int __init my_probe(struct platform_device *pdev)
{
    int retval;
    struct dma_private *dma_device;
    dma_cap_mask_t dma_m2m_mask;
    struct imx_dma_data m2m_dma_data = {0};
    struct dma_slave_config dma_m2m_config = {0};

    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_cap_zero(dma_m2m_mask);
    dma_cap_set(DMA_MEMCPY, dma_m2m_mask);
    m2m_dma_data.peripheral_type = IMX_DMATYPE_MEMORY;
    m2m_dma_data.priority = DMA_PRIO_HIGH;

    dma_device->dma_m2m_chan = dma_request_channel(dma_m2m_mask,
                                                   dma_m2m_filter,
                                                   &m2m_dma_data);

    if (!dma_device->dma_m2m_chan) {
        dev_err(&pdev->dev,
```

```
        "Error opening the SDMA memory to memory channel\n");
return -EINVAL;
}

dma_m2m_config.direction = DMA_MEM_TO_MEM;
dma_m2m_config.dst_addr_width = DMA_SLAVE_BUSWIDTH_4_BYTES;
dmaengine_slave_config(dma_device->dma_m2m_chan, &dma_m2m_config);

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 __exit 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 mmap memory to memory driver");
```

See in the next **Listing 9-4** the "sdma mmap" application source code (sdma.c) for the i.MX7D and SAMA5D2 processors. This application has not been implemented in the BCM2837 processor

Listing 9-4: sdma.c

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

#define SDMA_BUF_SIZE  (1024*63)

int main(void)
{
    char *virtaddr;
    char phrase[128];
    int my_dev = open("/dev/sdma_test", O_RDWR);
```

```
if (my_dev < 0) {
    perror("Fail to open device file: /dev/sdma_test.");
} else {
    printf("Enter phrase :\n");
    scanf("%[^\\n]*c", phrase);
    virtaddr = (char *)mmap(0, SDMA_BUF_SIZE,
                           PROT_READ | PROT_WRITE,
                           MAP_SHARED, my_dev, 0);
    strcpy(virtaddr, phrase);
    ioctl(my_dev, NULL);
    close(my_dev);
}

return 0;
}
```

sdma_imx_mmap.ko Demonstration

```
root@imx7dsabresd:~# insmod sdma_imx_mmap.ko /* load module */
root@imx7dsabresd:~# ./sdma /* map kernel DMA physical address into an user space
virtual address, write string to the user space virtual address returned, and do
ioctl() call that enables DMA transaction from dma_src to dma_dst buffer.
root@imx7dsabresd:~# rmmod sdma_imx_mmap.ko /* remove module */
```

<http://www.rejoiceblog.com/>

10

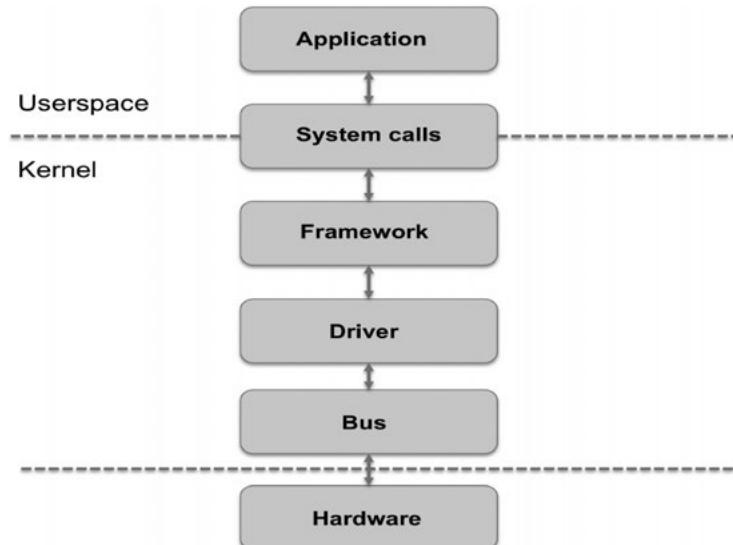
Input Subsystem Framework for Device Drivers

Many 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 framework allows 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 network framework of Linux 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 framework in detail. Several kernel modules are going to be developed helping you to understand the use of this framework to develop drivers for the same type of devices.

Observe in the following image how the driver interfaces with a framework to expose the hardware to user applications and also 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 an uniform format (struct 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 events 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 kind 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.

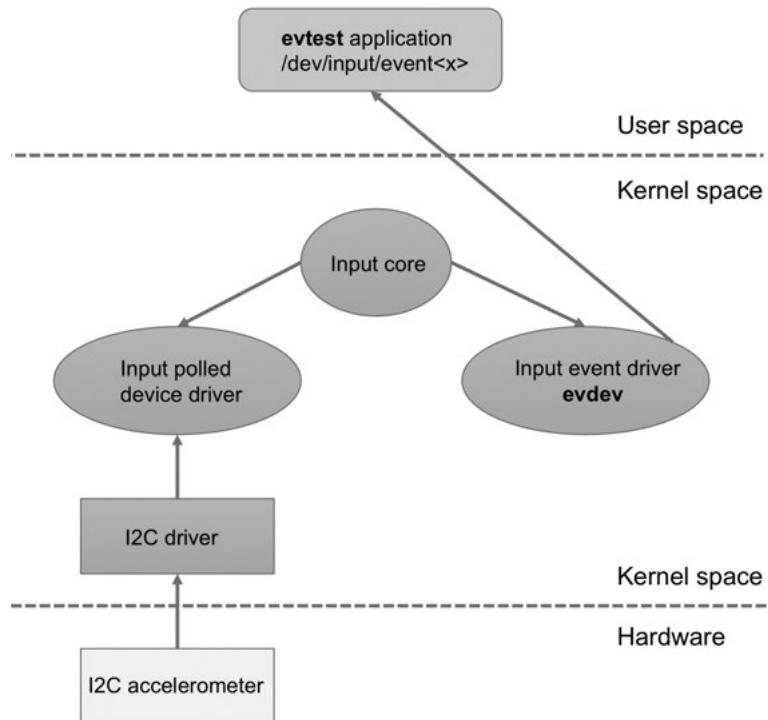
You can use blocking and nonblocking reads, and also select() on the /dev/input/eventX devices, and you'll always get a whole number of input events on a read. Their layout is:

```
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 located at <http://cgit.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. It then monitors the device and displays all the events layer events generated.

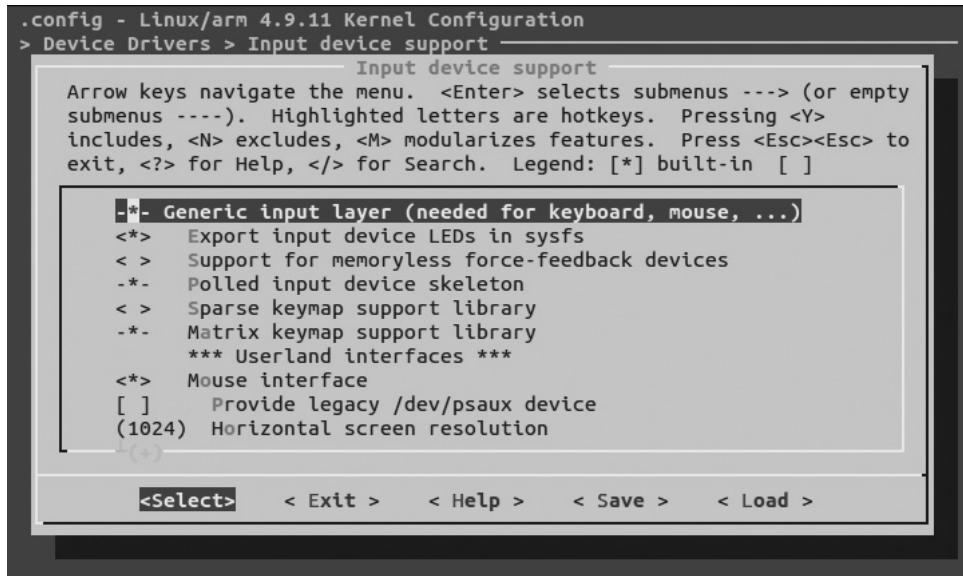
There are other event handlers like **keyboard**, **mousedev** and **joydev**.

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 using the Input subsystem.



Check that "Input device support" has been selected in the kernel configuration, also select "Polled input device skeleton":

```
~/my-linux-imx$ make menuconfig ARCH=arm
```



If it was not selected, add it to the kernel, compile the new image and load it to the target processor:

```
~/my-linux-imx$ make zImage  
~/my-linux-imx$ cp /arch/arm/boot/zImage /var/lib/tftpboot
```

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 processor. You can use the ADXL345 Accel click mikroBUS™ accessory board to develop the driver; you will access to the schematic of the board at <http://www.mikroe.com/click/accel/>.

Your driver will scan periodically the value of one of the accelerometer axes, and depending of 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 struct input_polled_dev defined in include/linux/input-poldev.h:

```
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 using the next 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 to 1 depending of 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 struct 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 by the driver to the event handler using the input_event() function.

After submitting the event, the input core must be notified using the function input_sync():

```
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 implement for

this driver the same hardware description of the lab 6.2 connecting the SDA and SCL pins of the processor to the SDA and SCL pins of the ADXL345 Accel click mikroBUS™ accessory board.

Device Tree

Modify the device tree files under arch/arm/boot/dts/ folder to include your DT driver's 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.

For the **MCIMX7D-SABRE** Board open the DT file imx7d-sdb.dts and add the adxl345@1c node inside the i2c3 controller master node. The reg property provides the ADXL345 I2C address:

```
&i2c3 {  
    clock-frequency = <100000>;  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_i2c3>;  
    status = "okay";  
  
    adxl345@1c {  
        compatible = "arrow,adxl345";  
        reg = <0x1d>;  
    };  
  
    sii902x: sii902x@39 {  
        compatible = "SII,sii902x";  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_sii902x>;  
        interrupt-parent = <&gpio2>;  
        interrupts = <13 IRQ_TYPE_EDGE_FALLING>;  
        mode_str ="1280x720M@60";  
        bits-per-pixel = <16>;  
        reg = <0x39>;  
        status = "okay";  
    };  
};  
[...]  
};
```

For the **SAMA5D2B-XULT** Board open the DT file at91-sama5d2_xplained_common.dtsi and add the adxl345@1bc sub-node inside the i2c1 controller master node. The reg property provides the ADXL345 I2C address:

```
i2c1: i2c@fc028000 {  
    dmas = <0>, <0>;  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_i2c1_default>;
```

```
status = "okay";  
[...]  
adxl345@1c {  
    compatible = "arrow,adxl345";  
    reg = <0x1d>;  
};  
[...]  
at24@54 {  
    compatible = "atmel,24c02";  
    reg = <0x54>;  
    pagesize = <16>;  
};  
};
```

For the **Raspberry Pi 3 Model B** Board open the DT file `bcm2710-rpi-3-b.dts` and add the `adxl345@1c` node below 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 your target processor.

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 a struct `i2c_driver` structure:

```
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,  
};
```

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 struct 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 next 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-polled.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 another labs throughout this book, this private structure allows the driver to manage multiple devices using the same driver. Add the next 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:

```
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 struct i2c_client using the i2c_set_clientdata() function. This function stores ioaccel in client->dev->driver_data. You can retrieve the ioccel 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 struct input_polled_dev structure in probe() using the next line of code:

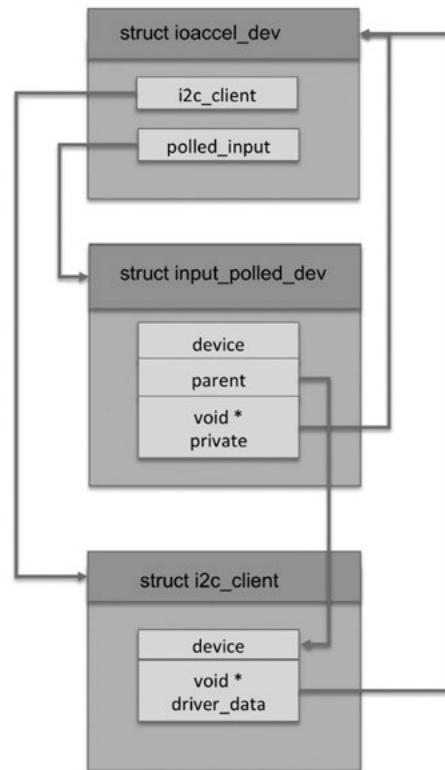
```
ioaccel->polled_input = devm_input_allocate_polled_device(&client->dev);
```

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 from */
```

```
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->dev.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 devices 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, the OUT_X_MSB register (address 0x33) of the ADXL345 accelerometer will be read using the i2c_smbus_read_byte_data() function. The first parameter of the i2c_smbus_read_byte_data() function is a pointer to the struct 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 struct i2c_client structure, then this I2C address is sent to the ioaccel_probe() function via a client pointer variable that points to this struct i2c_client structure.

An input event KEY_1 will be reported with values of 0 or 1 depending of 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);
}
```

See in the next **Listing 10-1** the "input subsystem accelerometer" driver source code (i2c_imx_accel.c) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (i2c_sam_accel.c) and BCM2837 (i2c_rpi_accel.c) drivers can be downloaded from the GitHub repository of this book.

Listing 10-1: i2c_imx_accel.c

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/i2c.h>
#include <linux/input-polldrv.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 the 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,adxl345", },
    { }
};
MODULE_DEVICE_TABLE(of, ioaccel_dt_ids);

static const struct i2c_device_id i2c_ids[] = {
    { "adxl345", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, i2c_ids);

/* create struct i2c_driver */
```

```
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,
};

/* register to i2c bus as a driver */
module_i2c_driver(ioaccel_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberalt@arroweurope.com>");
MODULE_DESCRIPTION("This is an accelerometer INPUT framework platform driver");
```

i2c_imx_accel.ko Demonstration

"Use i2c-utils suite to interact with sensors (before loading the module)"

"i2cdetect is a tool of the i2c-tools suite. It is able to probe an i2c bus from user space and report the addresses in use"

```
root@imx7dsabresd:~# i2cdetect -l /* list available buses, accelerometer is in bus 2 */
*/
```

i2c-3	i2c	30a50000.i2c	I2C adapter
i2c-1	i2c	30a30000.i2c	I2C adapter
i2c-2	i2c	30a40000.i2c	I2C adapter
i2c-0	i2c	30a20000.i2c	I2C adapter

```
root@imx7dsabresd:~# i2cdetect -y 2 /* see detected devices, see the "1d" accelerometer address */
```

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10:	-	-	-	-	-	-	-	-	-	-	-	1d	-	-	-	-
20:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
30:	-	-	-	-	-	-	-	-	-	UU	-	-	-	-	-	-
40:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
50:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
60:	60	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
70:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

```
root@imx7dsabresd:~# i2cset -y 2 0x1d 0x2d 0x08 /* enter measurement mode */
root@imx7dsabresd:~# while true; do i2cget -y 2 0x1d 0x33; done /* you can see the
OUT_X_MSB register values. The 0x33 value correspond to the OUT_X_MSB register
address. You can move the i.MX7D board and see the OUT_X_MSB register values
changing. Set different range of values to generate the event. This range of values
will be set inside the ioaccel_poll() function */

"Load now the i2_imx_accel.ko module"

root@imx7dsabresd:~# insmod i2c_imx_accel.ko
adxl345 2-001d: my_probe() function is called.
input: IOACCEL keyboard as /devices/soc0/soc/30800000.aips-bus/30a40000.i2c/i2c-
2/2-001d/input/input5

"Launch evtest application and see the input available devices. Select 4. After
launching the module and executing the evtest application move the i.MX7D board
until you see the event KEY_1 being generated"

root@imx7dsabresd:~# evtest
No device specified, trying to scan all of /dev/input/event*

Available devices:

/dev/input/event0: fxos8700
/dev/input/event1: fxa2100x
/dev/input/event2: 30370000.snvs:snvs-powerkey
/dev/input/event3: mpl3115
/dev/input/event4: IOACCEL keyboard

Select the device event number [0-4]: 4

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:

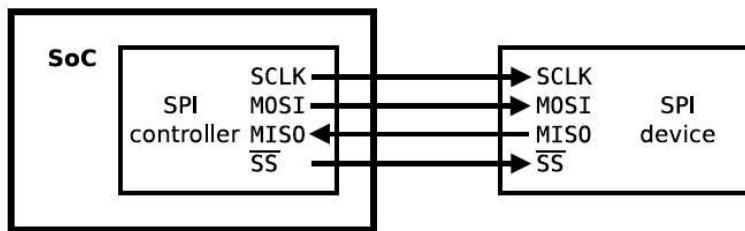
Testing ... (interrupt to exit)

Event: time 1510654662.383415, type 1 (EV_KEY), code 2 (KEY_1), value 1
Event: time 1510654662.383415, ----- SYN_REPORT -----
```

```
Event: time 1510654662.443578, type 1 (EV_KEY), code 2 (KEY_1), value 0
Event: time 1510654662.443578, ----- SYN_REPORT -----
Event: time 1510654669.763539, type 1 (EV_KEY), code 2 (KEY_1), value 1
Event: time 1510654669.763539, ----- SYN_REPORT -----
Event: time 1510654669.823578, type 1 (EV_KEY), code 2 (KEY_1), value 0
Event: time 1510654669.823578, ----- SYN_REPORT -----
Event: time 1510654679.063539, type 1 (EV_KEY), code 2 (KEY_1), value 1
Event: time 1510654679.063539, ----- SYN_REPORT -----
root@imx7dsabresd:~# rmmod i2c_imx_accel.ko /* remove the module */
```

Using SPI with Linux

The Serial Peripheral Interface (SPI) is a synchronous four wire serial link used to connect microprocessors to sensors, memory, 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

clock and chip selects, shift data bits on/off wire and configure basic SPI characteristics like clock frequency and mode; you can see for example the Linux driver for Broadcom BCM2835 auxiliary SPI controllers (drivers/spi/spi-bcm2835aux.c). The **protocol drivers** support the SPI slave specific functionality, are based on messages and transfer and rely on 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 struct 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 block until the call succeeds. When using asynchronous request, you are periodically checking if the transaction is finished. The driver for a SPI controller manages access to those devices through a queue of struct spi_message transactions, copying data between CPU memory and a 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 of the at25.c driver located under drivers/misc/eeprom/ as an example of a 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 using a callback provided with the message. After any detected error, the chip is deselected and processing of that struct 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 (and small, and "optional") 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. It's designed to support common

RPC-style requests, such as writing an eight bit command and reading a sixteen bit response -- `spi_w8r16()` being one its wrappers, doing exactly that.

```
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;
}
```

See the `m41t93_set_reg()` function of the `rtc-m41t93.c` driver located under `drivers/rtc/` as an example of a wrapper SPI transaction:

```
static inline int m41t93_set_reg(struct spi_device *spi, u8 addr, u8 data)
{
    u8 buf[2];

    /* MSB must be '1' to write */
    buf[0] = addr | 0x80;
    buf[1] = data;

    return spi_write(spi, buf, sizeof(buf));
}
```

The Linux SPI Subsystem

The Linux **SPI subsystem** is based in 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. The SPI core in the device model is a collection of code that provides interface support between an individual client driver and some SPI bus masters such as the i.MX7D SPI controllers. The SPI bus core is registered itself with the kernel using the `bus_register()` function and also declares the SPI struct `bus_type` structure:

```
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) used for a **SPI client device driver** to manage SPI transactions with a device connected to a SPI bus.

2. The **SPI controller drivers** are located under drivers/spi/ directory. The SPI controller is a platform device that must be registered as a device to the platform bus via the of_platform_populate() function and registered itself as a driver 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 is a set of custom functions that issues read/writes to the specific SPI controller hardware I/O addresses. There is specific code for each different processor's SPI master driver. The main task of this SPI driver is to provide a struct spi_master per each probed SPI controller. The spi_alloc_master() function is called to allocate the master, and spi_master_get_devdata() is called to get the driver-private data allocated for that device.

```
struct spi_master *master;
master = spi_alloc_master(dev, sizeof *c);
c = spi_master_get_devdata(master);
```

The driver will initialize the fields of struct spi_master with the methods used to interact with the SPI core and SPI protocol drivers. After that, struct spi_master is initialized and devm_spi_register_master() registers each SPI controller to the SPI bus core publishing it to the rest of the system. At that time, device nodes for the controller and any predeclared SPI devices will be made available, and the driver model core will take care of binding them to drivers. These are some of the SPI master methods:

- master->setup(struct spi_device *spi): this 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): the subsystem calls the driver to transfer a single transfer while queuing transfers that arrive in the meantime. When the driver has finished this transfer, it must call spi_finalize_current_transfer() so that the subsystem can issue the next transfer. This may sleep.

See below the initialization and registration of a Broadcom bcm2835 SPI master controller inside the probe() function of the spi-bcm2835.c driver located under drivers/spi/ folder:

```
static int bcm2835_spi_probe(struct platform_device *pdev)
{
    struct spi_master *master;
    struct bcm2835_spi *bs;
    struct resource *res;
    int err;

    master = spi_alloc_master(&pdev->dev, sizeof(*bs));

    platform_set_drvdata(pdev, master);

    master->mode_bits = BCM2835_SPI_MODE_BITS;
    master->bits_per_word_mask = SPI_BPW_MASK(8);
    master->num_chipselect = 3;
    master->setup = bcm2835_spi_setup;
    master->set_cs = bcm2835_spi_set_cs;
    master->transfer_one = bcm2835_spi_transfer_one;
    master->handle_err = bcm2835_spi_handle_err;
    master->prepare_message = bcm2835_spi_prepare_message;
    master->dev.of_node = pdev->dev.of_node;

    bs = spi_master_get_drvdata(master);

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    bs->regs = devm_ioremap_resource(&pdev->dev, res);

    bs->clk = devm_clk_get(&pdev->dev, NULL);

    bs->irq = platform_get_irq(pdev, 0);

    clk_prepare_enable(bs->clk);

    bcm2835_dma_init(master, &pdev->dev);

    /* initialise the hardware with the default polarities */
    bcm2835_wr(bs, BCM2835_SPI_CS,
               BCM2835_SPI_CS_CLEAR_RX | BCM2835_SPI_CS_CLEAR_TX);

    devm_request_irq(&pdev->dev, bs->irq, bcm2835_spi_interrupt, 0,
                     dev_name(&pdev->dev), master);

    devm_spi_register_master(&pdev->dev, master);

    return 0;
}
```

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

3. The **SPI device drivers** are located throughout `linux/drivers/`, depending on the type of device (for example, `drivers/input/` for input devices). The driver code is specific to the device (for example, an accelerometer, a digital analog converter, etc.) and uses the SPI core API to communicate with the SPI master driver and receive data to/from the SPI device.

For example, if the SPI client driver calls `spi_write_then_read()` declared in `drivers/spi/spi.c`, then this function calls `spi_sync()`, which in turn calls `__spi_sync()`. The `__spi_sync()` function calls `__spi_pump_messages()`, which processes 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 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 handling at the top of the function should deal with this safely. Finally `__spi_pump_messages()` calls `master->transfer_one_message()` that in the BCM2835 SPI master driver was initialized to `bcm2835_spi_transfer_one()` (master driver function that interacts with the specific SPI controller hardware).

Writing SPI Client Drivers

You will focus now in the writing of SPI client drivers. In this and in successive chapters, you will develop several SPI client drivers to control an accelerometer and an analog to digital converter device. In the next sections, it will be covered the main steps to set up a SPI client driver.

SPI Client Driver Registration

The SPI subsystem defines a `struct spi_driver` structure that is inherited from `struct device_driver`, and which must be instantiated and registered to the **SPI bus core** by each **SPI 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 the example below of a `struct spi_driver` definition for a 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 device driver you create an array of structures `struct of_device_id` where you specify compatible strings that hold the same value of the DT device node's compatible property. The `struct of_device_id` located in `include/linux/mod_devicetable.h` is defined as:

```
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 `struct spi_driver` is a pointer to the array of structures `struct of_device_id` that stores 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 field `compatible` in one of the `of_device_id` entries matches with the `compatible` property of a DT device node. The `probe()` function is responsible of initializing the device with the configuration values obtained from the matching DT device node and also to register the device to the appropriate kernel framework.

In your SPI device driver you also define an array of `struct 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 Device Tree

In the device tree, each SPI controller device is typically declared in the .dtsi file that describes the processor (for i.MX7D see arch/arm/boot/dts/imx7s.dtsi). The SPI DT controller is normally declared with status = "disabled". In the imx7s.dtsi file there are declared four DT SPI controllers devices, that are registered to the SPI bus core through the of_platform_populate() function. For the i.MX7D, the spi-imx.c driver located under drivers/spi will register itself to the SPI bus core using the module_spi_driver() function. The probe() function inside spi-imx.c will be called four times (one for each compatible = "fsl,imx51-ecspi" matching) initializing a struct spi_master for each controller and registering it to the SPI bus core using the devm_spi_register_master() function. See below the declaration of three of the four i.MX7D DT SPI controller nodes:

```
ecspi1: ecspi@30820000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx7d-ecspi", "fsl,imx51-ecspi";
    reg = <0x30820000 0x10000>;
    interrupts = <GIC_SPI 31 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clks IMX7D_ECSP1_ROOT_CLK>,
              <&clks IMX7D_ECSP1_ROOT_CLK>;
    clock-names = "ipg", "per";
    status = "disabled";
};

ecspi2: ecspi@30830000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx7d-ecspi", "fsl,imx51-ecspi";
    reg = <0x30830000 0x10000>;
    interrupts = <GIC_SPI 32 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clks IMX7D_ECSP2_ROOT_CLK>,
              <&clks IMX7D_ECSP2_ROOT_CLK>;
    clock-names = "ipg", "per";
    status = "disabled";
};

ecspi3: ecspi@30840000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx7d-ecspi", "fsl,imx51-ecspi";
    reg = <0x30840000 0x10000>;
    interrupts = <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clks IMX7D_ECSP3_ROOT_CLK>,
              <&clks IMX7D_ECSP3_ROOT_CLK>;
    clock-names = "ipg", "per";
    status = "disabled";
};
```

These are the required properties for an i.MX7D DT SPI controller node:

- compatible: "fsl,imx51-ecspi" for SPI compatible with the one integrated on i.MX7D.
- reg: offset and length of the register set for the device.
- interrupts: should contain eCSPI interrupt.
- clocks: clock specifiers for both ipg and per clocks.
- clock-names: clock names should include both "ipg" and "per". See the clock consumer binding, in Documentation/devicetree/bindings/clock/clock-bindings.txt
- dmas: DMA specifiers for tx and rx dma. See the DMA client binding, in Documentation/devicetree/bindings/dma/dma.txt
- dma-names: DMA request names should include "tx" and "rx" if present.

And these are the optional properties:

- cs-gpios: Specifies the gpio pins to be used for chip selects.
- num-cs: Total number of chip selects.

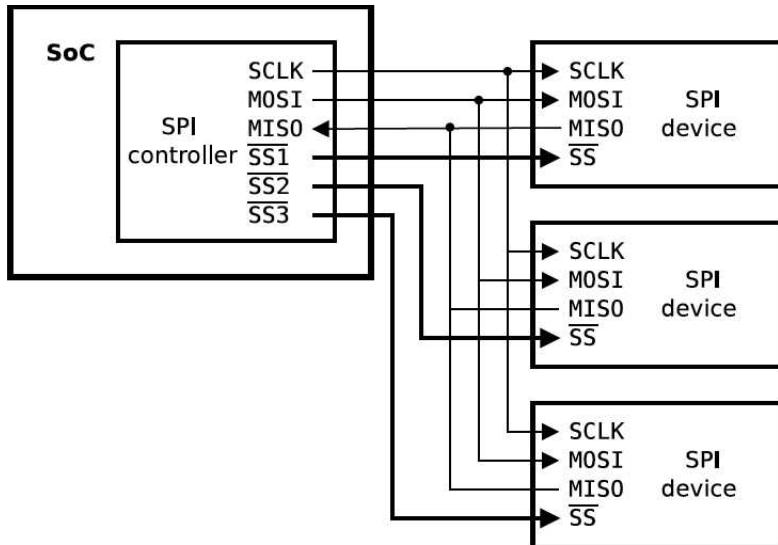
If cs-gpios is used the number of chip selects will be increased automatically with $\max(\text{cs-gpios} > \text{hw cs})$. So, if for example the controller has 2 CS lines, and the cs-gpios property looks like this:

```
cs-gpios = <&gpio1 0 0>, <0>, <&gpio1 1 0>, <&gpio1 2 0>;
```

Then it should be configured so that num_chipselect = 4 with the following mapping:

```
cs0 : &gpio1 0 0
cs1 : native
cs2 : &gpio1 1 0
cs3 : &gpio1 2 0
```

See in the following figure a SPI controller with multiple chip selects connected to several SPI devices:



The device tree declaration of SPI devices is done as sub-nodes of the SPI master controller at the board/platform level (arch/arm/boot/dts/imx7d-sdb.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.

Find in the DT imx7d-sdb.dts file the ecspi3 controller declaration. The ecspi3 controller is enabled writing "okay" to the status property. In the cs-gpios property you can see that there are two chip selects enabled, the first one will enable the tsc2046 device and the second one will enable the ADXL345 accelerometer. The pinctrl-0 property inside the ecspi3 node points to the pinctrl_ecspi3 and pinctrl_ecspi3_cs function nodes, where the chip selects pads are being multiplexed with GPIO functionality.

In the ADXL345 sub-node you can see that reg value is equal to 1, whereas in the tsc2046 sub-node the reg value is 0 to select the different chip selects. Inside each sub-node you can see some required properties like the spi-max-frequency and the compatible one.

```
&ecspi3 {
    fsl,spi-num-chipselects = <2>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ecspi3 &pinctrl_ecspi3_cs>;
    cs-gpios = <&gpio5 9 GPIO_ACTIVE_HIGH>, <&gpio6 22 0>;
    status = "okay";

    tsc2046@0 {
        compatible = "ti,tsc2046";
        reg = <0>;
        spi-max-frequency = <1000000>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_tsc2046_pendown>;
        interrupt-parent = <&gpio2>;
        interrupts = <29 0>;
        pendown-gpio = <&gpio2 29 GPIO_ACTIVE_HIGH>;
        ti,x-min = /bits/ 16 <0>;
        ti,x-max = /bits/ 16 <0>;
        ti,y-min = /bits/ 16 <0>;
        ti,y-max = /bits/ 16 <0>;
        ti,pressure-max = /bits/ 16 <0>;
        ti,x-plate-ohms = /bits/ 16 <400>;
        wakeup-source;
    };

    ADXL345@1 {
        compatible = "arrow,adxl345";
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_accel_gpio>;
        spi-max-frequency = <5000000>;
        spi-cpol;
        spi-cpha;
        reg = <1>;
        int-gpios = <&gpio6 14 GPIO_ACTIVE_LOW>;
        interrupt-parent = <&gpio6>;
        interrupts = <14 IRQ_TYPE_LEVEL_HIGH>;
    };
};
```

LAB 10.2: "SPI accel input device" Module

Throughout the upcoming lab, you will implement your first driver for a SPI device. The driver will manage an accelerometer device connected to the SPI bus. You can use the same ADXL345 Accel click mikroBUS™ accessory board from the previous lab.

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 Hennerich driver at <https://wiki.analog.com/resources/tools-software/linux-drivers/input-misc/adxl345>.

The driver will support single tap motion detection 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 for 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 pag 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 10.2 Hardware Description for the i.MX7D Processor

In this lab, you will use the SPI pins of the MCIMX7D-SABRE board mikroBUS™ to connect to the ADXL345 Accel click mikroBUS™ accessory board. The Accel Click™ board communicates with the main board via I2C or SPI interface depending on the position of the J1, J2 and J3 SMD jumpers. These jumpers are soldered in I2C interface position by default. You should solder the jumpers in SPI interface position for this lab.

Go to the pag.20 of the MCIMX7D-SABRE schematic to see the MikroBUS connector and look for the SPI pins. Connect these pins to the SPI ones of the ADXL345 Accel click mikroBUS™ accessory board. The ADXL345 will generate an interrupt, so connect INT pin between both mikroBUS™ boards. Connect also VCC 3.3V and GND between both boards. See below a description of the connections:

- Connect i.MX7D **MKBUS_ESPI3_SS0_B** (CS) to ADXL345 **CS** (CS)
- Connect i.MX7D **MKBUS_ESPI3_SCLK** (SCK) to ADXL345 **SCL** (SCK)
- Connect i.MX7D **MKBUS_ESPI3_MISO** (MISO) to ADXL345 **SDO** (MISO)
- Connect i.MX7D **MKBUS_ESPI3_MOSI** (MOSI) ADXL345 **SDI** (MOSI)
- Connect i.MX7D **MKBUS_INT** (INT) to ADXL345 **INT1** (INT)

LAB 10.2 Hardware Description for the SAMA5D2 Processor

For the SAMA5D2 processor, open the SAMA5D2B-XULT board schematic and look for connectors on board with pins that provide SPI signals. The SAMA5D2B-XULT board have five 8-pin, one 6-pin, one 10-pin and one 36-pin headers (J7, J8, J9, J16, J17, J20, J21, J22) that enable the PIO connection of various expansion cards. These headers' physical and electrical implementation match the Arduino R3 extension ("shields") system.

You can access to the SPI signals using the J17 header. See below a description of the connections between both boards:

- Connect SAMA5D2 **ISC_PCK/SPI1_NPCS0_PC4** (pin 26 of J17) to ADXL345 **CS** (CS)
- Connect SAMA5D2 **SC_D7/SPI1_SPCK_PC1** (pin 17 of J17) to ADXL345 **SCL** (SCK)
- Connect SAMA5D2 **ISC_D9/SPI1_MISO_PC3** (pin 22 of J17) to ADXL345 **SDO** (MISO)
- Connect SAMA5D2 **ISC_D8/SPI1_MOSI_PC2** (pin 23 of J17) to ADXL345 **SDI** (MOSI)
- Connect SAMA5D2 **ISC_D11/EXP_PB25** (pin 30 of J17) to ADXL345 **INT1** (INT)

LAB 10.2 Hardware Description for the BCM2837 Processor

For the BCM2837 processor, you will use the GPIO expansion connector to obtain the SPI signals. Go to the Raspberry-Pi-3B-V1.2-Schematics to see the J8 connector. See below a description of the connections between both boards:

- Connect BCM2837 **SPI_CE0_N** (pin 24 of J8) to ADXL345 **CS** (CS)
- Connect BCM2837 **SPI_SCLK** (pin 23 of J8) to ADXL345 **SCL** (SCK)
- Connect BCM2837 **SPI_MISO** (pin 21 of J8) to ADXL345 **SDO** (MISO)
- Connect BCM2837 **SPI_MOSI** (pin 19 of J8) to ADXL345 **SDI** (MOSI)
- Connect BCM2837 **GPIO23** (pin 16 of J8) to ADXL345 **INT1** (INT)

LAB 10.2 Device Tree for the i.MX7D Processor

Modify the device tree file `imx7d-sdb.dts` adding the `adxl345@1` sub-node inside the `ecspi3` controller master node. The `pinctrl-0` property of the `adxl345` node points to the `pinctrl_accel_gpio` pin configuration node, where the `SAI1_TX_SYNC` pad is multiplexed as a GPIO signal. The `int-gpios` property will make the GPIO pin 14 of the GPIO6 port available to the driver so that you can set the pin direction to input and get the Linux IRQ number associated to this pin. The `reg` property provides the CS number; there are two chip selects inside the `ecspi3` node, one for the `tsc2046` node

and the other one for the adxl345 node. Don't forget to set the status property to "okay", as it was "disabled" in previous labs. These are other DT properties that you can see inside the adxl345 node:

- spi-max-frequency: maximum SPI clock frequency for this device
- spi-cpha: needs to be set for the correct SPI mode
- spi-cpol: needs to be set for the correct SPI mode
- interrupt-parent: specifies which IRQ controller is used
- interrupts: the interrupt associated with the INT pin

```
&ecspi3 {  
    fsl,spi-num-chipselects = <1>;  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_ecspi3 &pinctrl_ecspi3_cs>;  
    cs-gpios = <&gpio5 9 GPIO_ACTIVE_HIGH>, <&gpio6 22 0>;  
    status = "okay";  
  
    tsc2046@0 {  
        compatible = "ti,tsc2046";  
        reg = <0>;  
        spi-max-frequency = <1000000>;  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_tsc2046_pendown>;  
        interrupt-parent = <&gpio2>;  
        interrupts = <29 0>;  
        pendown-gpio = <&gpio2 29 GPIO_ACTIVE_HIGH>;  
        ti,x-min = /bits/ 16 <0>;  
        ti,x-max = /bits/ 16 <0>;  
        ti,y-min = /bits/ 16 <0>;  
        ti,y-max = /bits/ 16 <0>;  
        ti,pressure-max = /bits/ 16 <0>;  
        ti,x-plate-ohms = /bits/ 16 <400>;  
        wakeup-source;  
    };  
  
    Accel: ADXL345@1 {  
        compatible = "arrow,adxl345";  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_accel_gpio>;  
        spi-max-frequency = <5000000>;  
        spi-cpol;  
        spi-cpha;  
        reg = <1>;  
        int-gpios = <&gpio6 14 GPIO_ACTIVE_LOW>;  
        interrupt-parent = <&gpio6>;  
        interrupts = <14 IRQ_TYPE_LEVEL_HIGH>;  
    };  
};
```

See below the pinctrl_accel_gpio pin configuration node located inside the iomuxc node, where the SAI1_TX_SYNC pad is multiplexed as a GPIO signal:

```
pinctrl_accel_gpio: pinctrl_accel_gpiogrp {  
    fsl,pins = <  
        MX7D_PAD_SAI1_TX_SYNC__GPIO6_IO14    0x2  
    >;  
};
```

LAB 10.2 Device Tree for the SAMA5D2 Processor

Open the device tree file at91-sama5d2_xplained_common.dtsi and create the spi1 controller node. The pinctrl-0 property of the spi node points to the pinctrl_spi1_default pin configuration node, where the spi controller pads are multiplexed as SPI signals. The spi controller is enabled setting the status property to "okay". Add now the adxl345@0 sub-node inside the spi1 controller master node. The pinctrl-0 property of the adxl345 node points to the pinctrl_accel_gpio_default pin configuration node, where the PB25 pad is multiplexed as a GPIO signal. The int-gpios property will make the GPIO pin 25 of the PIOB port available to the driver so that you can set the pin direction to input and get the Linux IRQ number associated to this pin. The reg property provides the CS number; there is only one CS inside the spi1 node.

```
spi1: spi@fc000000 {  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_spi1_default>;  
    status = "okay";  
  
    Accel: ADXL345@0 {  
        compatible = "arrow,adxl345";  
        reg = <0>;  
        spi-max-frequency = <5000000>;  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_accel_gpio_default>;  
        spi-cpol;  
        spi-cpha;  
        int-gpios = <&pioA 57 GPIO_ACTIVE_LOW>;  
        interrupt-parent = <&pioA>;  
        interrupts = <57 IRQ_TYPE_LEVEL_HIGH>;  
    };  
};
```

See below the pinctrl_spi_default pin configuration node, where the PC1, PC2, PC3, and PC4 pads are multiplexed as SPI signals:

```
pinctrl_spi1_default: spi1_default {
    pinmux = <PIN_PC1_SPI1_SPCK>,
    <PIN_PC2_SPI1_MOSI>,
    <PIN_PC3_SPI1_MISO>,
    <PIN_PC4_SPI1_NPCS0>;
    bias-disable;
};
```

See below the pinctrl_accel_gpio_default pin configuration node, where the PB25 pad is multiplexed as a GPIO signal:

```
pinctrl_accel_gpio_default: accel_gpio_default {
    pinmux = <PIN_PB25_GPIO>;
    bias-disable;
};
```

LAB 10.2 Device Tree for the BCM2837 Processor

Open and modify the device tree file bcm2710-rpi-3-b.dts adding the adxl345@0 sub-node inside 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 to 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>;  
    };  
};
```

See below the accel_int_pin pin configuration node, where the GPIO23 pad is multiplexed as a GPIO signal:

```
accel_int_pin: accel_int_pin {  
    brcm,pins = <23>;  
    brcm,function = <0>; /* Input */  
    brcm,pull = <0>; /* none */  
};
```

LAB 10.2 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.h>
```

```
#include <linux/of_gpio.h>
#include <linux/spi/spi.h>
#include <linux/interrupt.h>
```

2. Define the masks and the macros used to generate the specific command byte of the SPI transaction (spi_read(), spi_write(), spi_write_then_read()):

```
#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 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 */
```

4. Create the rest of #defines to perform operations in the ADXL345 registers 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 struct adxl345_bus_ops 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
 * event sent when a single tap interrupt is triggered
 */
__set_bit(EV_KEY, input_dev->evbit);
__set_bit(pdev->ev_code_tap[ADXL_X_AXIS], input_dev->keybit);
__set_bit(pdev->ev_code_tap[ADXL_Y_AXIS], input_dev->keybit);
__set_bit(pdev->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 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, 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. A threaded interrupt will be added to the driver 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 interrupt handler, which is often needed for I2C/SPI devices, as the interrupt handler needs to communicate with them. In this driver, you are going to communicate via SPI with the ADXL345 device inside the interrupt 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 adxl345_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. The events type EV_KEY will be generated with 3 different event codes that will be set depending of the axis where the tap motion detection has been selected. These events will be sent inside the ISR by calling the adxl345_do_tap() function.

```
/*
 * Set EV_KEY type event with 3 events code support
 * event 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 to the driver from user space. You can set and read the sample rate, read the 3 axis values, and show the last stored axis values using sysfs hooks.

The sysfs attributes will be created 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 as follows into a group:

```
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 adxl345_position_read() code where the tree axis values will be read:

```
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, DATAX0, DATAZ1 - DATAZ0 + 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` is a member of the struct `adxl345_bus_ops` initialized to the `adxl345_spi_read_block` SPI 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 are going to be read starting from the `reg` one. The `buf` parameter is a pointer to the buffer where the axis values are going to 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;

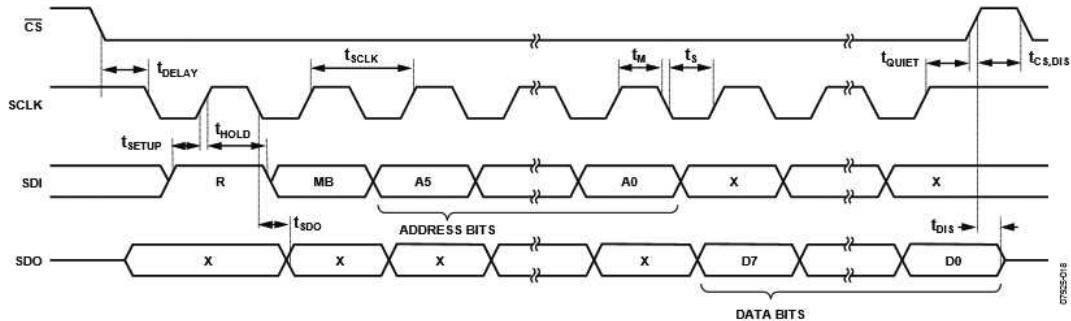
    /* Add MB flags to the reading */
    reg = ADXL345_READMB_CMD(reg);

    /* 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()` that sends to the SPI bus a command byte composed of 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 in the next figure a SPI 4-Wire Read diagram:



See below the macros for the SPI commands used to read 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 struct spi_device_id structures:

```
static const struct spi_device_id adxl345_id[] = {
    { .name = "adxl345", },
    { }
};
```

MODULE_DEVICE_TABLE(spi, adxl345_id);

13. Add a struct 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. Build the modified device tree, and load it to your target processor.

See in the next **Listing 10-2** the "SPI accel input device" driver source code (adxl345_imx.c) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (adxl345_sam.c) and BCM2837 (adxl345_rpi.c) drivers can be downloaded from the GitHub repository of this book.

Listing 10-2: adxl345_imx.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 DATA_X0           0x32 /* R X-Axis Data 0 */
#define DATA_X1           0x33 /* R X-Axis Data 1 */
#define DATA_Y0           0x34 /* R Y-Axis Data 0 */
#define DATA_Y1           0x35 /* R Y-Axis Data 1 */
#define DATA_Z0           0x36 /* R Z-Axis Data 0 */
#define DATA_Z1           0x37 /* R Z-Axis Data 1 */
#define FIFO_CTL          0x38 /* R/W FIFO control */

/* DEVIDs */
#define ID_AXL345 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, DATAZ0, DATAZ1 - DATAZ0 + 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 check
 * the ACT_TAP_STATUS (0x2B) TAP_X, TAP_Y, TAP_Z bits starting
 * for 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 adxl345_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 I set ac->pdata.low_power_mode = 1
 * then is 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 *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,
};

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;
    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 = &adxl345_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
 * event 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 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); // PTR_ERR return an int from a pointer
    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 the axis reading power
 * mode, 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;

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);
}

/*
 * Write the address of the register
 * and read the value of it
 */
static int adx1345_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_imx.ko Demonstration

```
root@imx7dsabresd:~# insmod adxl345_imx.ko /* load module */
adxl345_imx: loading out-of-tree module taints kernel.
adxl345 spi2.1: DEVID: 229
adxl345 spi2.1: ADXL345 is found
adxl345 spi2.1: The IRQ number is: 256
input: ADXL345 accelerometer as /devices/soc0/soc/30800000.aips-bus/30840000.ecs
pi/spi_master/spi2/spi2.1/input/input6

root@imx7dsabresd:~# cd /sys/class/input/input6/device/
root@imx7dsabresd:/sys/class/input/input6/device# ls /* see the sysfs entries */
root@imx7dsabresd:/sys/class/input/input6/device# cat read /* read the three axes
values */
(-1, 3, 241)

root@imx7dsabresd:/sys/class/input/input6/device# cat read /* move the accel board
and read again */
(-5, 250, -25)

root@imx7dsabresd:/sys/class/input/input6/device# cat rate /* read the data rate */
root@imx7dsabresd:/sys/class/input/input6/device# echo 10 > rate /* change the data
rate */
root@imx7dsabresd:~# evtest /* launch the evtest application. Move the accelerometer
board in the z axis direction and see the interrupts and events generated */
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0:      fxos8700
/dev/input/event1:      fxa2100x
/dev/input/event2:      30370000.snv:s:snvs-powerkey
/dev/input/event3:      ADS7846 Touchscreen
/dev/input/event4:      mpl3115
/dev/input/event5:      ADXL345 accelerometer
Select the device event number [0-5]: 5
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 spi2.1: single tap interrupt has occurred
Event: time 1510654071.237172, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 1
Event: time 1510654071.237172, ----- SYN_REPORT -----
Event: time 1510654071.237185, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 0
Event: time 1510654071.237185, ----- SYN_REPORT -----
adxl345 spi2.1: single tap interrupt has occurred
Event: time 1510654073.316372, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 1
Event: time 1510654073.316372, ----- SYN_REPORT -----
Event: time 1510654073.316385, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 0
Event: time 1510654073.316385, ----- SYN_REPORT -----
```

```
root@imx7dsabresd:~# rmmod adxl345_imx.ko /* remove module */
```

<http://www.rejoiceblog.com/>

11

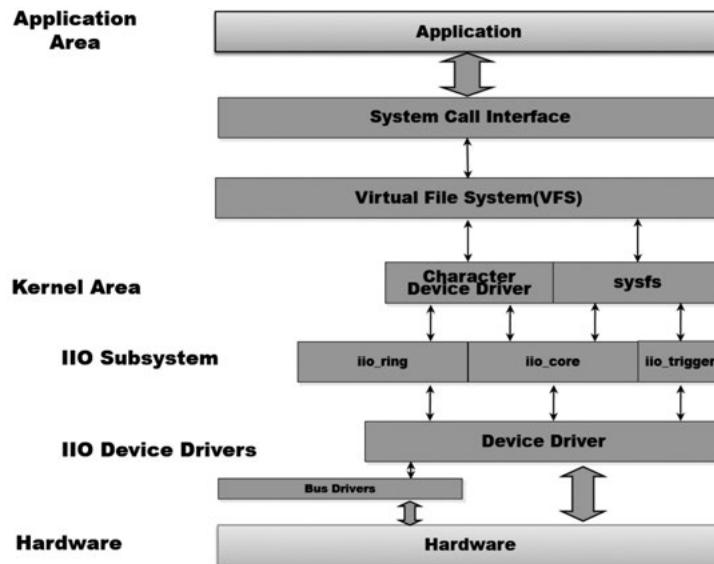
Industrial I/O Subsystem for Device Drivers

The main purpose of the Industrial I/O subsystem (IIO) is to provide support for devices that in some sense perform either analog-to-digital conversion (ADC), digital-to-analog conversion (DAC) or both. The aim is to fill the gap between the somewhat similar hwmon and input subsystems. Hwmon is directed at low sample rate sensors used to monitor and control the system itself, like fan speed control or temperature measurement. Input is, as its name suggests, focused on human interaction input devices (keyboard, mouse, touchscreen). In some cases there is considerable overlap between these and IIO.

Devices that fall into the IIO category include:

- 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 sensors 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.



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.

There are two ways for an user space application to interact with an IIO driver:

1. `/sys/bus/iio/iio:deviceX` - this represents a hardware sensor and groups together the data channels of the same chip.
2. `/dev/iio:deviceX` - this is the character device node interface used for buffered data transfer and for events information retrieval.

A typical IIO driver will register itself as an I2C or SPI driver and 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 IIO device fields with driver specific information (e.g., device name, device channels).

3. The driver will call `devm_iio_device_register()`, that 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 expose chip info and also allowing applications to set various configuration parameters. For a device with index X, attributes can be found under `/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/`, `scan elements/` subdirectories.

The available standard attributes for IIO devices are described in the `linux/Documentation/ABI/testing/sysfs-bus-iio` file.

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 `struct iio_chan_spec` structure.

In the next lab, you are going to develop a kernel module that controls three channels of a dual digital-to-analog converter, channel 0 represents the digital value sent to DAC A, channel 1 represents the digital value sent to DAC B, and channel 2 represents a digital value that will be sent simultaneously to both DAC A and DACB. You can see below the IIO channels definition that will be used in the next developed digital-to-analog converter driver:

```
static const struct iio_chan_spec ltc2607_channel[] = {
{
    .type      = IIO_VOLTAGE,
    .indexed   = 1,
    .output    = 1,
```

```

    .channel      = 0,
    .info_mask_separate = BIT(II0_CHAN_INFO_RAW),
}, {
    .type      = II0_VOLTAGE,
    .indexed   = 1,
    .output    = 1,
    .channel   = 1,
    .info_mask_separate = BIT(II0_CHAN_INFO_RAW),
}, {
    .type      = II0_VOLTAGE,
    .indexed   = 1,
    .output    = 1,
    .channel   = 2,
    .info_mask_separate = BIT(II0_CHAN_INFO_RAW),
}
};
```

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 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 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 channels definition above will generate the following data channel access attributes 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 char array pointers const iio_direction 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 char array pointers const iio_chan_type_name_spec:

```
static const char * const iio_chan_type_name_spec[] = {
    [IIO_VOLTAGE] = "voltage",
    [IIO_CURRENT] = "current",
    [IIO_POWER] = "power",
    [IIO_ACCEL] = "accel",
    [...]
    [IIO_UVINDEX] = "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 char array pointers const iio_modifier_names:

```
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 char array pointers const iio_chan_info_postfix:

```
/* relies on pairs of these shared then separate */
static const char * const iio_chan_info_postfix[] = {
    [IIO_CHAN_INFO_RAW] = "raw",
    [IIO_CHAN_INFO_PROCESSED] = "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 the core can use for this device. There are a lot of hooks available corresponding to interactions that user space can make through sysfs attributes. The read/write operations to sysfs data channel access attributes are mapped to kernel callbacks:

```
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. Writing for example a value x to the out_voltage0_raw sysfs attribute calls the write_raw hook, with the mask argument set to IIO_CHAN_INFO_RAW, the chan argument set with the struct 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 /sys/bus/iio/iio:deviceX/buffer/ directory. Here are some of the existing attributes:

- length, the total number of data samples (capacity) that can be stored by the buffer.
- enable, activate buffer capture.

The meta information associated with a channel reading placed in a buffer is called a scan element. The important bits configuring scan elements are exposed to user space applications via the sysfs entries under /sys/bus/iio/iio:deviceX/scan_elements/ directory. This directory contains type attributes:

- type, description of 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, if specified, 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 with 12 bit resolution where data is 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
```

An user space application will interpret 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

For implementing buffer support, a driver should initialize the following fields (marked in bold below) within a struct iio_chan_spec structure:

```
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;
};
```

You will implement an accelerometer driver in the last chapter of this book with the following channel definition, where you can see the initialization of the previous fields:

```
static const struct iio_chan_spec adxl345_channels[] = {
    ADXL345_CHANNEL(DATAX0, X, 0),
    ADXL345_CHANNEL(DATAY0, Y, 1),
    ADXL345_CHANNEL(DATAZ0, 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 an 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 responsible to allocate the **trigger buffer** for your device (usually done 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 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). See a description of the `iio_push_to_buffers_with_timestamp()` function's parameters below:

```
int iio_push_to_buffers_with_timestamp(struct iio_dev * indio_dev,
                                      void * data,
                                      int64_t timestamp)
```

- `struct iio_dev * indio_dev`: pointer to `struct iio_dev` structure.
- `void * data`: sample data
- `int64_t timestamp`: timestamp for the sample data

Triggers

In many situations, it is useful for a driver to be able to capture data based on some external event (trigger) as opposed to 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 also it 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 IIO triggers. 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 allow us to use sysfs entry to trigger 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 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 struct `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 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()` function's 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`: pointer to the IIO device structure.
- `irqreturn_t (*h)(int irq, void *p)`: 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 recording of the current timestamp and for this reason one can use the IIO core defined `iio_pollfunc_store_time()` function.
- `irqreturn_t (*thread)(int irq, void *p)`: function which will be used as pollfunc bottom half. This 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 how looks like the triggered buffer setup of the ADXL345 IIO driver that will be developed 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 = DATAX0;
    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 a struct `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 `struct iio_event_spec` structure for the ADXL345 IIO driver that will be developed 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)  
};
```

This `adxl345_event` structure will be integrated in each `struct iio_chan_spec` structure, as you can see in the next line of code in bold:

```
static const struct iio_chan_spec adxl345_channels[] = {
    ADXL345_CHANNEL(DATAX0, X, 0),
    ADXL345_CHANNEL(DATAY0, Y, 1),
    ADXL345_CHANNEL(DATAZ0, 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 have to create the kernel hooks corresponding to interactions that user space can make through 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, 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

See below how to deliver IIO events to user space inside the ISR of the ADXL345 IIO driver that will be developed 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;
}
```

The access from user space is via an IOCTL interface on the character device /dev/iio:deviceX. User space application will set up monitoring via IIO_GET_EVENT_FD_IOCTL, then polls for events. To receive an event, follow these steps after setting the threshold using the event sysfs entries:

1. Include <linux/iio/events.h> for the events and ioctl definitions:

```
#include <linux/iio/events.h>
```

2. Use the iio:deviceX directory name and open the device file /dev/iio:deviceX.

3. Use the fd of the step 2 to get the event file descriptor:

```
ioctl(fd, IIO_GET_EVENT_FD_IOCTL, &event_fd)
```

4. Call read on this event_fd:

```
read(event_fd, &event, sizeof(event));
```

Here, event is of type struct iio_event_data. Decipher this event as described in events.h file to locate the exact event. This read is a blocking call, which is released when any event has occurred.

IIO Utils

There are some useful tools you can use during your IIO driver development. They are available under /tools/iio/ folder:

- **lssiio**: 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, and available at <https://github.com/analogdevicesinc/libiio>.

LAB 11.1: "IIO subsystem DAC" Module

This new kernel module will control the Analog Devices LTC2607 device. The LTC2607 is a dual 12-bit, 2.7V to 5.5V rail-to-rail voltage output DAC. It uses a 2-wire, I2C compatible serial interface. The LTC2607 operates in both the standard mode (clock rate of 100kHz) and the fast mode (clock rate of 400kHz).

The driver will control each LTC2607 internal DAC individually or both DACA + DACB in a simultaneous mode. The IIO framework will generate three separate sysfs files (attributes) used for sending data to the dual DAC from user space application.

You will implement for this driver the same hardware description of the lab 6.2 connecting the SDA and SCL pins of the processor to the SDA and SCL pins of the LTC2607 DC934A evaluation board.

Download the schematics of the Analog Devices DC934A evaluation board at <http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/dc934a.html>.

You are going to power the LTC2607 with a 5V supply connected to V+, pin 1 of the DC934A's connector J1. Also connect GND between the DC934A (i.e., pin 3 of connector J1) and processor's boards. You will use the device U3's (LT1790ACS6-5) 5V output as the VREF (select JP1 VREFA jumper) and this 5V output will be also used to supply the LTC2607 DAC (select 5V REF in jumper JP2).

Using the 5V regulator (5V REG in JP2) as the source for VCC has the limitation that VCC may be slightly lower than VREF, which may affect the full-scale error. Selecting the 5V REF as the source for VCC overcomes this, however the total current that the LTC2607 can source will be limited to approximately 5mA.

Remove device U7 from the DC934A board and also the LTC2607 I2C pull-up resistors.

In the LTC2607 device, CA0, CA1, and CA2 are set to Vcc. If you go to the LTC2607 data-sheet you can see that it matches with the next I2C slave address= 01110010=0x72.

The main code sections of the driver will be described using three categories: Device Tree, Industrial Framework as an I2C Interaction, and Industrial Framework as an IIO device.

Device Tree

Modify the device tree files under arch/arm/boot/dts/ folder to include your DT driver's 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.

For the **MCIMX7D-SABRE** Board open the DT file imx7d-sdb.dts and add the ltc2607@72 and ltc2607@73 sub-nodes inside the i2c3 controller master node. The reg properties provide the LTC2607 I2C addresses. The I2C address = 0x72 is provided setting CA0, CA1, and CA2 to Vcc, and the I2C address = 0x73 is always present in the device in addition to the HW selected I2C address.

```
&i2c3 {
    clock-frequency = <100000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c3>;
    status = "okay";

    ltc2607@72 {
        compatible = "arrow,ltc2607";
        reg = <0x72>;
    };

    ltc2607@73 {
        compatible = "arrow,ltc2607";
        reg = <0x73>;
    };
};
```

```
sii902x: sii902x@39 {
    compatible = "SiI,sii902x";
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_sii902x>;
    interrupt-parent = <&gpio2>;
    interrupts = <13 IRQ_TYPE_EDGE_FALLING>;
    mode_str = "1280x720M@60";
    bits-per-pixel = <16>;
    reg = <0x39>;
    status = "okay";
};

[...]

};
```

For the **SAMA5D2B-XULT** Board open the DT file `at91-sama5d2_xplained_common.dtsi` and add the `ltc2607@72` and `ltc2607@73` sub-nodes inside the `i2c1` controller master node.

```
i2c1: i2c@fc028000 {
    dmas = <0>, <0>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c1_default>;
    status = "okay";

[...]

    ltc2607@72 {
        compatible = "arrow,ltc2607";
        reg = <0x72>;
    };

    ltc2607@73 {
        compatible = "arrow,ltc2607";
        reg = <0x73>;
    };

[...]

    at24@54 {
        compatible = "atmel,24c02";
        reg = <0x54>;
        pagesize = <16>;
    };
};
```

For the **Raspberry Pi 3 Model B** Board open the DT file `bcm2710-rpi-3-b.dts` and add the `ltc2607@72` and `ltc2607@73` sub-nodes below the `i2c1` controller master node.

```
&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    clock-frequency = <100000>;
    status = "okay";

[...]

ltc2607@72 {
    compatible = "arrow,ltc2607";
    reg = <0x72>;
};

ltc2607@73 {
    compatible = "arrow,ltc2607";
    reg = <0x73>;
};
};
```

Build the modified device tree and load it to your target processor.

Industrial 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 a struct i2c_driver structure:

```
static struct i2c_driver ltc2607_driver = {
    .driver = {
        .name    = LTC2607_DRV_NAME,
        .owner   = THIS_MODULE,
        .of_match_table = dac_dt_ids,
    },
    .probe     = ltc2607_probe,
    .remove    = ltc2607_remove,
    .id_table  = ltc2607_id,
};
```

3. Register to the I2C bus as a driver:

```
module_i2c_driver(ltc2607_driver);
```

4. Add "Ltc2607" to the list of devices supported by the driver:

```
static const struct of_device_id dac_dt_ids[] = {
    { .compatible = "arrow,ltc2607", },
```

```
    { }
};

MODULE_DEVICE_TABLE(of, dac_dt_ids);
```

5. Define an array of struct i2c_device_id structures:

```
static const struct i2c_device_id ltc2607_id[] = {
    { "ltc2607", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, ltc2607_id);
```

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. 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 Industrial 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. This way, when the remove() routine is called (typically if the bus detects the removal of a device), you can find out which logical device to unregister. Conversely, when there is an event on the logical side (such as opening or closing an input device for the first time), you can find out which I2C slave this corresponds to, to do the specific things with the hardware. Each device's private structure is allocated dynamically each time probe() is called. A pointer to the private structure has to be stored somewhere. The bus abstraction gives us a void pointer to a struct device. Some getter/setter functions are defined e.g., i2c_set_clientdata()/i2c_get_clientdata() so that you can attach/get each private structure to/from the struct device at probe()/remove() calls.

Now, add the next private structure definition to your driver code:

```
struct ltc2607_device {
    struct i2c_client *client;
    char name[8];
};
```

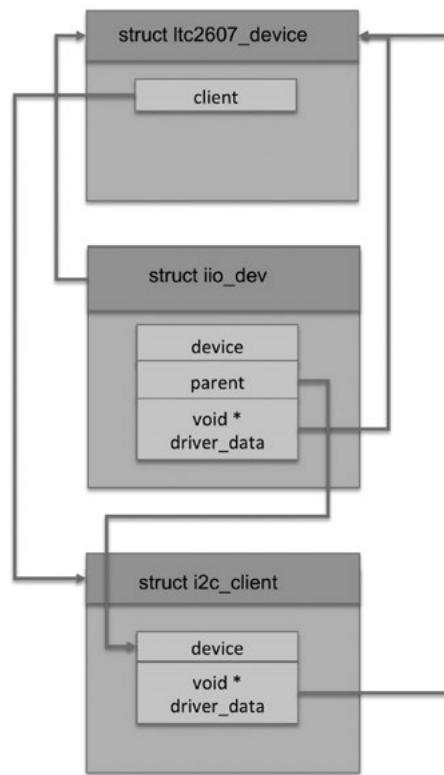
3. Allocate the struct iio_dev structure in ltc2607_probe() using the devm_iio_device_alloc() function:

```
struct iio_dev *indio_dev;
indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*data));
```

4. Initialize the struct iio_device and the data private structure within the ltc2607_probe() function. The data private structure should be previously allocated using the iio_priv() function. Keep pointers between physical devices (devices as handled by the physical bus, I2C in this case) and logical devices:

```
struct ltc2607_device *data;  
  
data = 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 "data" to the  
private structure using the same function iio_priv() */  
  
data->client = client; /* Keep pointer to the I2C device, needed for exchanging  
data with the LTC2607 device */  
  
sprintf(data->name, "DAC%02d", counter++); /* create a different name for each  
device attached to the DT. In the driver two DAC names will be created, one  
for each i2c address. Store the names in each private structure. The probe()  
function will be called twice, once per DT LTC2607 node found */  
  
indio_dev->name = data->name; /* store the name in the IIO device */  
  
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 = &ltc2607_info; /* store the address of the iio_info structure  
which contains a pointer variable to the IIO raw writing callback */  
  
indio_dev->channels = ltc2607_channel; /* store address of the iio_chan_spec  
structure which stores each channel info for the LTC2607 dual DAC */  
  
indio_dev->num_channels = 3; /* set number of channels of the device */  
indio_dev->modes = INDIO_DIRECT_MODE;
```

See the links between physical and logical devices structures of the LTC2607 driver in the following image:



5. Register the devices to the IIO core (two devices will be registered, one with I2C address 0x72 and other one with I2C address 0x73). Now, the devices are global to the rest of the driver functions until they are unregistered. After this call, the devices are ready to accept requests from user space applications. The `probe()` function will be called twice, registering one of the devices in each call. You will control the same device with two different I2C addresses, this has the effect that two different devices are connected to the bus, but you are adding only one physical device. We are doing this for teaching purposes.

```
devm_iio_device_register(&client->dev, indio_dev);
```

6. An IIO device channel is a representation of a data channel. An IIO device can have one or multiple channels. Add the code below for the LTC2607 IIO channels definition:

```
static const struct iio_chan_spec ltc2607_channel[] = {  
    {  
        .type          = IIO_VOLTAGE,  
        .indexed      = 1,  
    },
```

```
        .output          = 1,  
        .channel        = 0,  
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),  
    }, {  
        .type          = IIO_VOLTAGE,  
        .indexed      = 1,  
        .output        = 1,  
        .channel        = 1,  
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),  
    }, {  
        .type          = IIO_VOLTAGE,  
        .indexed      = 1,  
        .output        = 1,  
        .channel        = 2,  
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),  
    }  
};
```

The IIO channels definition above will generate the following data channel access attributes for each iio:device:

```
/sys/bus/iio/devices/iio:device0/out_voltage0_raw  
/sys/bus/iio/devices/iio:device0/out_voltage1_raw  
/sys/bus/iio/devices/iio:device0/out_voltage2_raw  
  
/sys/bus/iio/devices/iio:device1/out_voltage0_raw  
/sys/bus/iio/devices/iio:device1/out_voltage1_raw  
/sys/bus/iio/devices/iio:device1/out_voltage2_raw
```

The attribute's name is automatically generated by the IIO core with the following pattern:
{direction} {type} {index} {modifier} {info mask}:

out_voltage0_raw is the sysfs entry where you write the digital value that is sent to DACA, out_voltage1_raw is the sysfs entry where you write the digital value that is sent to DACB, and out_voltage2_raw is the sysfs entry where you write the digital value that is sent simultaneously to both DACA and DACB.

7. Write the struct `iio_info` structure. The read/write user space operations to sysfs data channel access attributes are mapped to kernel callbacks.

```
static const struct iio_info ltc2607_info = {
    .write_raw = ltc2607_write_raw,
    .driver_module = THIS_MODULE,
};
```

You will write a single IIO write_raw writing callback function named `ltc2607_write_raw()` to map in the kernel all the user space write operations to the sysfs data channel attributes

of both IIO devices. This kernel function will receive the next parameters when a sysfs attribute is accessed from user space:

- struct iio_dev *indio_dev: a pointer to the struct iio_dev structure related with the accessed device.
- struct iio_chan_spec const *chan: the accessed channel number of the IIO device.
- int val: The value written from user space to the sysfs attribute.
- long mask: The info_mask included in the accessed sysfs attribute name.

The ltc2607_write_raw() function contains a switch(mask) that is setting different tasks depending of the received parameter values. If the received info_mask value is [IIO_CHAN_INFO_RAW] = "raw", the ltc2607_set_value() function is called recovering the private data through the iio_priv() function. Once the private info is recovered, the I2C device address will be retrieved from the I2C client pointer variable (data->client). This data->client pointer variable is the first parameter of the i2c_master_send() function, which is used to communicate with the Analog Devices DAC writing each channel in an independent way, or in a simultaneous mode.

Use the command value 0x03 to write and update the DAC device. The DAC values will range from 0 to 0xFFFF (65535). For example, if the DAC value is set to 0xFFFF, then the DAC output is close to 5V (Vref). Depending of the DAC address, the data will be written to DACA (0x00), DACB (0x01), or both DACs (0x0F). In the next table, you can see all the commands and addresses with their respective descriptions.

Output Voltage = Vref x DAC value/65535

COMMAND*						
C3	C2	C1	C0			
0	0	0	0	Write to Input Register		
0	0	0	1	Update (Power Up) DAC Register		
0	0	1	1	Write to and Update (Power Up)		
0	1	0	0	Power Down		
1	1	1	1	No Operation		
ADDRESS*						
A3	A2	A1	A0			
0	0	0	0	DAC A		
0	0	0	1	DAC B		
1	1	1	1	All DACs		

See below the code of the `ltc2607_write_raw()` and the `ltc2607_set_value()` functions:

```
static int ltc2607_set_value(struct iio_dev *indio_dev, int val, int channel)
{
    struct ltc2607_device *data = iio_priv(indio_dev);
    u8 outbuf[3];
    int chan;

    if (channel == 2)
        chan = 0x0F;
    else
        chan = channel;

    if (val >= (1 << 16) || val < 0)
        return -EINVAL;

    outbuf[0] = 0x30 | chan; /* write and update DAC */
    outbuf[1] = (val >> 8) & 0xff; /* MSB byte of dac_code */
    outbuf[2] = val & 0xff; /* LSB byte of dac_code */

    i2c_master_send(data->client, outbuf, 3);
    return 0;
}

static int ltc2607_write_raw(struct iio_dev *indio_dev,
                           struct iio_chan_spec const *chan,
                           int val, int val2, long mask)
{
    int ret;

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        ret = ltc2607_set_value(indio_dev, val, chan->channel);
        return ret;
    default:
        return -EINVAL;
    }
}
```

See in the next **Listing 11-1** the "IIO subsystem DAC" driver source code (`ltc2607_imx_dual_device.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`ltc2607_sam_dual_device.c`) and BCM2837 (`ltc2607_rpi_dual_device.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 11-1: ltc2607_imx_dual_device.c

```
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/iio/iio.h>

#define LTC2607_DRV_NAME "ltc2607"

struct ltc2607_device {
    struct i2c_client *client;
    char name[8];
};

static const struct iio_chan_spec ltc2607_channel[] = {
{
    .type      = IIO_VOLTAGE,
    .indexed   = 1,
    .output    = 1,
    .channel   = 0,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
}, {
    .type      = IIO_VOLTAGE,
    .indexed   = 1,
    .output    = 1,
    .channel   = 1,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
}, {
    .type      = IIO_VOLTAGE,
    .indexed   = 1,
    .output    = 1,
    .channel   = 2,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
}
};

static int ltc2607_set_value(struct iio_dev *indio_dev, int val, int channel)
{
    struct ltc2607_device *data = iio_priv(indio_dev);
    u8 outbuf[3];
    int ret;
    int chan;

    if (channel == 2)
        chan = 0x0F;
    else
        chan = channel;
}
```

```
if (val >= (1 << 16) || val < 0)
    return -EINVAL;

outbuf[0] = 0x30 | chan; /* write and update DAC */
outbuf[1] = (val >> 8) & 0xff; /* MSB byte of dac_code */
outbuf[2] = val & 0xff; /* LSB byte of dac_code */

ret = i2c_master_send(data->client, outbuf, 3);
if (ret < 0)
    return ret;
else if (ret != 3)
    return -EIO;
else
    return 0;
}

static int ltc2607_write_raw(struct iio_dev *indio_dev,
                           struct iio_chan_spec const *chan,
                           int val, int val2, long mask)
{
    int ret;

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        ret = ltc2607_set_value(indio_dev, val, chan->channel);
        return ret;
    default:
        return -EINVAL;
    }
}

static const struct iio_info ltc2607_info = {
    .write_raw = ltc2607_write_raw,
    .driver_module = THIS_MODULE,
};

static int ltc2607_probe(struct i2c_client *client,
                        const struct i2c_device_id *id)
{
    static int counter = 0;
    struct iio_dev *indio_dev;
    struct ltc2607_device *data;
    u8 inbuf[3];
    u8 command_byte;
    int err;
    dev_info(&client->dev, "DAC_probe()\n");
```

```
command_byte = 0x30 | 0x00; /* Write and update register with value 0xFF */
inbuf[0] = command_byte;
inbuf[1] = 0xFF;
inbuf[2] = 0xFF;

/* Allocate the iio_dev structure */
indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*data));
if (indio_dev == NULL)
    return -ENOMEM;

data = iio_priv(indio_dev);
i2c_set_clientdata(client, data);
data->client = client;

sprintf(data->name, "DAC%02d", counter++);
dev_info(&client->dev, "data_probe is entered on %s\n", data->name);

indio_dev->name = data->name;
indio_dev->dev.parent = &client->dev;
indio_dev->info = &ltc2607_info;
indio_dev->channels = ltc2607_channel;
indio_dev->num_channels = 3;
indio_dev->modes = INDIO_DIRECT_MODE;

err = i2c_master_send(client, inbuf, 3); /* write DAC value */
if (err < 0) {
    dev_err(&client->dev, "failed to write DAC value");
    return err;
}

dev_info(&client->dev, "the dac answer is: %x.\n", err);

err = devm_iio_device_register(&client->dev, indio_dev);
if (err)
    return err;

dev_info(&client->dev, "ltc2607 DAC registered\n");

return 0;
}

static int ltc2607_remove(struct i2c_client *client)
{
    dev_info(&client->dev, "DAC_remove()\n");
    return 0;
}

static const struct of_device_id dac_dt_ids[] = {
```

```
    { .compatible = "arrow,ltc2607", },
    { }
};

MODULE_DEVICE_TABLE(of, dac_dt_ids);

static const struct i2c_device_id ltc2607_id[] = {
    { "ltc2607", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, ltc2607_id);

static struct i2c_driver ltc2607_driver = {
    .driver = {
        .name    = LTC2607_DRV_NAME,
        .owner   = THIS_MODULE,
        .of_match_table = dac_dt_ids,
    },
    .probe     = ltc2607_probe,
    .remove    = ltc2607_remove,
    .id_table  = ltc2607_id,
};
module_i2c_driver(ltc2607_driver);

MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("LTC2607 16-bit DAC");
MODULE_LICENSE("GPL");
```

LAB 11.2: "IIO subsystem DAC" Module with "SPIDEV dual ADC user" Application

You have just developed a driver for the dual DAC, and the next challenge will be to develop a driver that reads the analog outputs from the DAC device. To accomplish this task, you will use the LTC2422 dual ADC SPI device that is included in the DC934A board. The LTC2607 DAC outputs are connected to both LTC2422 ADC inputs. Prior to developing the driver you are going to read the DAC analog outputs using the Linux user space SPI API.

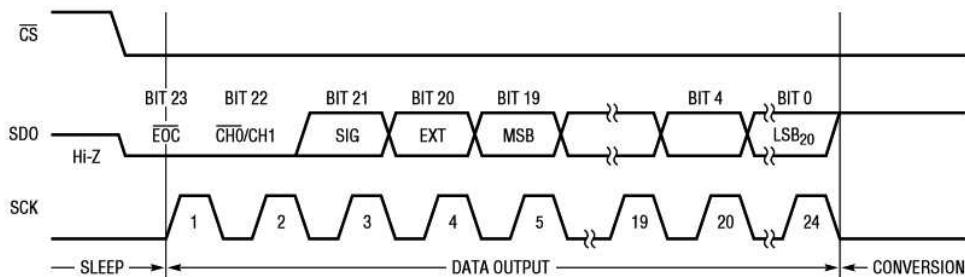
The LTC2422 Analog Devices device is a 2-channel 2.7V to 5.5V micropower 20-bit analog-to-digital converter with an integrated oscillator, 8ppm INL and 1.2ppm RMS noise. This device uses delta-sigma technology and a new digital filter architecture that settles in a single cycle. This eliminates the latency found in conventional sigma delta converters and simplifies multiplexed applications. This converter accept an external reference voltage from 0.1V to VCC.

The LTC2422 serial output data stream is 24 bits long. The first 4 bits represent status information indicating the sign, selected channel, input range and conversion state. The next 20 bits are the conversion result, MSB first:

- **Bit 23** (first output bit) is the end of conversion (EOC) indicator. This bit is available at the SDO pin during the conversion and sleep states whenever the CS pin is LOW. This bit is HIGH during the conversion and goes LOW when the conversion is complete.
- **Bit 22** (second output bit) for the LTC2422, this bit is LOW if the last conversion was performed on CH0 and HIGH for CH1. This bit is always LOW for the LTC2421.
- **Bit 21** (third output bit) is the conversion result sign indicator (SIG). If VIN is > 0 , this bit is HIGH. If VIN is < 0 , this bit is LOW. The sign bit changes state during the zero code.
- **Bit 20** (fourth output bit) is the extended input range (EXR) indicator. If the input is within the normal input range $0 \leq \text{VIN} \leq \text{VREF}$, this bit is LOW. If the input is outside the normal input range, $\text{VIN} > \text{VREF}$ or $\text{VIN} < 0$, this bit is HIGH.
- **Bit 19** (fifth output bit) is the most significant bit (MSB).
- **Bits 19-0** are the 20-bit conversion result MSB first.
- **Bit 0** is the least significant bit (LSB).

Data is shifted out of the SDO pin under control of the serial clock (SCK), whenever CS is HIGH, SDO remains high impedance and any SCK clock pulses are ignored by the internal data out shift register. In order to shift the conversion result out of the device, CS must first be driven LOW. EOC is seen at the SDO pin of the device once CS is pulled LOW. EOC changes real time from HIGH to LOW at the completion of a conversion. This signal may be used as an interrupt for an external microcontroller. Bit 23 (EOC) can be captured on the first rising edge of SCK. Bit 22 is shifted out of the device on the first falling edge of SCK. The final data bit (Bit 0) is shifted out on the falling edge

of the 23rd SCK and may be latched on the rising edge of the 24th SCK pulse. On the falling edge of the 24th SCK pulse, SDO goes HIGH indicating a new conversion cycle has been initiated. This bit serves as EOC (Bit23) for the next conversion cycle.



In the DC934A schematic you can see that the LTC2607 DAC output B is connected to LTC2422 channel 0 and that the LTC2607 DAC output A is connected to the LTC2422 channel 1.

There is a generic SPI device driver `spidev.c` located under `drivers/spi/`, which you can enable through the kernel configuration `CONFIG_SPI_SPIDEV`. Configure your kernel and add `spidev` driver selecting Device Drivers -> SPI support -> <*> User mode SPI device driver. It creates a device node for each SPI controller, which allows you to access SPI chips from user space. The device nodes are named `spidev[bus].[chip select]`.

In this lab you are going to use this `spidev` driver to access to the LTC2422 SPI device.

SPI devices have a limited Linux user space API, supporting basic half-duplex `read()` and `write()` access to SPI slave devices. Using `ioctl()` requests, full duplex transfers and device I/O configurations are also available.

Some reasons you might want to use this user space interface include:

- Prototyping in an environment that's not crash-prone; stray pointers in user space won't normally bring down a Linux system.
- Developing simple protocols used to talk to microcontrollers acting as SPI slaves, which you may need to change quite often.

Of course, there are drivers that can never be written in user space, because they need to access kernel interfaces (such as IRQ handlers or other layers of the driver stack) that are not accessible to user space. See more info about the `spidev` driver at <https://www.kernel.org/doc/Documentation/spi/spidev>.

LAB 11.2 Hardware Description for the i.MX7D Processor

In this lab, you will use the SPI pins of the MCIMX7D-SABRE board mikroBUS™ to connect to the LTC2422 dual ADC SPI device that is included in the DC934A board.

Go to the pag.20 of the MCIMX7D-SABRE schematic to see the MikroBUS connector and look for the SPI pins. The CS, SCK and MISO (Master In, Slave Out) signals will be used. The MOSI (Master out, Slave in) signal won't be needed, as you are only going to receive data from the LTC2422 device. Connect the next processor's pins to the LTC2422 SPI ones obtained from the DC934A board J1 connector:

- Connect i.MX7D **MKBUS_ESPI3_SS0_B** (CS) to LTC2422 **CS**
- Connect i.MX7D **MKBUS_ESPI3_SCLK** (SCK) to LTC2422 **SCK**
- Connect i.MX7D **MKBUS_ESPI3_MISO** (MISO) to LTC2422 **MISO**

LAB 11.2 Hardware Description for the SAMA5D2 Processor

For the SAMA5D2 processor, open the SAMA5D2B-XULT board schematic and look for connectors on board with pins that provide SPI signals.

You can access to the SPI signals using the J17 header. See below a description of the connections between both boards:

- Connect SAMA5D2 **ISC_PCK/SPI1_NPCS0_PC4** (pin 26 of J17) to LTC2422 **CS**
- Connect SAMA5D2 **SC_D7/SPI1_SPCK_PC1** (pin 17 of J17) to LTC2422 **SCK**
- Connect SAMA5D2 **ISC_D9/SPI1_MISO_PC3** (pin 22 of J17) to LTC2422 **MISO**

LAB 11.2 Hardware Description for the BCM2837 Processor

For the BCM2837 processor, you will use the GPIO expansion connector to obtain the SPI signals. Go to the Raspberry-Pi-3B-V1.2-Schematics to see the J8 connector. See below a description of the connections between both boards:

- Connect BCM2837 **SPI_CE0_N** (pin 24 of J8) to LTC2422 **CS**
- Connect BCM2837 **SPI_SCLK** (pin 23 of J8) to LTC2422 **SCK**
- Connect BCM2837 **SPI_MISO** (pin 21 of J8) to LTC2422 **MISO**

LAB 11.2 Device Tree for the i.MX7D Processor

Modify the device tree file imx7d-sdb.dts adding the spidev@1 sub-node inside the ecspi3 controller master node. The reg property provides the CS number; there are two chip selects inside the ecspi3 node, one for the tsc2046 node and the other one for the spidev node.

```
&ecspi3 {
    fsl.spi-num-chipselects = <1>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ecspi3 &pinctrl_ecspi3_cs>;
    cs-gpios = <&gpio5 9 GPIO_ACTIVE_HIGH>, <&gpio6 22 0>;
    status = "okay";

    tsc2046@0 {
        compatible = "ti,tsc2046";
        reg = <0>;
        spi-max-frequency = <1000000>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_tsc2046_pendown>;
        interrupt-parent = <&gpio2>;
        interrupts = <29 0>;
        pendown-gpio = <&gpio2 29 GPIO_ACTIVE_HIGH>;
        ti,x-min = /bits/ 16 <0>;
        ti,x-max = /bits/ 16 <0>;
        ti,y-min = /bits/ 16 <0>;
        ti,y-max = /bits/ 16 <0>;
        ti,pressure-max = /bits/ 16 <0>;
        ti,x-plate-ohms = /bits/ 16 <400>;
        wakeup-source;
    };

    spidev@1 {
        compatible = "spidev";
        spi-max-frequency = <2000000>; /* SPI CLK Hz */
        reg = <1>;
    };
};
```

LAB 11.2 Device Tree for the SAMA5D2 Processor

For the **SAMA5D2B-XULT** Board open the DT file at91-sama5d2_xplained_common.dtsi and add the spidev@0 sub-node inside the spi1 controller master node.

```
spi1: spi@fc000000 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_spi1_default>;
    status = "okay";
```

```
    spidev@0 {
        compatible = "spidev";
        spi-max-frequency = <2000000>;
        reg = <0>;
    };
};
```

LAB 11.2 Device Tree for the BCM2837 Processor

Open and modify the device tree file `bcm2710-rpi-3-b.dts` adding the `spidev@0` sub-node inside the `spi0` controller master node.

```
&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 = <500000>;
    };
}
```

Create the `LTC2422_spidev` application in `my_apps` project. Modify the application Makefile to build and deploy `LTC2422_spidev` to your target processor.

See in the next **Listing 11-2** the "SPIDEV dual ADC user" application source code (`LTC2422_spidev.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`LTC2422_spidev.c`) and BCM2837 (`LTC2422_spidev.c`) can be downloaded from the GitHub repository of this book.

Listing 11-2: LTC2422_spidev.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
```

```
#include <sys/ioctl.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>

int8_t read_adc();

/* Demo Board Name */
char demo_name[] = "DC934";

/* Global Variable. The LTC2422 LSB value with 5V full-scale */
float LTC2422_lsb = 4.7683761E-6;

/* Global Constants. Set 1 second LTC2422 SPI timeout */
const uint16_t LTC2422_TIMEOUT= 1000;

#define SPI_CLOCK_RATE 2000000 /* SPI Clock in Hz */

#define SPI_DATA_CHANNEL_OFFSET 22
#define SPI_DATA_CHANNEL_MASK (1 << SPI_DATA_CHANNEL_OFFSET)

#define LTC2422_CONVERSION_TIME 137 /* ms */

/* MISO timeout in ms */
#define MISO_TIMEOUT 1000

/*
 * Returns the Data and Channel Number(0=channel 0, 1=Channel 1)
 * Returns the status of the SPI read. 0=successful, 1=unsuccessful.
 */
int8_t LTC2422_read(uint8_t *adc_channel, int32_t *code, uint16_t timeout);

/* Returns the Calculated Voltage from the ADC Code */
float LTC2422_voltage(uint32_t adc_code, float LTC2422_lsb);

int8_t LTC2422_read(uint8_t *adc_channel, int32_t *code, uint16_t timeout)
{
    int fd;
    int ret;
    int32_t value;
    uint8_t buffer[4];
    unsigned int val;

    struct spi_ioc_transfer tr = {
        .tx_buf = 0, /* no data to send */
        .rx_buf = (unsigned long) buffer, /* store received data */
        .delay_usecs = 0, /* no delay */
        .speed_hz = SPI_CLOCK_RATE, /* SPI clock speed (in Hz) */
        .bits_per_word = 8, /* transaction size */
        .len = 3 /* number bytes to transfer */
    };
}
```

```
};

/* Open the device */
fd = open("/dev/spidev2.1", O_RDWR);
if (fd < 0)
{
    close(fd);
    return (1);
}

/* Perform the transfer */
ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
if (ret < 1)
{
    close(fd);
    return (1);
}

/* Close the device */
close(fd);

value  = buffer[0] << 16;
value |= buffer[1] << 8;
value |= buffer[2];

/* Determine the channel number */
*adc_channel = (value & SPI_DATA_CHANNEL_MASK) ? 1 : 0;
printf("the value is %x\n", value);

/* Return the code */
*code = value;

return(0);
}

/* Returns the Calculated Voltage from the ADC Code */
float LTC2422_voltage(uint32_t adc_code, float LTC2422_lsb)
{
    float adc_voltage;
    if (adc_code & 0x200000)
    {
        adc_code &= 0xFFFF;

        /* Clears Bits 20-23 */
        adc_voltage=((float)adc_code)*LTC2422_lsb;
    }
    else
    {
```

```
adc_code &= 0xFFFFF;

/* Clears Bits 20-23 */
adc_voltage = -1*((float)adc_code)*LTC2422_lsb;
}
return(adc_voltage);
}

void delay(unsigned int ms)
{
    usleep(ms*1000);
}

int8_t read_adc()
{
    float adc_voltage;
    int32_t adc_code;
    uint8_t adc_channel;

    /* Array for ADC data
     * Useful because you don't know which channel until the LTC2422 tells you.
     */
    int32_t adc_code_array[2];
    int8_t return_code;

    /* Read ADC. Throw out the stale data */
    LTC2422_read(&adc_channel, &adc_code, LTC2422_TIMEOUT);
    delay(LTC2422_CONVERSION_TIME);

    /* Get current data for both channels */
    return_code = LTC2422_read(&adc_channel, &adc_code, LTC2422_TIMEOUT);

    /* Note that channels may return in any order */
    adc_code_array[adc_channel] = adc_code;
    delay(LTC2422_CONVERSION_TIME);

    /* that is, adc_channel will toggle each reading */
    return_code = LTC2422_read(&adc_channel, &adc_code, LTC2422_TIMEOUT);
    adc_code_array[adc_channel] = adc_code;

    /* The DC934A board connects VOUTA to CH1 */
    adc_voltage = LTC2422_voltage(adc_code_array[1], LTC2422_lsb);
    printf("      ADC A : %6.4f\n", adc_voltage);

    /* The DC934A board connects VOUTB to CH0 */
    adc_voltage = LTC2422_voltage(adc_code_array[0], LTC2422_lsb);
    printf("      ADC B : %6.4f\n", adc_voltage);
    return(return_code);
```

```
}

int main(void)
{
    read_adc();
    printf("Application terminated\n");
    return 0;
}
```

ltc2607_imx_dual_device.ko with LTC2422_spidev Demonstration

"Use i2c-utils suite to interact with the LTC2607 (before loading the module)"
"i2cdetect is a tool of the i2c-tools suite. It is able to probe an i2c bus from user space and report the addresses in use"

```
root@imx7dsabresd:~# i2cdetect -l /* list available buses, LTC2607 is in bus 2 */
i2c-3  i2c            30a50000.i2c          I2C adapter
i2c-1  i2c            30a30000.i2c          I2C adapter
i2c-2  i2c            30a40000.i2c          I2C adapter
i2c-0  i2c            30a20000.i2c          I2C adapter

root@imx7dsabresd:~# i2cdetect -y 2 /* see detected devices, see the "72" and "73" LTC2607 addresses */
0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:      - - - - - - - - - - - - - - - - - - - -
10:  - - - - - - - - - - - - - - - - - - - -
20:  - - - - - - - - - - - - - - - - - - - -
30:  - - - - - - - - - - - - - - - - - - - - UU
40:  - - - - - - - - - - - - - - - - - - - -
50:  - - - - - - - - - - - - - - - - - - - -
60: 60 - - - - - - - - - - - - - - - - - - - -
70:  - - - 72 73 - - - - - - - - - - - - - -
```

root@imx7dsabresd:~# insmod ltc2607_imx_dual_device.ko /* Load the module. You will see that the probe() function is called twice. The reason is that the driver matches two devices with the same DT compatible property. Now you can manage both devices from user space */

```
ltc2607 2-0072: DAC_probe()
ltc2607 2-0072: data_probe is entered on DAC00
```

```
ltc2607 2-0072: the dac answer is: 3.
ltc2607 2-0072: ltc2607 DAC registered
ltc2607 2-0073: DAC_probe()
ltc2607 2-0073: data_probe is entered on DAC01
ltc2607 2-0073: the dac answer is: 3.
ltc2607 2-0073: ltc2607 DAC registered

root@imx7dsabresd:~# cd /sys/bus/iio/devices/
root@imx7dsabresd:/sys/bus/iio/devices# ls /* check the iio devices */
iio:device0 iio:device1 iio:device2 iio:device3 iio_sysfs_trigger

root@imx7dsabresd:/sys/bus/iio/devices/iio:device2# ls /* see the sysfs entries
under iio_device2 */
dev of_node out_voltage1_raw power uevent
name out_voltage0_raw out_voltage2_raw subsystem

root@imx7dsabresd:/sys/bus/iio/devices/iio:device3# ls /* see the sysfs entries
under iio_device3 */
dev of_node out_voltage1_raw power uevent
name out_voltage0_raw out_voltage2_raw subsystem

root@imx7dsabresd:/sys/bus/iio/devices/iio:device3# echo 65535 > out_voltage2_raw
/* set both DAC outputs to 5V */

root@imx7dsabresd:~# ./LTC2422_spidev /* get both ADC ouputs with your app */
the value is 6ffa77
the value is 2ffc59
the value is 6ffa34
    ADC A : 4.9929
    ADC B : 4.9955
Application terminated

root@imx7dsabresd:/sys/bus/iio/devices/iio:device3# echo 0 > out_voltage0_raw /* set
VOUTA to 0V */

root@imx7dsabresd:~# ./LTC2422_spidev /* get both ADC ouputs with your app */
the value is 2ffc5a
the value is 6000c2
the value is 2ffc47
    ADC A : 0.0009
    ADC B : 4.9955
Application terminated

root@imx7dsabresd:/sys/bus/iio/devices/iio:device3# echo 0 > out_voltage1_raw /* set
VOUTB to 0V */

root@imx7dsabresd:~# ./LTC2422_spidev /* get both ADC ouputs with your app */
the value is 600086
the value is 2000dd
the value is 600045
    ADC A : 0.0003
```

```
ADC B : 0.0011
Application terminated

root@imx7dsabresd:~# cd /sys/bus/iio/devices/iio:device2/ /* change to the iio_
device2 */
root@imx7dsabresd:/sys/bus/iio/devices/iio:device2# echo 65535 > out_voltage2_raw /*
set both outputs to 5V */

root@imx7dsabresd:~# ./LTC2422_spidev /* get both ADC ouputs with your app */
the value is 2000ba
the value is 6ffa1c
the value is 2fffc6b
    ADC A : 4.9928
    ADC B : 4.9956
Application terminated

root@imx7dsabresd:~# rmmod ltc2607_imx_dual_device.ko /* remove the module */
ltc2607 2-0073: DAC_remove()
ltc2607 2-0072: DAC_remove()
```

LAB 11.3: "IIO subsystem ADC" Module

You have controlled the LTC2422 from user space using the spidev driver. Now you are going to develop a LTC2422 driver using the IIO framework. You will use this driver to read the two ADC channels values via SPI; then, you will transform these digital values to real analog voltages with the user application LTC2422_app.

The main code sections of the driver will be described using three categories: Device Tree, Industrial Framework as a SPI Interaction, and Industrial Framework as an IIO device.

Device Tree

Modify the device tree files under arch/arm/boot/dts/ folder to include your DT driver's 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.

For the **MCIMX7D-SABRE** Board open the DT file imx7d-sdb.dts and add the ltc2422@1 sub-node inside the ecspi3 controller master node. The reg property provides the CS number; there are two chip selects inside the ecspi3 node, one for the tsc2046 node and the other one for the ltc2422 node.

```
&ecspi3 {
    fsl,spi-num-chipselects = <1>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ecspi3 &pinctrl_ecspi3_cs>;
    cs-gpios = <&gpio5 9 GPIO_ACTIVE_HIGH>, <&gpio6 22 0>;
    status = "okay";
```

```
tsc2046@0 {
    compatible = "ti,tsc2046";
    reg = <0>;
    spi-max-frequency = <1000000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_tsc2046_pendown>;
    interrupt-parent = <&gpio2>;
    interrupts = <29 0>;
    pendown-gpio = <&gpio2 29 GPIO_ACTIVE_HIGH>;
    ti,x-min = /bits/ 16 <0>;
    ti,x-max = /bits/ 16 <0>;
    ti,y-min = /bits/ 16 <0>;
    ti,y-max = /bits/ 16 <0>;
    ti,pressure-max = /bits/ 16 <0>;
    ti,x-plate-ohms = /bits/ 16 <400>;
    wakeup-source;
};

ADC: ltc2422@1 {
    compatible = "arrow,ltc2422";
    spi-max-frequency = <2000000>;
    reg = <1>;
};
};
```

For the **SAMA5D2B-XULT** Board open the DT file `at91-sama5d2_xplained_common.dtsi` and add the `ltc2422@0` sub-node inside the `spi1` controller master node.

```
spi1: spi@fc000000 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_spi1_default>;
    status = "okay";

    ADC: ltc2422@0 {
        compatible = "arrow,ltc2422";
        spi-max-frequency = <2000000>;
        reg = <0>;
    };
};
```

For the **Raspberry Pi 3 Model B** Board open the DT file `bcm2710-rpi-3-b.dts` and add the `ltc2422@0` sub-node below the `spi0` controller master node.

```
&spi0 {
    pinctrl-names = "default";
    pinctrl-0 = <&spi0_pins &spi0_cs_pins>;
    cs-gpios = <&gpio 8 1>, <&gpio 7 1>;
```

```
ADC: ltc2422@0 {
    compatible = "arrow,ltc2422";
    spi-max-frequency = <2000000>;
    reg = <0>;
};

};
```

Build the modified device tree and load it to your target processor.

Industrial Framework as a SPI Interaction

These are the main code sections:

1. Include the required header files:

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

2. Create a struct spi_driver structure:

```
static struct spi_driver ltc2422_driver = {
    .driver = {
        .name    = "ltc2422",
        .owner   = THIS_MODULE,
        .of_match_table = ltc2422_dt_ids,
    },
    .probe     = ltc2422_probe,
    .id_table = ltc2422_id,
};
```

3. Register to the SPI bus as a driver:

```
module_spi_driver(ltc2422_driver);
```

4. Add "ltc2422" to the list of devices supported by the driver:

```
static const struct of_device_id ltc2422_dt_ids[] = {
    { .compatible = "arrow,ltc2422", },
    { }
};
MODULE_DEVICE_TABLE(of, ltc2422_dt_ids);
```

5. Define an array of struct spi_device_id structures:

```
static const struct spi_device_id ltc2422_id[] = {
    { .name = "ltc2422", },
    { }
};
MODULE_DEVICE_TABLE(spi, ltc2422_id);
```

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 private data structure to manage the device.

```
struct ltc2422_state {  
    struct spi_device *spi;  
    u8 buffer[4];  
};
```

3. In the ltc2422_probe() function, declare an instance of the private structure and allocate the iio_dev structure.

```
struct iio_dev *indio_dev;  
struct ltc2422_state *st;  
indio_dev = devm_iio_device_alloc(&spi->dev, sizeof(*st));
```

4. Initialize the iio_device and the data private structure within the ltc2422_probe() function.

The data private structure will be previously allocated 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 "data" to the private structure using the same function iio_priv() */*

st->spi = spi; / Keep pointer to the SPI device, needed for exchanging data with the LTC2422 device */*

indio_dev->name = id->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() */*

indio_dev->dev.parent = &spi->dev; / keep pointers between physical devices (devices as handled by the physical bus, SPI in this case) and logical devices */*

indio_dev->info = <c2422_info; / store the address of the iio_info structure which contains a pointer variable to the IIO raw reading callback */*

indio_dev->channels = ltc2422_channel; / store address of the iio_chan_spec structure which stores each channel info for the LTC2422 dual ADC */*

indio_dev->num_channels = 1; / set number of channels of the device */*

indio_dev->modes = INDIO_DIRECT_MODE;

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(&spi->dev, indio_dev);
```

6. An IIO device channel is a representation of a data channel. An IIO device can have one or multiple channels. Add the code below for the LTC2422 IIO channel definition:

```
static const struct iio_chan_spec ltc2422_channel[] = {
```

```
{  
    .type      = IIO_VOLTAGE,  
    .indexed   = 1,  
    .output    = 1,  
    .channel   = 0,  
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),  
}
```

```
};
```

The IIO channel definition above will generate the following data channel access attribute for the iio:device:

```
/sys/bus/iio/devices/iio:device2/out_voltage0_raw
```

The attribute's name is automatically generated by the IIO core with the following pattern: {direction}_{type}_{index}_{modifier}_{info_mask}:

out_voltage0_raw is the sysfs entry where you are going to read each ADC channel:

```
cat /sys/bus/iio/devices/iio:device2/out_voltage_0_raw /* discard first val */  
cat /sys/bus/iio/devices/iio:device2/out_voltage_0_raw /* read first chan */  
cat /sys/bus/iio/devices/iio:device2/out_voltage_0_raw /* read second chan */
```

7. Write the struct iio_info structure. The read/write user space operations to sysfs data channel access attributes are mapped to kernel callbacks.

```
static const struct iio_info ltc2422_info = {  
    .read_raw = &ltc2422_read_raw,  
    .driver_module = THIS_MODULE,  
};
```

You will write a single IIO read_raw reading callback function named ltc2422_read_raw() to map in the kernel the user reads to the sysfs data channel attribute out_voltage_0_raw. This kernel function will receive the next parameters when a sysfs attribute is read from user space:

- struct iio_dev *indio_dev: pointer to the struct iio_dev structure related with the accessed device.
- struct iio_chan_spec const *chan: the accessed channel of the IIO device.
- long mask: the info_mask included in the accessed sysfs attribute name.

When the ltc2422_read_raw() function receives the info_mask value

[IIO_CHAN_INFO_RAW] = "raw", it reads the ADC channel value using the spi_read() function. Before reading the ADC value, the private info is recovered using the iio_priv() function, then the struct spi_device structure is retrieved from the private structure and sent as a first parameter to the spi_read() function, which is used to communicate with the Analog Devices ADC to get each channel value. The ADC value is stored in the val variable and returned with IIO_VAL_INT.

See below the code of the ltc2422_read_raw() function:

```
static int ltc2422_read_raw(struct iio_dev *indio_dev,
                           struct iio_chan_spec const *chan,
                           int *val, int *val2, long m)
{
    int ret;
    struct ltc2422_state *st = iio_priv(indio_dev);

    switch (m) {
    case IIO_CHAN_INFO_RAW:

        ret = spi_read(st->spi, &st->buffer, 3);
        if (ret < 0)
            return ret;

        *val = st->buffer[0] << 16;
        *val |= st->buffer[1] << 8;
        *val |= st->buffer[2];

        return IIO_VAL_INT;

    default:
        return -EINVAL;
    }
}
```

See in the next **Listing 11-3** the "IIO subsystem ADC" driver source code (ltc2422_imx_dual.c) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (ltc2422_sam_dual.c) and BCM2837 (ltc2422_rpi_dual.c) drivers can be downloaded from the GitHub repository of this book.

Listing 11-3: ltc2422_imx_dual.c

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

struct ltc2422_state {
    struct spi_device *spi;
    u8 buffer[4];
};

static const struct iio_chan_spec ltc2422_channel[] = {

{
    .type          = IIO_VOLTAGE,
    .indexed      = 1,
    .output        = 1,
    .channel       = 0,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
}
};

static int ltc2422_read_raw(struct iio_dev *indio_dev,
    struct iio_chan_spec const *chan, int *val, int *val2, long m)
{
    int ret;
    struct ltc2422_state *st = iio_priv(indio_dev);

    switch (m) {
    case IIO_CHAN_INFO_RAW:

        ret = spi_read(st->spi, &st->buffer, 3);
        if (ret < 0)
            return ret;

        *val  = st->buffer[0] << 16;
        *val |= st->buffer[1] << 8;
        *val |= st->buffer[2];

        dev_info(&st->spi->dev, "the value is %x\n", *val);

        return IIO_VAL_INT;

    default:
        return -EINVAL;
    }
}
```

```
}

static const struct iio_info ltc2422_info = {
    .read_raw = &ltc2422_read_raw,
    .driver_module = THIS_MODULE,
};

static int ltc2422_probe(struct spi_device *spi)
{
    struct iio_dev *indio_dev;
    struct ltc2422_state *st;
    int err;
    dev_info(&spi->dev, "my_probe() function is called.\n");

    const struct spi_device_id *id = spi_get_device_id(spi);

    indio_dev = devm_iio_device_alloc(&spi->dev, sizeof(*st));
    if (indio_dev == NULL)
        return -ENOMEM;

    st = iio_priv(indio_dev);

    st->spi = spi;

    indio_dev->dev.parent = &spi->dev;
    indio_dev->channels = ltc2422_channel;
    indio_dev->info = &ltc2422_info;
    indio_dev->name = id->name;
    indio_dev->num_channels = 1;
    indio_dev->modes = INDIO_DIRECT_MODE;

    err = devm_iio_device_register(&spi->dev, indio_dev);
    if (err < 0)
        return err;

    return 0;
}

static const struct of_device_id ltc2422_dt_ids[] = {
    { .compatible = "arrow,ltc2422", },
    { }
};
MODULE_DEVICE_TABLE(of, ltc2422_dt_ids);

static const struct spi_device_id ltc2422_id[] = {
    { .name = "ltc2422", },
    { }
};
MODULE_DEVICE_TABLE(spi, ltc2422_id);
```

```
static struct spi_driver ltc2422_driver = {
    .driver = {
        .name    = "ltc2422",
        .owner   = THIS_MODULE,
        .of_match_table = ltc2422_dt_ids,
    },
    .probe     = ltc2422_probe,
    .id_table = ltc2422_id,
};

module_spi_driver(ltc2422_driver);

MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("LTC2422 DUAL ADC");
MODULE_LICENSE("GPL");
```

LTC2422_app User Space Application

The LTC2422 ADC 24-bit output word is read using the ltc2422_dual driver, but you will select the reading channel, transform the digital values to analog voltages and display them to the console using the application ltc2422_app.

Create the ltc2422_app application in my_apps project. Modify the application Makefile to build and deploy ltc2422_app.

Listing 11-4: ltc2422_app.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/types.h>

int8_t read_adc();

/* The LTC2422 least significant bit value with 5V full-scale */
float LTC2422_lsb = 4.7683761E-6;

/* The LTC2422 least significant bit value with 3.3V full-scale */
/* float LTC2422_lsb = 3.1471252E-6; */
/* check which number is the ADC iio:deviceX and replace x by the number */
```

```
#define LTC2422_FILE_VOLTAGE      "/sys/bus/iio/devices/iio:device4/out_voltage0_raw"
#define SPI_DATA_CHANNEL_OFFSET 22
#define SPI_DATA_CHANNEL_MASK    (1 << SPI_DATA_CHANNEL_OFFSET)
#define LTC2422_CONVERSION_TIME    137 /* ms */

/*
 * Returns the Data and Channel Number(0- channel 0, 1-Channel 1)
 * Returns the status of the SPI read. 0=successful, 1=unsuccessful.
 */
int8_t LTC2422_read(uint8_t *adc_channel, int32_t *code);

/* Returns the Calculated Voltage from the ADC Code */
float LTC2422_voltage(uint32_t adc_code, float LTC2422_lsb);

int8_t LTC2422_read(uint8_t *adc_channel, int32_t *code)
{
    int a2dReading = 0;
    FILE *f = fopen(LTC2422_FILE_VOLTAGE, "r");
    int read = fscanf(f, "%d", &a2dReading);
    if (read <= 0) {
        printf("ERROR: Unable to read values from voltage input file.\n");
        exit(-1);
    }

    /* Determine the channel number */
    *adc_channel = (a2dReading & SPI_DATA_CHANNEL_MASK) ? 1 : 0;
    *code = a2dReading;
    fclose(f);

    return(0);
}

/* Returns the Calculated Voltage from the ADC Code */
float LTC2422_voltage(uint32_t adc_code, float LTC2422_lsb)
{
    float adc_voltage;
    if (adc_code & 0x200000)
    {
        adc_code &= 0xFFFF; /* Clears Bits 20-23 */
        adc_voltage=((float)adc_code)*LTC2422_lsb;
    }
    else
    {
        adc_code &= 0xFFFF; /* Clears Bits 20-23 */
        adc_voltage = -1*((float)adc_code)*LTC2422_lsb;
    }
    return(adc_voltage);
}
```

```
void delay(unsigned int ms)
{
    usleep(ms*1000);
}

int8_t read_adc()
{
    float adc_voltage;
    int32_t adc_code;
    uint8_t adc_channel;
    int32_t adc_code_array;
    int8_t return_code;
    int a2dReading = 0;

    LTC2422_read(&adc_channel, &adc_code);
    delay(LTC2422_CONVERSION_TIME);

    LTC2422_read(&adc_channel, &adc_code);
    adc_voltage = LTC2422_voltage(adc_code, LTC2422_lsb);
    printf("the value of ADC channel %d\n", adc_channel);
    printf("    is : %6.4f\n", adc_voltage);
    delay(LTC2422_CONVERSION_TIME);

    LTC2422_read(&adc_channel, &adc_code);
    adc_voltage = LTC2422_voltage(adc_code, LTC2422_lsb);
    printf("the value of ADC channel %d\n", adc_channel);
    printf("    is : %6.4f\n", adc_voltage);

    return(0);
}

int main(void)
{
    read_adc();
    printf("Application terminated\n");
    return 0;
}
```

ltc2422_imx_dual.ko with ltc2422_app Demonstration

```
root@imx7dsabresd:~# insmod ltc2607_imx_dual_device.ko /* load the ltc2607 module */
root@imx7dsabresd:~# insmod ltc2422_imx_dual.ko /* load the ltc2422 module */
root@imx7dsabresd:/sys/bus/iio/devices# ls /* check the iio_devices */
iio:device0 iio:device2 iio:device4
iio:device1 iio:device3 iio_sysfs_trigger

root@imx7dsabresd:/sys/bus/iio/devices# cd iio:device4
root@imx7dsabresd:/sys/bus/iio/devices/iio:device4# ls /* see the sysfs entries
under the iio:device4, this is your ADC device */
dev name of_node out_voltage0_raw power subsystem uevent

root@imx7dsabresd:/sys/bus/iio/devices/iio:device3# echo 65535 > out_voltage2_raw
/* set both DAC outputs to 5V */
root@imx7dsabresd:/sys/bus/iio/devices/iio:device4# cat out_voltage0_raw /* read ADC
device and discard first value */
root@imx7dsabresd:/sys/bus/iio/devices/iio:device4# cat out_voltage0_raw /* read
first ADC channel */
root@imx7dsabresd:/sys/bus/iio/devices/iio:device4# cat out_voltage0_raw /* read
second ADC channel */
root@imx7dsabresd:~# ./LTC2422_app /* Load your ADC app that calls the LTC2422_
dual.ko driver and shows analog values. The first readed value is discarded */

ltc2422 spi2.1: the value is 2ffc9a
ltc2422 spi2.1: the value is 6ff9e8
the value of ADC channel 1
    is : 4.9926
ltc2422 spi2.1: the value is 2ffc22
the value of ADC channel 0
    is : 4.9953
Application terminated

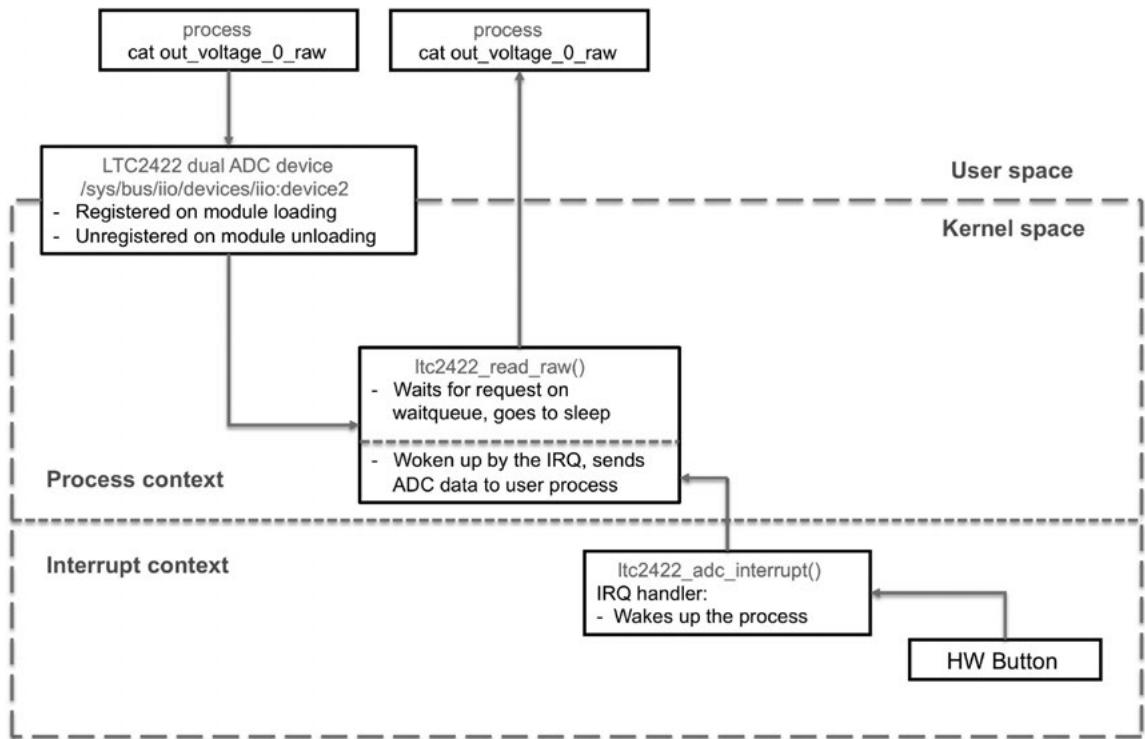
root@imx7dsabresd:~# rmmod ltc2607_imx_dual_device.ko /* remove the DAC module */
root@imx7dsabresd:~# rmmod ltc2422_imx_dual.ko /* remove the ADC module */
```

LAB 11.4: "IIO subsystem ADC with hardware triggering" Module

In this last lab of the chapter, you are going to reuse part of the ltc2422_dual driver, but this time the ADC conversion will be started using a hardware trigger. As in lab 7.1, you will use a button that generates an interrupt triggering the ADC conversion. A wait queue will be used to synchronize the kernel code running in process and interrupt contexts. When the user application reads the out_voltage_0_raw sysfs entry, the process is put to sleep inside the kernel

callback `ltc2422_read_raw()` function. Every time you press the button, the generated interrupt will wake up the process and the driver's read callback function will send to user space the ADC value.

The following figure details the main parts of the new driver:



You will keep the same HW button configuration as in previous lab 7.1 for all the processor variants. You will also keep the same HW SPI ADC configuration as in previous lab 11.3 for all the processor variants.

This time the main code segments included in the driver will not be described in detail. A large portion of the previous lab 11.3 driver will be reused, hence only the new parts will be highlighted.

LAB 11.4 DT for the i.MX7D, SAMA5D2 and BCM2837 Processors

In this lab 11.4, you will keep the same DT configuration as in the previous lab 11.3, but adding an `int-gpios` property for the GPIO pin that is going to be connected to the button and a `pinctrl-0`

property that points to a pin configuration node, where a processor's pad is multiplexed as a GPIO; this GPIO matches with the GPIO pin requested in the int-gpios property.

For the **MCIMX7D-SABRE** Board open the DT file imx7d-sdb.dts and add the ltc2422@1 sub-node inside the ecspi3 controller master node. The reg property provides the CS number; there are two chip selects inside the ecspi3 node, one for the tsc2046 node and another one for the ltc2422 node. The int-gpios property will make the GPIO available to the driver so that you can set the pin direction to input and get the Linux IRQ number associated to this pin.

```
&ecspi3 {
    fsl,spi-num-chipselects = <1>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_ecspi3 &pinctrl_ecspi3_cs>;
    cs-gpios = <&gpio5 9 GPIO_ACTIVE_HIGH>, <&gpio6 22 0>;
    status = "okay";

    tsc2046@0 {
        compatible = "ti,tsc2046";
        reg = <0>;
        spi-max-frequency = <1000000>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_tsc2046_pendown>;
        interrupt-parent = <&gpio2>;
        interrupts = <29 0>;
        pendown-gpio = <&gpio2 29 GPIO_ACTIVE_HIGH>;
        ti,x-min = /bits/ 16 <0>;
        ti,x-max = /bits/ 16 <0>;
        ti,y-min = /bits/ 16 <0>;
        ti,y-max = /bits/ 16 <0>;
        ti,pressure-max = /bits/ 16 <0>;
        ti,x-plate-ohms = /bits/ 16 <400>;
        wakeup-source;
    };

    ADC: ltc2422@1 {
        compatible = "arrow,ltc2422";
        spi-max-frequency = <2000000>;
        reg = <1>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_key_gpio>;
        int-gpios = <&gpio5 10 GPIO_ACTIVE_LOW>;
    };
};
```

Add the pinctrl_key_gpio configuration node inside the iomuxc node:

```
&iomuxc {
    pinctrl-names = "default";
```

```
pinctrl-0 = <&pinctrl_hog_1>;  
  
imx7d-sdb {  
  
    pinctrl_hog_1: hoggrp-1 {  
        fsl,pins = <  
                    MX7D_PAD_EPDC_BDR0__GPIO2_I028      0x59  
                  >;  
    };  
  
    [...]  
  
    pinctrl_key_gpio: key_gpiogrp {  
        fsl,pins = <  
                    MX7D_PAD_SD2_WP__GPIO5_I010      0x32  
                  >;  
    };  
  
    [...]  
};  
};
```

For the **SAMA5D2B-XULT** Board open the DT file `at91-sama5d2_xplained_common.dtsi` and add the `ltc2422@0` sub-node inside the `spi1` controller master node.

```
spi1: spi@fc000000 {  
    pinctrl-names = "default";  
    pinctrl-0 = <&pinctrl_spi1_default>;  
    status = "okay";  
  
    ADC: ltc2422@0 {  
        compatible = "arrow,ltc2422";  
        spi-max-frequency = <2000000>;  
        reg = <0>;  
        pinctrl-0 = <&pinctrl_key_gpio_default>;  
        int-gpios = <&pioA 41 GPIO_ACTIVE_LOW>;  
    };  
};
```

Add the `pinctrl_key_gpio_default` configuration node inside the `pinctrl` node:

```
pinctrl@fc038000 {  
  
    pinctrl_adc_default: adc_default {  
        pinmux = <PIN_PD23_GPIO>;  
        bias-disable;  
    };  
  
    [...]
```

```
pinctrl_key_gpio_default: key_gpio_default {
    pinmux = <PIN_PB9_GPIO>;
    bias-pull-up;
};

[...]

};
```

For the **Raspberry Pi 3 Model B** Board open the DT file `bcm2710-rpi-3-b.dts` and add the `ltc2422@0` sub-node inside the `spi0` controller master node.

```
&spi0 {
    pinctrl-names = "default";
    pinctrl-0 = <&spi0_pins &spi0_cs_pins>;
    cs-gpios = <&gpio 8 1>, <&gpio 7 1>;

    ADC: ltc2422@0 {
        compatible = "arrow,ltc2422";
        spi-max-frequency = <2000000>;
        reg = <0>;
        pinctrl-0 = <&key_pin>;
        int-gpios = <&gpio 23 0>;
    };
};
```

Add the `key_pin` configuration node inside the `gpio` node:

```
&gpio {
    sdhost_pins: sdhost_pins {
        brcm,pins = <48 49 50 51 52 53>;
        brcm,function = <4>; /* alt0 */
    };

    [...]

    key_pin: key_pin {
        brcm,pins = <23>;
        brcm,function = <0>; /* Input */
        brcm,pull = <1>; /* Pull down */
    };
};
```

Build the modified device tree and load it to your target processor.

Sleep and Wake up in the Driver

These are the main steps to put a process to sleep and wake it up in the driver:

1. You will put the process to sleep by means of a wait queue structure. A wait queue is a list of processes, all waiting for a specific event. In Linux, a wait queue is managed by means of a structure of type `wait_queue_head_t`, which is defined in `linux/linux/wait.h`. In your driver, the `wait_queue_head_t` structure will be declared inside the private structure:

```
struct ADC_data {  
    struct gpio_desc      *gpio;  
    int irq;  
    wait_queue_head_t     wq_data_available;  
    struct spi_device    *spi;  
    u8 buffer[4];  
    bool                 conversion_done;  
    struct mutex          lock;  
};
```

then initialized dynamically within the `probe()` function:

```
init_waitqueue_head(&st->wq_data_available);
```

2. Put the user process to sleep. When a process sleeps, it does so in expectation that some condition will become "true" in the future. Any process that sleeps must check to be sure that the condition it was waiting for is really true when it wakes up again. The simplest way of sleeping in the Linux kernel is a macro called `wait_event` (with a few variants); it combines handling the details of sleeping with a check on the condition a process is waiting for. You will use in the driver the `wait_event_interruptible()` variant, which is called inside the `ltc2422_read_raw()` callback function. The `wait_event_interruptible()` function will not wake up the process until the starting of the ADC conversion is signaled by the ISR (condition is set to "true"):

```
static int ltc2422_read_raw(struct iio_dev *indio_dev,  
    struct iio_chan_spec const *chan, int *val, int *val2, long m)  
{  
    struct ADC_data *st = iio_priv(indio_dev);  
  
    switch (m) {  
    case IIO_CHAN_INFO_RAW:  
  
        wait_event_interruptible(st->wq_data_available,  
            st->conversion_done);  
  
        spi_read(st->spi, &st->buffer, 3);  
  
        *val = st->buffer[0] << 16;
```

```
        *val |= st->buffer[1] << 8;
        *val |= st->buffer[2];

        st->conversion_done = false;

        return IIO_VAL_INT;

    default:
        break;
    }
    return -EINVAL;
}
```

3. You will wake up the process inside the interrupt handler:

```
static irqreturn_t ltc2422_adc_interrupt(int irq, void *data)
{
    struct ADC_data *st = data;

    /* set true condition, ADC conversion is starting pressing button */
    st->conversion_done = true;
    wake_up_interruptible(&st->wq_data_available);
    return IRQ_HANDLED;
}
```

Interrupt Management

In the probe() function, you will get the GPIO descriptor from the int-gpios property of the DT ADC node ltc2422 using the devm_gpiod_get_index() function, then you will obtain the Linux IRQ number corresponding to the given GPIO using the function gpiod_to_irq(), which takes the GPIO descriptor as a parameter.

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 struct device, the Linux IRQ number, a handler that will be called when the interrupt is generated (ltc2422_adc_interrupt), a flag that will instruct the kernel about the desired interrupt behaviour (IRQF_TRIGGER_FALLING), the name of the device using this interrupt (id->name), and a pointer variable st that points to your private structure.

```
struct ADC_data *st;
st->gpio = devm_gpiod_get_index(&spi->dev, LTC2422_GPIO_NAME, 0, GPIO_IN);
st->irq = gpiod_to_irq(st->gpio);
devm_request_irq(&spi->dev, st->irq, ltc2422_adc_interrupt,
                 IRQF_TRIGGER_FALLING, id->name, st);
```

See in the next **Listing 11-5** the "IIO subsystem ADC with hardware triggering" driver source code (`ltc2422_imx_trigger.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`ltc2422_sam_trigger.c`) and BCM2837 (`ltc2422_rpi_trigger.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 11-5: `ltc2422_imx_trigger.c`

```
#include <linux/module.h>
#include <linux/spi/spi.h>
#include <linux/interrupt.h>
#include <linux/of_gpio.h>
#include <linux/iio/iio.h>
#include <linux/wait.h>

#define LTC2422_GPIO_NAME "int"

struct ADC_data {
    struct gpio_desc      *gpio;
    int irq;
    wait_queue_head_t     wq_data_available;
    struct spi_device    *spi;
    u8 buffer[4];
    bool                 conversion_done;
    struct mutex          lock;
};

static irqreturn_t ltc2422_adc_interrupt(int irq, void *data)
{
    struct ADC_data *st = data;
    st->conversion_done = true;
    wake_up_interruptible(&st->wq_data_available);
    return IRQ_HANDLED;
}

static const struct iio_chan_spec ltc2422_channel[] = {
{
    .type      = IIO_VOLTAGE,
    .indexed   = 1,
    .output    = 1,
    .channel   = 0,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
}
};
```

```
static int ltc2422_read_raw(struct iio_dev *indio_dev,
                           struct iio_chan_spec const *chan,
                           int *val, int *val2, long m)
{
    int ret;
    struct ADC_data *st = iio_priv(indio_dev);

    dev_info(&st->spi->dev, "Press PB_USER key to start conversion\n");

    switch (m) {
    case IIO_CHAN_INFO_RAW:
        mutex_lock(&st->lock);

        ret = wait_event_interruptible(st->wq_data_available,
                                       st->conversion_done);
        if (ret)
            dev_err(&st->spi->dev, "Failed to request interrupt\n");
        return ret;
    }
    spi_read(st->spi, &st->buffer, 3);

    *val = st->buffer[0] << 16;
    *val |= st->buffer[1] << 8;
    *val |= st->buffer[2];

    st->conversion_done = false;

    mutex_unlock(&st->lock);

    return IIO_VAL_INT;

default:
    break;
}
return -EINVAL;
}

static const struct iio_info ltc2422_info = {
    .read_raw = &ltc2422_read_raw,
    .driver_module = THIS_MODULE,
};

static int ltc2422_probe(struct spi_device *spi)
{
    struct iio_dev *indio_dev;
    struct ADC_data *st;
    int ret;
    dev_info(&spi->dev, "my_probe() function is called.\n");
```

```
/* get the id from the driver structure to use the name */
const struct spi_device_id *id = spi_get_device_id(spi);

indio_dev = devm_iio_device_alloc(&spi->dev, sizeof(*st));
if (indio_dev == NULL)
    return -ENOMEM;

st = iio_priv(indio_dev);
st->spi = spi;
spi_set_drvdata(spi, indio_dev);

/*
 * you can also use
 * devm_gpiod_get(&spi->dev, LTC2422_GPIO_NAME, GPIOD_IN);
 */
st->gpio = devm_gpiod_get_index(&spi->dev, LTC2422_GPIO_NAME, 0, GPIOD_IN);
if (IS_ERR(st->gpio)) {
    dev_err(&spi->dev, "gpio get index failed\n");
    return PTR_ERR(st->gpio);
}

st->irq = gpiod_to_irq(st->gpio);
if (st->irq < 0)
    return st->irq;
dev_info(&spi->dev, "The IRQ number is: %d\n", st->irq);

indio_dev->dev.parent = &spi->dev;
indio_dev->channels = ltc2422_channel;
indio_dev->info = &ltc2422_info;
indio_dev->name = id->name;
indio_dev->num_channels = 1;
indio_dev->modes = INDIO_DIRECT_MODE;

init_waitqueue_head(&st->wq_data_available);
mutex_init(&st->lock);

ret = devm_request_irq(&spi->dev, st->irq, ltc2422_adc_interrupt,
                      IRQF_TRIGGER_FALLING, id->name, st);
if (ret) {
    dev_err(&spi->dev, "failed to request interrupt %d (%d)", st->irq, ret);
    return ret;
}

ret = devm_iio_device_register(&spi->dev, indio_dev);
if (ret < 0)
    return ret;
```

```
    st->conversion_done = false;

    return 0;
}

static int ltc2422_remove(struct spi_device *spi)
{
    dev_info(&spi->dev, "my_remove() function is called.\n");
    return 0;
}

static const struct of_device_id ltc2422_dt_ids[] = {
    { .compatible = "arrow,ltc2422", },
    { }
};
MODULE_DEVICE_TABLE(of, ltc2422_dt_ids);

static const struct spi_device_id ltc2422_id[] = {
    { .name = "ltc2422", },
    { }
};
MODULE_DEVICE_TABLE(spi, ltc2422_id);

static struct spi_driver ltc2422_driver = {
    .driver = {
        .name    = "ltc2422",
        .owner   = THIS_MODULE,
        .of_match_table = ltc2422_dt_ids,
    },
    .probe     = ltc2422_probe,
    .remove    = ltc2422_remove,
    .id_table  = ltc2422_id,
};

module_spi_driver(ltc2422_driver);

MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("LTC2422 DUAL ADC with triggering");
MODULE_LICENSE("GPL");
```

ltc2422_imx_trigger.ko with LTC2422_app Demonstration

```
root@imx7dsabresd:~# insmod ltc2607_imx_dual_device.ko /* load the ltc2607 module */
root@imx7dsabresd:~# insmod ltc2422_imx_trigger.ko /* load the ltc2422 module */
root@imx7dsabresd:/sys/bus/iio/devices/iio:device3# echo 65535 > out_voltage2_raw
/* set both DAC outputs to 5V */
root@imx7dsabresd:~# ./LTC2422_app /* Launch LTC2422_app that calls your ltcC2422_imx_trigger.ko driver and shows the readed analog values. Press the FUNC2 button three times to read the ADC spi digital output discarding the first readed value, then the DAC output analog values are displayed before exiting the application */
ltc2422 spi2.1: Press FUNC2 key to start conversion
ltc2422 spi2.1: Press FUNC2 key to start conversion
the value of ADC channel 0
    is : 4.9954
ltc2422 spi2.1: Press FUNC2 key to start conversion
the value of ADC channel 1
    is : 4.9931
Application terminated

root@imx7dsabresd:~# rmmod ltc2607_imx_dual_device.ko
root@imx7dsabresd:~# rmmod ltc2422_imx_trigger.ko
```

12

Using the Regmap API in Linux Device Drivers

As you have seen throughout 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 to entire Linux through the regmap API.

So far, you have developed several I2C and SPI device drivers 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.

A device that can be accessed using the SPI or the I2C buses, as the ADXL345 accelerometer, is a good candidate to use the regmap API to read and write to their respective buses. For this ADXL345 device you can develop two simple drivers using the regmap API, one for the I2C bus support (adx345-i2c.c) and another for the SPI bus support (adx345-spi.c), writing specific code for each bus. This specific code includes the configuration for the register map by a struct regmap_config structure and the initialization of the register map using the next functions:

The following function initialises the regmap data structures based on the SPI configuration:

```
struct regmap * devm_regmap_init_spi(struct spi_device *spi,  
                                     const struct regmap_config);
```

The following function initialises regmap data structures based on the I2C configuration:

```
struct regmap * devm_regmap_init_i2c(struct i2c_client *i2c,  
                                     const struct regmap_config);
```

In the two previous regmap initialisation 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. For example, in the case of the SPI bus, the regmap read and write function pointers will point to the SPI read and write functions.

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 have implemented the regmap configuration and initialization using two specific SPI/I2C drivers you will 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);
```

Implementing Regmap

The regmap infrastructure provides two important data structures defined in `include/linux/regmap.h` to implement Linux regmap; these are `regmap_config` and `regmap` structures.

The `regmap_config` structure is a per device configuration structure used by the regmap subsystem to talk to the device. It is defined by driver code, and contains all the information related to the registers of the device. Descriptions of its important fields are listed 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.
- **lock:** This is an optional callback function written in driver code, which is called before starting any read or write operation. The function should take a lock and return it.
- **unlock:** This is an optional callback function written in driver code for unlocking the lock, which is created by the lock callback function.
- **fast_io:** regmap internally uses mutex to lock and unlock, if a custom lock and unlock mechanism is not provided. If the driver wants regmap to use the spinlock, then fast_io should be set to "true"; else, regmap will use the mutex based lock.
- **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 defined in include/linux/regmap.h and implemented under drivers/base/regmap/. The following are the details of the regmap_write and the regmap_read APIs:

1. **regmap_write:** This function is used to write data to the device. It takes in the regmap structure returned during initialisation, 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, then it will check if the register address passed is less than max_register. If it is less than max_register, then only the write operation is performed; else, -EIO (invalid I/O) is returned.
 - After that, if the writeable_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 writeable_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 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 initialisation, 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" being the read operation 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 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 last lab of the book, you will develop a driver with similar functionality to the lab 10.2 one, but this time you will use the IIO framework instead of the input framework to develop it. You will also access to the registers of the ADXL345 device using the regmap API instead of the SPI specific core APIs.

As in the lab 10.2 driver, this new driver will support single tap motion detection 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 for 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 pag 28 of the ADXL345 data-sheet). The tap motion detection will be exposed to user space sending an IIO event by the iio_push_event() function, which is called within the ISR 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 three axis values (plus a timestamp value) captured by an IIO trigger (iio-trig-hrtimer or a iio-trig-sysfs trigger) in each of the IIO buffer entries.

You will keep the same HW and DT configuration as in previous lab 10.2 for all the processor variants.

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 #defines to perform operations in the ADXL345 registers and pass 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; // creo que no es necesario
    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 the 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 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 its 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,
    .channel12 = 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(DATAX0, X, 0),
    ADXL345_CHANNEL(DATAY0, Y, 1),
    ADXL345_CHANNEL(DATAZ0, Z, 2),
    IIO_CHAN_SOFT_TIMESTAMP(3),
};
```

6. Create the struct iio_info structure to declare the hooks the IIO core will use for this device. There are four kernel 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 each axis value when user space access to each channel sysfs attribute with bitmask IIO_CHAN_INFO_RAW. It also returns the accelerometer scale when user space reads the shared sysfs attribute with bitmask IIO_CHAN_INFO_SCALE. See in the code below the regmap regmap_bulk_read() function 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. See in the code below the regmap 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 and DUR registers, whenever each axis sysfs attributes with bitmask IIO_EV_INFO_VALUE and IIO_EV_INFO_PERIOD are 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 and DUR registers, whenever each axis sysfs attributes with bitmask IIO_EV_INFO_VALUE and IIO_EV_INFO_PERIOD are 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 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. A threaded interrupt will be added to the driver to service the single tap interrupt. In a threaded interrupt, the interrupt handler adxl345_event_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 this driver, you are going to communicate via SPI with the ADXL345 device inside the interrupt handler using the regmap regmap_read() function. The SINGLE_TAP events will be sent to user space 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 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 pollfunc bottom half adxl345_trigger_handler runs in the context of a

kernel thread and all the processing takes place here. It reads the tree axis values from the ADXL345 device and stores them in the internal buffer (together with the timestamp obtained in the top half) 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 recording of the current timestamp and for this reason you can use the IIO core `iio_pollfunc_store_time()` function. Before calling `devm_iio_triggered_buffer_setup()` the struct `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 = DATAX0;
    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 struct spi_device_id structures:

```
static const struct spi_device_id adxl345_id[] = {
    { .name = "adxl345", },
    { }
};
MODULE_DEVICE_TABLE(spi, adxl345_id);
```

12. Add a struct 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. Build the modified device tree, and load it to the target processor.

See in the next **Listing 12-1** the "SPI regmap IIO device" driver source code (`adxl345_imx_iio.c`) for the i.MX7D processor.

Note: The source code for the SAMA5D2 (`adxl345_sam_iio.c`) and BCM2837 (`adxl345_rpi_iio.c`) drivers can be downloaded from the GitHub repository of this book.

Listing 12-1: `adxl345_imx_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 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 */

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(DATAX0, X, 0),
    ADXL345_CHANNEL(DATAY0, Y, 1),
    ADXL345_CHANNEL(DATAZ0, 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, later 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
 * 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);
    }
}
```

```
        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[3];
    s16 buf[8] /* 16 bytes */
    int i, ret, j = 0, base = DATAX0;
    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;
    }

    /* Set the device ID register to the correct value */
    regmap_update_bits(regmap, DEVID, 0x03, 0x02);
}
```

```
}

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, GPIO_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 = &adx1345_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, less or higher noise reducing 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, &adx1345_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 = "adx1345", },
    { }
};
MODULE_DEVICE_TABLE(spi, adxl345_id);

static const struct of_device_id adxl345_dt_ids[] = {
    { .compatible = "arrow,adx1345" },
    { },
};
MODULE_DEVICE_TABLE(of, adxl345_dt_ids);

static struct spi_driver adxl345_driver = {
    .driver = {
        .name    = "adx1345",
        .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_imx_iio.ko Demonstration

"In the Host build the IIO tools. Edit the Makefile under my-linux-imx/tools/iio/ folder"

```
~/my-linux-imx/tools/iio$ gedit Makefile /* Comment out the first line and modify second line */

//CC = $(CROSS_COMPILE)gcc
CFLAGS += -Wall -g -D_GNU_SOURCE -I$(INSTALL_HDR_PATH)/include

~/my-linux-imx$ source /opt/fsl-imx-x11/4.9.11-1.0.0/environment-setup-cortexa7hf-neon-poky-linux-gnueabi

~/my-linux-imx$ make headers_install INSTALL_HDR_PATH=~/my-linux-sam/headers/
~/my-linux-imx$ make -C tools/iio/ INSTALL_HDR_PATH=~/my-linux-sam/headers/ /* Build IIO tools */
~/my-linux-sam/tools/iio$ scp iio_generic_buffer root@10.0.0.10: /* send iio_generic_buffer application to the target */
~/my-linux-sam/tools/iio$ scp iio_event_monitor root@10.0.0.10: /* send iio_event_monitor application to the target */

"Boot now the i.MX7D device"

root@imx7dsabresd:~# insmod adxl345_imx_iio.ko /* load module */
root@imx7dsabresd:~# cd /sys/bus/iio/devices/iio:device2/
root@imx7dsabresd:/sys/bus/iio/devices/iio:device2# ls /* see the sysfs entries under the iio device */
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

root@imx7dsabresd:/sys/bus/iio/devices/iio:device2# cat name /* read the device name */
*/
adxl345

root@imx7dsabresd:/sys/bus/iio/devices/iio:device2# cat in_accel_z_raw /* read the z axis value */
245

root@imx7dsabresd:/sys/bus/iio/devices/iio:device2# cat in_accel_z_raw /* move the accel board and read again the z axis value */
-252

root@imx7dsabresd:/sys/bus/iio/devices/iio:device2# /* move the accel board until several interrupts are being generated */
adxl345 spi2.1: single tap interrupt has occurred
adxl345 spi2.1: single tap interrupt has occurred
```

```
adxl345 spi2.1: single tap interrupt has occurred
adxl345 spi2.1: single tap interrupt has occurred

root@imx7dsabresd:/sys/bus/iio/devices# cat in_accel_scale /* read the
accelerometer scale */
0.038300

"Sysfs trigger interface"

root@imx7dsabresd:/sys/bus/iio/devices# ls /* see the created iio_sysfs_trigger
folder */
iio:device0 iio:device1 iio:device2 iio_sysfs_trigger

root@imx7dsabresd:/sys/bus/iio/devices# echo 1 > iio_sysfs_trigger/add_trigger /*
create a sysfs trigger */
root@imx7dsabresd:/sys/bus/iio/devices# ls /* see the trigger0 folder created */
iio:device0 iio:device1 iio:device2 iio_sysfs_trigger trigger0

root@imx7dsabresd:/sys/bus/iio/devices# cat trigger0/name > iio:device2/trigger/
current_trigger /* attach the trigger to the iio device */

root@imx7dsabresd:/sys/bus/iio/devices# echo 1 > iio:device2/scan_elements/in_ac
cel_x_en /* enable x axis scan element */

root@imx7dsabresd:/sys/bus/iio/devices# echo 1 > iio:device2/scan_elements/in_ac
cel_y_en /* enable y axis scan element */

root@imx7dsabresd:/sys/bus/iio/devices# echo 1 > iio:device2/scan_elements/in_ac
cel_z_en /* enable z axis scan element */

root@imx7dsabresd:/sys/bus/iio/devices# echo 1 > iio:device2/scan_elements/in_ti
mestamp_en /* enable timestamp scan element */

root@imx7dsabresd:/sys/bus/iio/devices# echo 100 > iio:device2/buffer/length /* set
the number of sample sets that may be held by the buffer */
root@imx7dsabresd:/sys/bus/iio/devices# echo 1 > iio:device2/buffer/enable /* enable
the buffer */
root@imx7dsabresd:/sys/bus/iio/devices# echo 1 > trigger0/trigger_now /* do first
adquisition */
root@imx7dsabresd:/sys/bus/iio/devices# echo 1 > trigger0/trigger_now /* do second
adquisition */
root@imx7dsabresd:/sys/bus/iio/devices# hexdump -v -e '16/1 "%02x" "\n"' < /dev
/iio\:device2 /* show the adquired values: the tree axis + timestamp */

ff ff f3 ff f0 00 0c a8 47 52 1e 21 b6 ec f6 14
ff ff f1 ff f4 00 0c a8 10 4f 91 93 ba ec f6 14

root@imx7dsabresd:/sys/bus/iio/devices# echo 0 > iio:device2/buffer/enable /*
disable the buffer */
root@imx7dsabresd:/sys/bus/iio/devices# echo "" > iio:device2/trigger/current_
trigger /* detach the trigger */

root@imx7dsabresd:~# rmmod adxl345_imx_iio.ko /* remove the module */

"Reboot again your target system. Use the iio_generic_buffer application to set
```

the buffer length and number of acquisitions, enable the scan elements and show the acquired values"

```
root@imx7dsabresd:~# insmod adxl345_imx_iio.ko /* load module */
/* create the sysfs trigger */
root@imx7dsabresd:~# echo 1 > /sys/bus/iio/devices/iio_sysfs_trigger/add_trigger
root@imx7dsabresd:~# echo sysfstrig1 > /sys/bus/iio/devices/iio:device2/trigger/current_trigger /* attach the trigger to the device */
root@imx7dsabresd:~# ./iio_generic_buffer --device-num 2 -T 0 -a -l 10 -c 5 & /* launch the iio_generic_buffer application */
[1] 600
root@imx7dsabresd:~# iio device number being used is 2
iio trigger number being used is 0
No channels are enabled, 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:device2/sysfstrig1
root@imx7dsabresd:~# echo 1 > /sys/bus/iio/devices/trigger0/trigger_now /* do first acquisition, you can do until 5 conversions */
-0.038300 -0.536200 9.268600 1510654204332659500
root@imx7dsabresd:~# echo 1 > /sys/bus/iio/devices/trigger0/trigger_now /* do second acquisition */
0.229800 0.153200 -9.919700 1510654243352620000
root@imx7dsabresd:/sys/bus/iio/devices# echo 0 > iio:device2/buffer/enable /* disable the buffer */
root@imx7dsabresd:/sys/bus/iio/devices# echo "" > iio:device2/trigger/current_trigger /* detach the trigger */
root@imx7dsabresd:~# rmmod adxl345_imx_iio.ko /* remove the module */
"Capture now data using the hrtimer trigger. Reboot your target system"
root@imx7dsabresd:~# insmod adxl345_imx_iio.ko /* load the module */
root@imx7dsabresd:~# mkdir /config /* create the config folder */
root@imx7dsabresd:~# mount -t configfs none /config /* mount the configfs file system */
root@imx7dsabresd:~# mkdir /config/iio/triggers/hrtimer/trigger0 /* create the hrtimer trigger */
root@imx7dsabresd:/sys/bus/iio/devices/trigger0# echo 50 > sampling_frequency /* set the sampling frequency */
root@imx7dsabresd:~# echo trigger0 > /sys/bus/iio/devices/iio:device2/trigger/current_trigger /* attach the trigger to the device */
root@imx7dsabresd:~# ./iio_generic_buffer --device-num 2 -T 0 -a -l 10 -c 10 & /* use the iio_generic_buffer application to set the buffer length and number of conversions; the application enable the scan elements, do the acquisitions and show them; after that, disable the scan elements */
```

```
iio device number being used is 2
iio trigger number being used is 0
No channels are enabled, 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:device2 trigger0
-0.574500 -0.612800 9.306900 1510654381725464500
-0.612800 -0.536200 9.268600 1510654381745453000
-0.612800 -0.536200 9.268600 1510654381765445875
-0.612800 -0.536200 9.230300 1510654381785415750
-0.612800 -0.536200 9.230300 1510654381805414625
-0.612800 -0.497900 9.268600 1510654381825415000
-0.612800 -0.497900 9.268600 1510654381845414750
-0.612800 -0.536200 9.268600 1510654381865414500
-0.612800 -0.536200 9.268600 1510654381885414625
-0.574500 -0.574500 9.345200 1510654381905413000
Disabling: in_accel_y_en
Disabling: in_accel_x_en
Disabling: in_timestamp_en
Disabling: in_accel_z_en

root@imx7dsabresd:~# rmdir /config/iio/triggers/hrtimer/trigger0 /* remove the
trigger */

"Now launch the iio_event_monitor application and move the accel board until you see
several IIO events"

root@imx7dsabresd:~# ./iio_event_monitor /dev/iio\:device2
adxl345 spi2.1: single tap interrupt has occurred
Event: time: 1510654097324288875,adxl345 spi2.1: single tap interrupt has occurred
type: accel(z), channel: 0, evtype: thresh, direction: either
Event: time: 1510654097329211750, type: accel(z), channel: 0, evtype: thresh, direction: either
adxl345 spi2.1: single tap interrupt has occurred
Event: time: 1510654107219470250, type: accel(z), channel: 0, evtype: thresh, direction: either
adxl345 spi2.1: single tap interrupt has occurred
Event: time: 1510654107238069500,adxl345 spi2.1: single tap interrupt has occurred
type: accel(z), channel: 0, evtype: thresh, direction: either
Event: time: 1510654107242770125, type: accel(z), channel: 0, evtype: thresh, direction: either

root@imx7dsabresd:~# rmmod adxl345_imx_iio.ko /* remove the module */
```

References

1. NXP, "i.MX 7Dual Applications Processor Reference Manual". Document Number: IMX7DRM Rev. 0.1, 08/2016.
2. Microchip, "SAMA5D2 Series Datasheet". Datasheet number: DS60001476B.
3. Broadcom, "BCM2835 ARM Peripherals" guide.
4. NXP, "i.MX Yocto Project User's Guide". Document Number: IMXLXYOCTOUG. Rev. L4.9.11_1.0.0-ga+mx8-alpha, 09/2017.
5. Marcin Bis, "Exploring Linux Kernel Source Code with Eclipse and QTCreator". ELCE 2016, Berlin, 10/2016.
https://bis-linux.com/en/elc_europe2016
6. Raspberry Pi Linux Documentation.
<https://www.raspberrypi.org/documentation/linux/>
7. Linux & Open Source related information for AT91 Smart ARM Microcontrollers.
<http://www.at91.com/linux4sam/bin/view/Linux4SAM>
8. The Linux kernel DMAEngine Documentation.
<https://www.kernel.org/doc/html/v4.15/driver-api/dmaengine/index.html>
9. The Linux kernel Input Documentation.
<https://www.kernel.org/doc/html/v4.12/input/input.html>
10. The Linux kernel PINCTRL (PIN CONTROL) subsystem Documentation.
<https://www.kernel.org/doc/html/v4.15/driver-api/pinctl.html>
11. The Linux kernel General Purpose Input/Output (GPIO) Documentation.
<https://www.kernel.org/doc/html/v4.17/driver-api/gpio/index.html>
12. Hans-Jürgen Koch, "The Userspace I/O HOWTO".
<http://www.hep.by/gnu/kernel/uio-howto/>
13. Daniel Baluta, "Industrial I/O driver developer's guide".
<https://dbaluta.github.io/>
14. Rubini, Corbet, and Kroah-Hartman, "Linux Device Drivers", third edition, O'Reilly, 02/2005.
<https://lwn.net/Kernel/LDD3/>
15. bootlin, "Linux Kernel and Driver Development Training".
<https://bootlin.com/doc/training/linux-kernel/>

References

16. Corbet, LWN.net, Kroah-Hartman, The Linux Foundation, "Linux Kernel Development report", seventh edition, 08/2016.

<https://www.linux.com/publications/linux-kernel-development-how-fast-it-going-who-doing-it-what-they-are-doing-and-who-5/>

Index

A

`alloc_chrdev_region()` function 94-95

B

`binding`

 matching, device and driver 15, 76, 117, 211, 438

`Bootloader` 20-21

`Broadcom BCM2837 processor` 50

`bus_register()` function 74

`bus_type` structure 74

C

`C compiler` 37, 38, 54

`C runtime library`

 about 25

`glibc` 26

`cdev` structure 94

`cdev_add()` function 95

`cdev_init()` function 95

`character device`

 about 91

`devtmpfs`, creation 104

 identification 91

 major and minor numbers 93

 misc framework, creation 109-111

`character device driver`

 about 92

`alloc_chrdev_region()` function 94-95

`cdev_add()` function 95

`cdev_init()` function 95

`copy_from_user()` function 93

`copy_to_user()` function 93

`file_operations` structure 92

`MAKEDEV` script 94-96

`registering devices` 94-95

`register_chrdev_region()` function 94, 95

`unregister_chrdev_region()` function 94

`class, Linux`

 about 174

`class_create()` function 105

`class_destroy()` function 105

`device_create()` function 105

`device_destroy()` function 105

`LED class` 174

`copy_from_user()` function 93, 145

`copy_to_user()` function 93, 145

`create_singlethread_workqueue()` function 295-296

`create_workqueue()` function 295-296

D

`DECLARE_WAIT_QUEUE_HEAD()` function 299

`DECLARE_WORK()` function 294

`deferred work`

 about 283-284

 bottom-half, about 284

 softirqs 283, 284-285

 tasklets 283, 286

 threaded interrupts 283, 290-292

 timers 283, 286-287

 top-half, about 284

 workqueues 283, 292-296

delayed_work structure 294
del_timer() function 287
del_timer_sync() function 287
destroy_workqueue() function 296
device node 91
device tree
 about 78
 building, on BCM2837 processor 53
 building, on i.MX7D processor 40
 building, on SAMA5D2 processor 48
 chosen node 80
 compatible property, about 79
 loading 29
 location 78
 machine_desc structure 79
 of_platform_populate() function 80-81
 of_scan_flat_dt() function 80
 setup_machine_fdt() function 79, 80
 unflatten_device_tree() function 80
device tree, bindings
 about 78
 gpios 143-144
 I2C controller and devices 212-213
 interrupt, connections 268-271
 interrupt, controller 268
 interrupt, device nodes 269
 pin controller 134-138
 resource properties 172
 spi controller 443
 spi devices 444
device structure 73
device_create() function 105
device_destroy() function 105
device_driver structure 73
device_for_each_child_node() function 243, 326
device_get_child_node_count() function 241
device_register() function 75-76
devm_get_gpiod_from_child() function 326
devm_gpiochip_add_data() function 139
devm_gpiod_get() function 141
devm_gpiod_get_index() function 141
devm_gpiod_put() function 141
devm_iio_device_alloc() function 480
devm_iio_device_register() function 481
devm_iio_triggered_buffer_setup() function 488
devm_input_allocate_polled_device() function 428
devm_ioremap() function 146
devm_iounmap() function 146
devm_kmalloc() function 362
devm_kzalloc() function 362
devm_led_classdev_register() function 174
devm_pinctrl_register() function 130
devm_regmap_init_i2c() function 541
devm_regmap_init_spi() function 541
devm_request_irq() function 272-273
devm_request_threaded_irq() function 290-291
devm_spi_register_master() function 438, 442
Direct Memory Access (DMA)
 about 373
 cache coherency 373-374
 dma_map_ops structure 374
DMA engine Linux API
 about 375
 bus address 377
 Contiguous Memory Allocator (CMA) 377-378
 Descriptor, for transaction 375-376
 dmaengine_prep_slave_sg() function 376
 dmaengine_slave_config() function 375
 dma_async_issue_pending() function 376
 dma_request_chan() function 375
 dma_slave_config structure 375
DMA from user space
 about 407-409
 mmap() function 407-409

remap_pfn_range() function 409
DMA mapping, coherent
 about 378-380
 dma_alloc_coherent() function 378-380
 dma_free_coherent() function 380
DMA mapping, scather/gather
 about 396-397
 dma_map_sg() function 397
 dma_unmap_sg() function 397
 scatterlist structure 396
 sg_init_table() function 396
 sg_set_buf() function 396
DMA mapping, streaming
 about 380-382
 dma_map_single() function 380, 381
 dma_unmap_single() function 380, 382
dmaengine_prep_slave_sg() function 376
dmaengine_slave_config() function 375
dmaengine_submit() function 376
dma_alloc_coherent() function 378-380
dma_async_issue_pending() function 376
dma_free_coherent() function 380
dma_map_ops structures 374
dma_map_sg() function 397
dma_map_single() function 380, 381
dma_request_chan() function 375
dma_unmap_sg() function 397
dma_unmap_single() function 380, 382
documentation, for processors 123
driver_register() function 77

E

Eclipse
 about 55
 configuring, for kernel sources 56-66
 configuring, for Linux drivers 67-72
 installing 56
 URL, for installing 56

environment variables
 U-Boot, on i.MX7D processor 42-43
 U-Boot, on SAMA5D2 processor 50
 Yocto SDK 36-37
ethernet, setting up 31-32
exit() function 83

F

file_operations structure 92
flush_scheduled_work() function 294, 295
flush_workqueue() function 296
for_each_child_of_node() function 177, 181
free_irq() function 272
fsl,pins, device tree 136-137
fwnode_handle structure 241

G

get_user() function 144
glibc 26
GPIO controller driver
 about 138-140
 gpio_chip structure 138-139
GPIO interface, descriptor based
 devm_get_gpiod_from_child() function 326
 devm_gpiod_get() function 141
 devm_gpiod_get_index() function 141
 devm_gpiod_put() function 141
 gpiod_direction_input() function 141
 gpiod_direction_output() function 141
 gpiod_to_irq() function 143
 mapping, GPIOs to IRQs 143
 obtaining GPIOs 141
 using GPIOs 141-142
GPIO interface, integer based 140
GPIO irqchips, categories 139-140, 264-268
GPIO linux number 127
 gpiod_direction_input() function 141

gpiod_direction_output() function 141
gpiod_to_irq() function 143
gpio_chip.to_irq() function 262

H

hardware irq's (hwirq) 257

I

I2C, definition 205
I2C device driver
 i2c_add_driver() function 210
 i2c_device_id structure 211
 i2c_driver structure 210
 registering 210

I2C subsystem, Linux
 I2C bus core 206
 I2C controller drivers 206-207
 I2C device drivers 208-209
 registering, I2C controller devices 209
 registering, I2C controller drivers 207

i2cdetect, application 433, 516
i2c_adapter structure 212
i2c_add_driver() function 210
i2c_del_driver() function 210
i2c_set_clientdata() function 222
i2c-utils, application 433, 516

IIO buffers
 about 485
 iio_chan_spec structure 485-486
 iio_push_to_buffers_with_timestamp() function 487
 setup 485-487
 sysfs interface 485
IIO device, channels
 about 481
 iio_chan_spec structure 481-482
 sysfs attributes 482-484

IIO device, sysfs interface 481
IIO driver
 devm_iio_device_alloc() function 480
 devm_iio_device_register() function 481
 registration 480-481
IIO events
 about 48
 iio_event_spec structure 490
 iio_push_event() function 492
 kernel hooks 491
 sysfs attributes 490
IIO triggered buffers
 about 487-488
 devm_iio_triggered_buffer_setup() function 488
 iio_triggered_buffer_cleanup() function 488
 iio_triggered_buffer_setup() function 487
IIO utils 494
iio_info structure
 about 484
 kernel hooks 484
 iio_priv() function 499, 502, 521
 iio_push_event() function 492
 iio_push_to_buffers_with_timestamp() function 487
 iio_triggered_buffer_cleanup() function 488
 iio_triggered_buffer_setup() function 487
Industrial I/O framework (IIO), about 479
init process 30
init programs 30
init() function 83
init_waitqueue_head() function 299
INIT_WORK() function 294
Input subsystem framework
 about 419
 drivers 420-421
 evtest, application 421
 input_event() function 424
 input_polled_dev structure 424

input_register_polled_device() function 424
input_sync() function 424
input_unregister_polled_device() function 424
set_bit() function 424
interrupt context, kernel 283
interrupt controllers 257
interrupt handler
 about 272
 function, parameters 272
 function, return values 273
interrupt links, device tree 269-270
interrupts, device tree 269
interrupt-cells, device tree 268
interrupt-controller, device tree 268
interrupt-parent, device tree 269
ioremap() function 146
IRQ domain 260
IRQ number 257
irq_chip structure 257-258
irq_create_mapping() function 261-262
irq_data structure 259-260
irq_desc structure 258-259
irq_domain structure 260-261
irq_domain_add_*() functions 261
irq_find_mapping() function 262, 268
irq_set_chip_and_handler() function 263

J

jiffies 287

K

kernel memory, allocators
 kmalloc allocator 362
 PAGE allocator 357-358
 PAGE allocator API 358
 SLAB allocator 358-360
 SLAB allocator API 361-362

kernel modules
 building and installing, on BCM2837
 processor 53
 building and installing, on i.MX7D
 processor 42
 building and installing, on SAMA5D2
 processor 49
kernel object (kobject)
 attributes 231-233
 structures 230
kernel physical memory
 memory-mapped I/O 356
 ZONE_DMA 356
 ZONE_HIGMEM 356
 ZONE_NORMAL 356
kernel threads
 definition 312
 kthread_create() function 312-313
 kthread_run() function 313
 kthread_stop() function 313
 wake_up_process() function 313
kfree() function 362
kmalloc allocator
 about 362
 devm_kmalloc() function 362
 devm_kzalloc() function 362
 kfree() function 362
 kmalloc() function 362
 kzalloc() function 362
 kthread_create() function 312-313
 kthread_run() function 313
 kthread_stop() function 313

L

led_classdev structure 174-176
Linux, address types
 bus addresses 353
 kernel logical addresses 354

- kernel virtual addresses 354
- physical addresses 353
- user virtual addresses 353
- Linux, boot process
 - DDR initialization 28
 - main stages 28-30
 - on-chip boot ROM 28-29
 - start_kernel() function 29-30
- Linux, device and driver model
 - about 73
 - binding 76
 - bus controller drivers 76
 - bus core drivers 74-76
 - bus_register() function 74
 - bus_type structure 74
 - data structures 73
 - device drivers 77
 - device_driver structure 77
 - device_register() function 75, 76
 - driver_register() function 76, 77
 - probe() function 76, 77
- Linux embedded
 - about 19
 - building, cross toolchain 30
 - building, methods 31
 - building, on BCM2837 processor 50-51
 - building, on i.MX7D processor 32-35
 - building, on SAMA5D2 processor 43-46
 - main components 19
- Linux kernel
 - about 22
 - building, on BCM2837 processor 51-52
 - building, on i.MX7D processor 39-40
 - building, on SAMA5D2 processor 47-48
 - distribution kernels 25
 - licensing 84
 - longterm 22, 24
 - mainline 22, 23
 - prepatch 23
- stable 22, 23
- subsystems 23
- Linux kernel modules
 - about 83
- Linux LED class
 - about 174
 - devm_led_classdev_register() function 174
 - registering 174
 - list_head devres_head structure 146
- locking, kernel
 - about 296-297
 - mutex 297
 - spinlock 297

M

- menuconfig 39, 47, 51, 103, 104, 192, 423
- Microchip SAMA5D2 processor 43
- mikroBUS™ standard 148
- miscellaneous devices 109, 110
- miscdevice structure 110
- misc_deregister() function 110
- misc_register() function 110
- mmap() function 407-409
- MMIO (Memory-Mapped I/O)
 - about 145
 - devm_ioremap() function 146
 - ioremap() function 146
 - reading/writing, interfaces 146
- MMU (Memory Management Unit) 345-346
- MMU translation tables 346-352
- mm_struct structure 351
- module_exit() function 83
- module_init() function 83
- MODULE_LICENSE macro 84
- module_platform_driver() function 117
- module_spi_driver() function 441
- multiplexing, pin 123
- mutex, kernel lock 297

N

net name, schematic 124
Network File System (NFS) server
installing, on i.MX7D processor 42
installing, on SAMA5D2 processor 49
NXP i.MX7D processor 32

O

of_device_id structure 117
of_match_table
I2C device driver 211
platform device driver 115
SPI device driver 441
of_platform_populate() function 81
of_property_read_string() function 163
of_scan_flat_dt() function 80
open() system call 92

P

pad
definition 123
gpio function 126
iomux mode, on BCM2837 processor 151
iomux mode, on i.MX7D processor 126
iomux mode, on SAMA5D2 processor 150
logical/canonical name 123
PAGE allocator, kernel memory 357-358
PAGE allocator API, kernel memory 358
Page Global Directory (PGD), MMU 351
Page Middle Directory (PMD), MMU 352
Page Table (PTE), MMU 352
Page Upper Directory (PUD), MMU 352
pin configuration node, device tree 135, 136
pin control subsystem, about 127-134
pin controller, about 124-127
pin, definition 123

pinctrl-0, device tree 135
pinctrl_desc structure 130
platform bus, about 115
platform device driver
about 115
module_platform_driver() function 117
platform_driver structure 115-116
platform_driver_register() function 116
platform_get_irq() function 174, 278
platform_get_resource() function 173
platform_set_drvdata() function 163
probe() function 115, 116
registering 116, 117
resources, structures and APIs 172-174
resource_size() function 173
platform devices
about 115
registering 81
platform_driver_register() function 116
platform_get_drvdata() function 164
platform_get_irq() function 174, 278
platform_get_resource() function 173
platform_set_drvdata() function 163
probe() function 116
process context, kernel 283
put_user() function 144

Q

queue_work() function 296

R

race, condition 296
Raspbian
about 50-51
installing, on BCM2837 processor 51
read() system call 92, 93
register_chrdev_region() function 94, 95

Regmap

about 541
devm_regmap_init_i2c() function 541
devm_regmap_init_spi() function 541
regmap structure 541
regmap_config structure 542

Regmap, implementation

APIs 544-546
regmap_config structure 543-544
regmap_bulk_read() function 550
remap_pfn_range() function 409
repo, utility 33
request_irq() function 272
resource structure 172
resource_size() function 173
root filesystem 27-28

S

schedule_delayed_work_on() function 295
schedule_on_each_cpu() function 295
schedule_work() function 294
setup_arch() function 79
setup_machine_fdt() function 79, 80
set_bit() function 424
sg_init_table() function 396
sg_set_buf() function 396
SLAB allocator, kernel memory 358-360
SLAB allocator API, kernel memory 361-362
sleeping, kernel
about 298-299
DECLARE_WAIT_QUEUE_HEAD() function 299
init_waitqueue_head() function 299
wait queue 299
wait_event() function 299
wait_event_interruptible() function 299
wait_queue_head_t structure 299
wake_up() function 299-300
SMBus, definition 205

softirqs, deferred work 283, 284-285

SPI, about 435

SPI device drivers

about 440
registration 440-441
spi_driver structure 440-441

SPI, Linux

controller drivers 435
protocol drivers 436
spi_async() function 436
spi_read() function 436
spi_sync() function 436
spi_write() function 436
spi_write_then_read() function 436, 438, 440

SPI, Linux subsystem

SPI bus core drivers 437
SPI controller drivers 438-440
SPI device drivers 440

spidev driver 509

spinlock, kernel lock 297

spi_w8r16() function 437

spi_write_then_read() function 436, 438, 440

start_kernel() function 29-30

sysfs filesystem

about 87, 229-233

system call, interface

about 25

open() system call 92

read() system call 92, 93

write() system call 92, 93

system shared libraries

about 26

locations 27

T

tasklets, deferred work 283, 286

task_struct structure 351

threaded interrupts, deferred work 283, 290-292
timers, deferred work 283, 286-287
timer_list structure 286
timeval structure 88-89
toolchain
 about 30
 setting up, on BCM2837 processor 51
 setting up, on i.MX7D processor 36-37
 setting up, on SAMA5D2 processor 46-47
Translation Lookaside Buffers (TLBs), MMU 346
Translation Table Base Control Register (TTBCR), MMU 348
Translation Table Base Registers (TTRB0 and TTRB1), MMU 348
Trivial File Transfer Protocol (TFTP) server
 installing, on i.MX7D processor 41
 installing, on SAMA5D2 processor 49

U

UIO driver 190-192
UIO framework
 APIs 193-194
 definition 190
 working 192-193
UIO platform device driver 191
uio_register_device() function 194
unflatten_device_tree() function 80
Unified Device Properties, API
 about 240
 functions 241
unregister_chrdev_region() function 94
user space, drivers
 advantages 189
 disadvantages 189
U-Boot
 about 20

main features 21
setting up environment variables, on i.MX7D processor 42-43
setting up environment variables, on SAMA5D2 processor 50

V

virtual file, about 91
virtual interrupt ID 257
virtual memory layout, user space process
 data segment 354
 memory mapping segment 354-355
 stack segment 355
 text segment 354
virtual to physical, memory mapping, kernel 355-356

W

wait queue, kernel sleeping 299
wait_event() function 299
wait_event_interruptible() function 299
wake_up() function 299-300
workqueues, deferred work
 about 283, 292-296
 create_singlethread_workqueue() function 295-296
 create_workqueue() function 295-296
 DECLARE_WORK() function 294
 destroy_workqueue() function 296
 flush_scheduled_work() function 294, 295
 flush_workqueue() function 296
 INIT_WORK() function 294
 queue_work() function 296
 schedule_delayed_work_on() function 295
 schedule_on_each_cpu() function 295
 schedule_work() function 294
 work item 292

Index

work types 293-294

worker 292

workqueue_struct structure 295

work_struct structure 294

write() system call 92, 93

Y

Yocto

about 31

host packages 33, 44

setting up, on i.MX7D processor 33-35

setting up, on SAMA5D2 processor 44-46

Yocto Project SDK

about 36

setting up, on i.MX7D processor 36-38

setting up, on SAMA5D2 processor 46-47

<http://www.rejoiceblog.com/>

<http://www.rejoiceblog.com/>