Linux Driver Development for Embedded Processors

Raspberry Pi 4 Practical Labs

Building a Linux embedded system for the Raspberry Pi 4 Model B

Raspberry Pi 4 Model B is the latest product in the popular Raspberry Pi range of computers. It offers ground-breaking increases in processor speed, multimedia performance, memory, and connectivity compared to the prior-generation Raspberry Pi 3 Model B+, while retaining backwards compatibility and similar power consumption. For the end user, Raspberry Pi 4 Model B provides desktop performance comparable to entry-level x86 PC systems.

This product's key features include the high-performance Broadcom BCM2711, 64-bit quad-core Cortex-A72 (ARM v8) processor, dual-display support at resolutions up to 4K via a pair of micro-HDMI ports, hardware video decode at up to 4Kp60, up to 4GB of RAM, dual-band 2.4/5.0 GHz wireless LAN, Bluetooth 5.0, Gigabit Ethernet, USB 3.0, and PoE capability (via a separate PoE HAT add-on).

You can find Raspberry Pi 4 Tech Specs at https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/

Raspberry Pi OS

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

You will install on a uSD a **Raspberry Pi OS** image based on **kernel 5.4.y.** Go to https://www.raspberrypi.org/software/operating-systems/ and download Raspberry Pi OS with desktop and recommended software image.

Raspberry Pi OS with desktop and recommended software

Release date: August 20th 2020
Kernel version: 5.4
Size: 2,523MB
Show SHA256 file integrity hash:
Release notes

Download

Download torrent

To write the compressed image on the uSD 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

Enable UART, SPI and I2C peripherals in the programmed uSD:

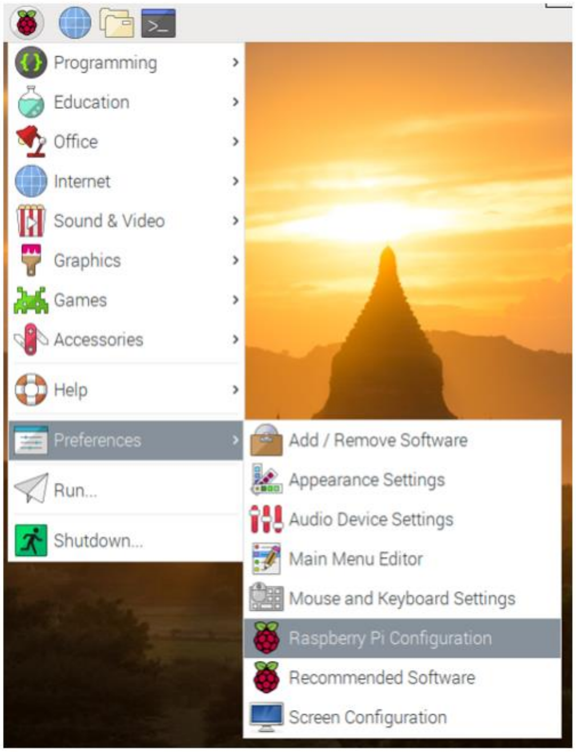
```
~$ lsblk
~$ mkdir ~/mnt
~$ mkdir ~/mnt/fat32
~$ mkdir ~/mnt/ext4
~$ sudo mount /dev/sdc1 ~/mnt/fat32
~$ ls -l ~/mnt/fat32/ /* see the files in the fat32 partition, check that
config.txt is included */
```

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

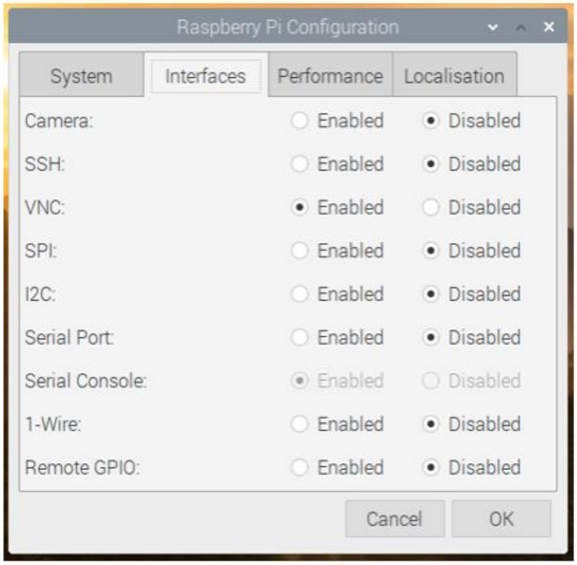
```
~$ cd ~/mnt/fat32/
~/mnt/fat32$ sudo nano config.txt
```

```
dtparam=i2c_arm=on
dtparam=spi=on
dtoverlay=spi0-cs
# Enable UART
enable_uart=1
kernel=kernel7l.img
```

You can also update previous settings (after booting the Raspberry Pi 4 board) through the Raspberry Pi 4 Configuration application found in Preferences on the menu.

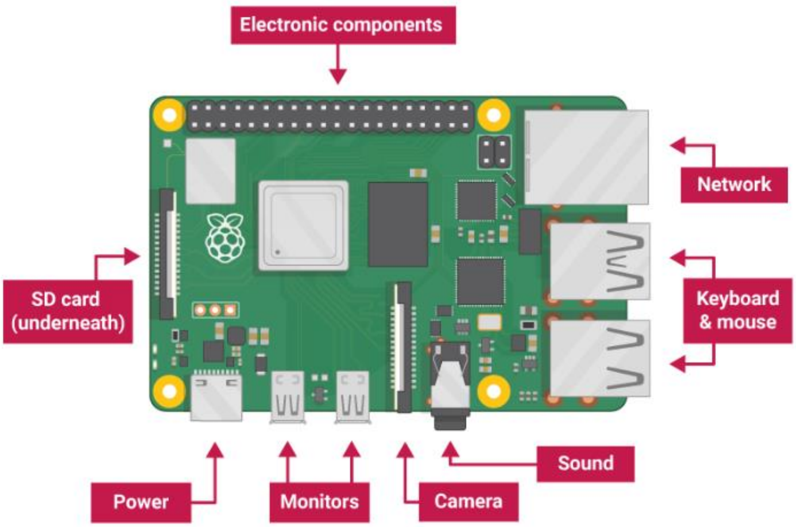


The Interfaces tab is where you turn these different connections on or off, so that the Pi recognizes that you've linked something to it via a particular type of connection:

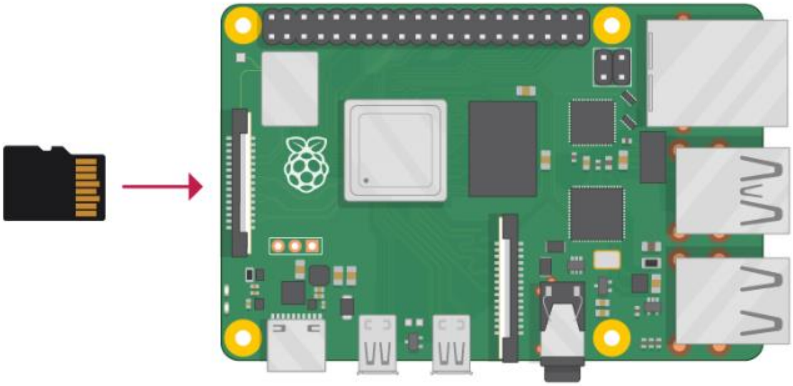


Connect and set up hardware

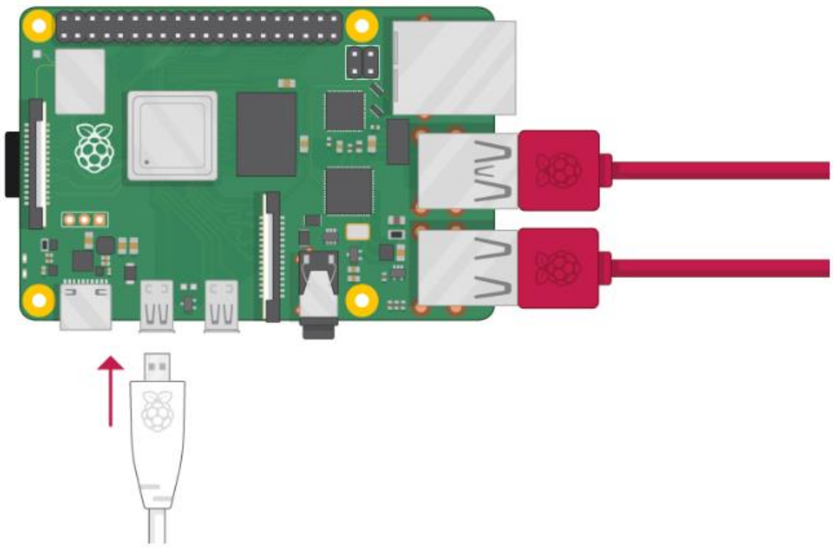
Now get everything connected to your Raspberry Pi 4. It's important to do this in the right order, so that all your components are safe.



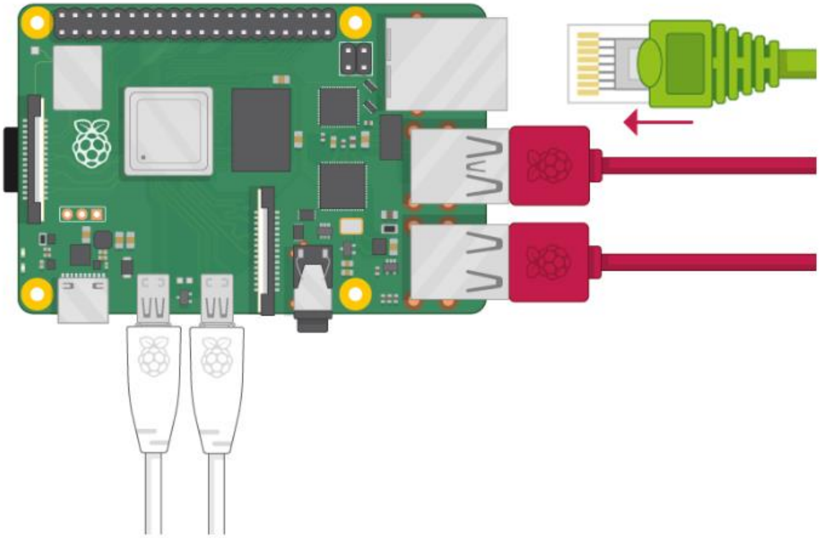
Insert the uSD card you've set up with **Raspberry Pi OS** into the microSD card slot on the underside of your Raspberry Pi 4.



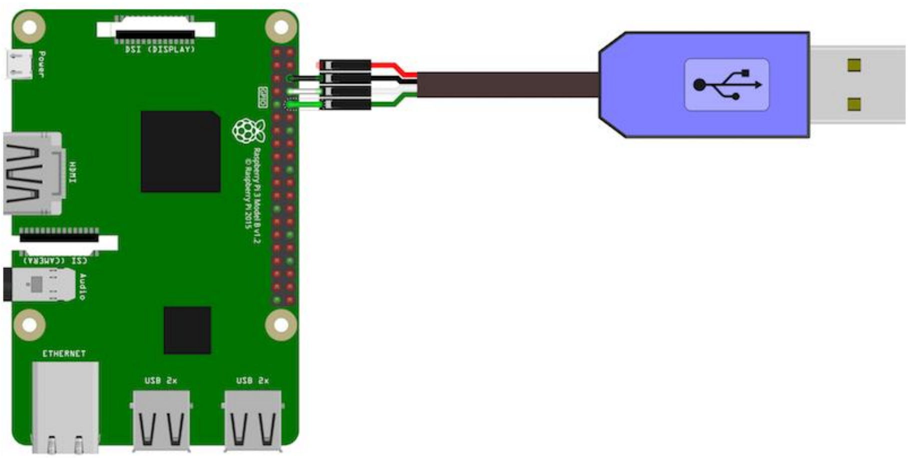
Connect your screen to the first of Raspberry Pi 4's HDMI ports, labelled HDMI0. You can also connect a mouse to a USB port and keyboard in the same way.



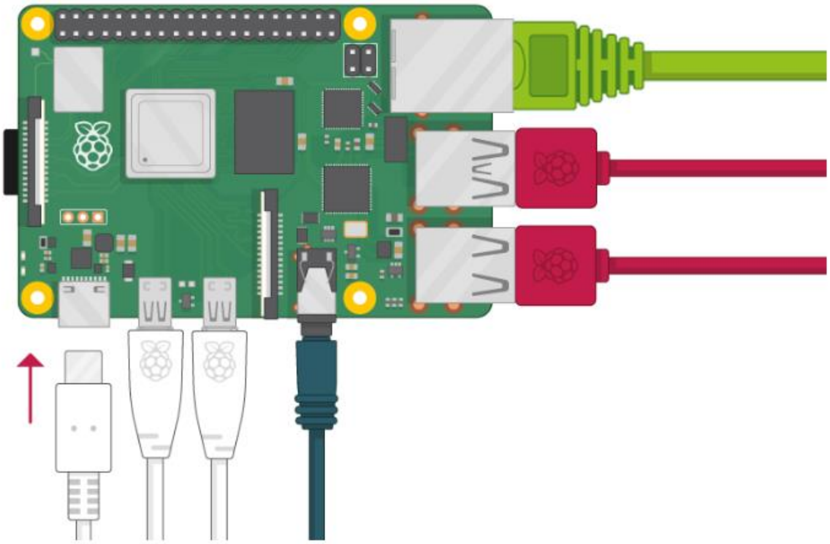
Connect your Raspberry Pi 4 to the internet via Ethernet, use an Ethernet cable to connect the Ethernet port on Raspberry Pi 4 to an Ethernet socket on the host PC.



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



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



You should see a red LED light up on the Raspberry Pi 4, which indicates that Raspberry Pi 4 is connected to power. As it starts up, you will see raspberries appear in the top left-hand corner of your screen. After a few seconds the Raspberry Pi OS Desktop will appear.

Launch a terminal on the host Linux PC by clicking on the Terminal icon. Type `dmesg` at the command prompt:

```
~$ dmesg
```

In the log message you can see that the new USB device is found and installed, for example **ttyUSB0**.

Launch and configure a serial console, for example **minicom** in your host to see the booting of the system. Through this console, you can access and control the Linux based system on the Raspberry Pi 4 Model B. Set the following configuration: **"115.2 kbaud, 8 data bits, 1 stop bit, no parity"**.

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

Reset the board. You can disconnect your screen from the Raspberry Pi 4's HDMI port during the development of the labs.

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

To see Linux boot messages on the console, change the loglevel to **8** in the file cmdline.txt under /boot

```
pi@raspberrypi:~$ sudo sudo nano /boot/cmdline.txt // loglevel=8
```

To change your current console_loglevel simply write to this file:

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

For example:

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

In that case, every kernel messages will appear on your console, as all priority higher than 8 (lower loglevel values) will be displayed. Please note that after reboot, this configuration is reset. To keep the configuration permanently, just append following line to /etc/sysctl.conf file in the Raspberry Pi 4:

```
kernel.printk = 8 4 1 3
```

```
pi@raspberrypi:~$ sudo nano /etc/sysctl.conf
```

Setting up ethernet communication

Connect an Ethernet cable between your host PC and the Raspberry Pi 4 Model B board. Set up the Linux host PC´s IP Address:

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.

Copying files to your Raspberry Pi

You can access the command line of a Raspberry Pi 4 remotely from another computer or device on the same network using SSH. Make sure the Raspberry Pi 4 is properly set up and connected. Configure the eth0 interface with IP address 10.0.0.10:

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

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

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

Now, verify that you can ping your Linux host machine from the Raspberry Pi 4 Model B. Exit the ping command by typing "Ctrl-c".

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

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

```
~$ ping 10.0.0.10
```

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

```
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** (comment the line out). After editing the file, type "Ctrl+x", then type "yes" and press "enter" to exit.

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

Building the Linux kernel

There are two main methods for building the kernel. You can build locally on the Raspberry Pi 4, which will take a long time; or you can cross-compile, which is much quicker, but requires more setup. You will use the second method.

Install Git and the build dependencies:

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

Get the kernel sources. The `git clone` command below will download the current active branch (the one we are building Raspberry Pi OS images from) without any history. Omitting the `--depth=1` will download the entire repository, including the full history of all branches, but this takes much longer and occupies much more storage.

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

Download the toolchain to the home folder:

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

Compile the kernel, modules and device tree files. First, apply the default configuration:

```
~/linux$ KERNEL=kernel7l
```

```
~/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2711_defconfig
```

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

```
~/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

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

Device drivers >
   <*> Industrial I/O support  --->
          -*-   Enable buffer support within IIO
          -*-   Industrial I/O buffering based on kfifo
          <*>  Enable 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

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

Save the configuration and exit from menuconfig.

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

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

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

```
~$ lsblk
~$ mkdir ~/mnt
~$ mkdir ~/mnt/fat32
~$ mkdir ~/mnt/ext4
~$ sudo mount /dev/sdd1 ~/mnt/fat32/
~$ sudo mount /dev/sdd2 ~/mnt/ext4/
~/linux$ sudo env PATH=$PATH make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
INSTALL_MOD_PATH=~/mnt/ext4 modules_install
```

Finally, update kernel, device tree files and modules:

```
~/linux$ sudo cp ~/mnt/fat32/kernel7l.img ~/mnt/fat32/kernel7l-backup.img
~/linux$ sudo cp arch/arm/boot/zImage ~/mnt/fat32/kernel7l.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/
```

```
~$ sudo umount ~/mnt/fat32
~$ sudo umount ~/mnt/ext4
```

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

```
pi@raspberrypi:~$ uname -r
5.4.77-v7l+
```

If you modify and compile the kernel or device tree files later, you can copy them to the Raspberry Pi 4 remotely using SSH:

```
~/linux$ scp arch/arm/boot/zImage root@10.0.0.10:/boot/kernel7l.img
~/linux$ scp arch/arm/boot/dts/bcm2711-rpi-4-b.dtb root@10.0.0.10:/boot/
```

Hardware descriptions for the Raspberry Pi 4 Model B labs

Use the same hardware descriptions of the Raspberry Pi 3 Model B labs developed through this book.

Software descriptions for the Raspberry Pi 4 Model B labs

LAB 5.2 software description

Change the peripheral base address from 0x3F000000 (Raspberry Pi 3 Model B) to 0xfe000000 (Raspberry Pi 4 Model B).

LAB 10.1 software description

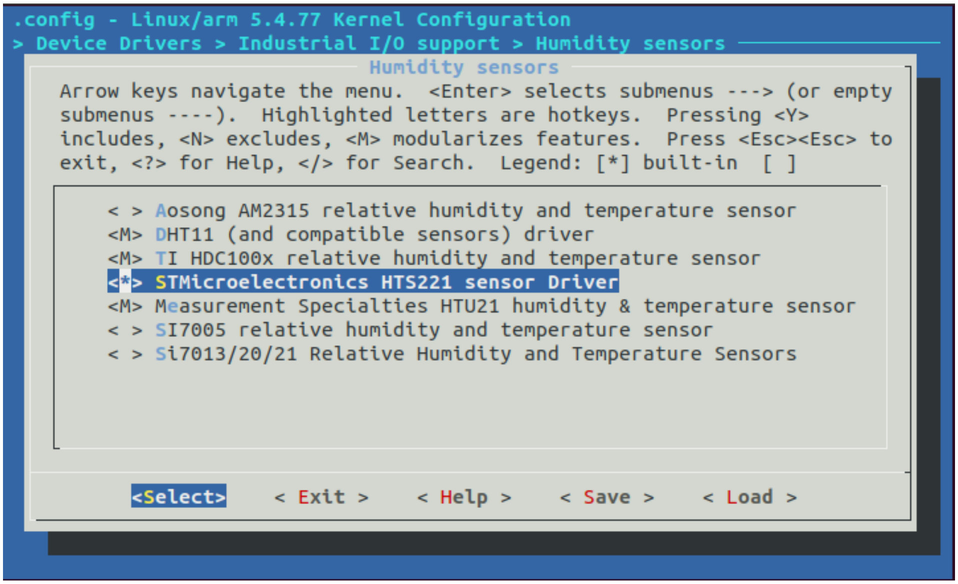
You have to install the `evtest` application to test this driver. Connect you Raspberry Pi 4 to the Internet and download the application:

```
root@raspberrypi:/home# sudo apt-get install evtest
```

LAB 12.1 software description

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

```
~/linux$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make menuconfig
```




```
~/linux$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage
~/linux$ scp arch/arm/boot/zImage root@10.0.0.10:/boot/kernel7l.img
```

Build the IIO tools on the host:

```
~/linux$ cd tools/iio/
~/linux/tools/iio$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
~/linux/tools/iio$ scp iio_generic_buffer pi@10.0.0.10:/home/
~/linux/tools/iio$ scp iio_event_monitor pi@10.0.0.10:/home/
```

The kernel 5.4 modules developed for the Raspberry Pi 4 Model B board are included in the linux_5.4_rpi4_drivers.zip file and can be downloaded from the GitHub repository at

https://github.com/ALIBERA/linux_book_2nd_edition.

Since the end of November 2020, the Linux drivers included in this book have been adapted to run on the Raspberry Pi 4 Model B board using Linux kernel version 5.4. The Raspberry Pi 4 Linux drivers and device tree settings can be downloaded from the Github repository of this book.

LAB 11.5: "IIO Mixed-Signal I/O Device" module

This new lab has been added to the labs of Chapter 11 to reinforce the concepts of creating IIO drivers explained during this chapter, and apply in a practical way how to create a gpio controller reinforcing thus the theory developed during Chapter 5. You will also develop several user applications to control GPIOs from user space.

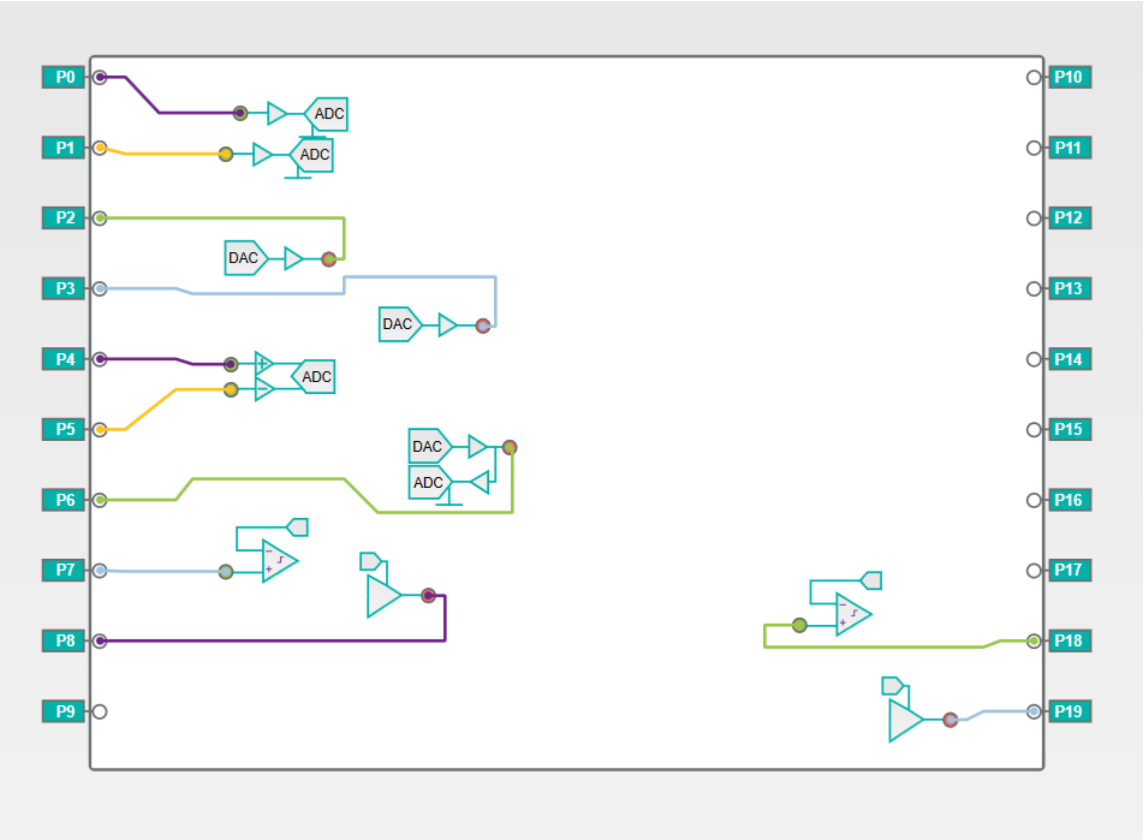
A new low cost evaluation board based on the MAX11300 device will be used, thus expanding the number of evaluation boards that can be acquired to practice with the theory explained in the Chapter 11.

This new kernel module will control the Maxim MAX11300 device. The MAX11300 integrates a PIXI™, 12-bit, multichannel, analog-to-digital converter (ADC) and a 12-bit, multichannel, buffered digital-to-analog converter (DAC) in a single integrated circuit (IC). This device offers 20 mixed-signal high-voltage, bipolar ports, which are configurable as an ADC analog input, a DAC analog output, a general-purpose input port (GPI), a general-purpose output port (GPO), or an analog switch terminal. You can check all the info related to this device at https://www.maximintegrated.com/en/products/analog/data-converters/analog-to-digital-converters/MAX11300.html

The hardware platforms used in this lab are the Raspberry Pi 4 Model B board and the PIXI™ CLICK from MIKROE. The documentation of these boards can be found at https://www.raspberrypi.org/products/raspberry-pi-4-model-b/?resellerType=home and https://www.mikroe.com/pixi-click

Before developing the driver, you can first create a custom design using a GUI software that configures the MAX11300 device. This tool is available for download at the Maxim's website. The MAX11300ConfigurationSetupV1.4.zip tool and the custom design used as a starting point for the development of the driver are included in the lab folder.

In the next screenshot of the tool, you can see the configuration that will be used during the development of the driver:

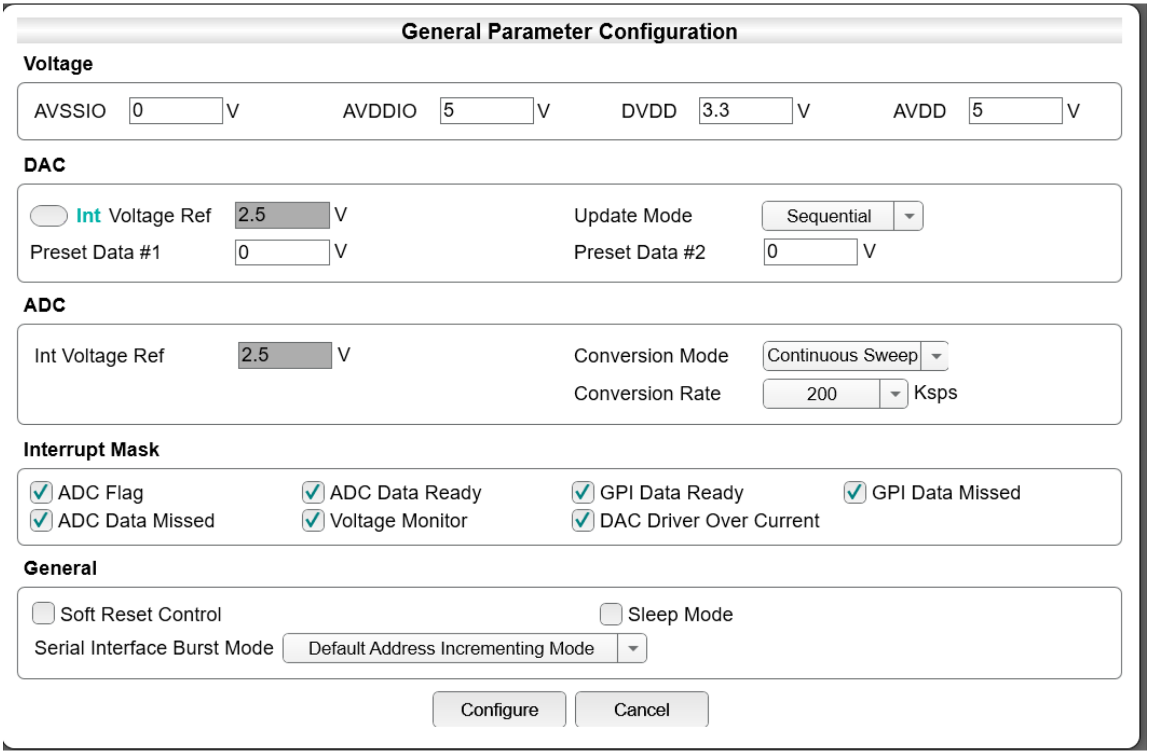


These are the parameters used during the configuration of the MAX11300´s PIXI ports:

- **Port 0 (P0)** -> Single Ended ADC, Average of samples = 1, Reference Voltage = internal, Voltage Range = 0V to 10V.
- **Port 1 (P1)** -> Single Ended ADC, Average of samples = 1, Reference Voltage = internal, Voltage Range = 0V to 10V.
- **Port 2 (P2)** -> DAC, Voltage Output Level = 0V, Voltage Range = 0V to 10V.
- **Port 3 (P3)** -> DAC, Voltage Output Level = 0V, Voltage Range = 0V to 10V.
- **Port 4 (P4) and Port 5 (P5)** -> Differential ADC, Pin info: Input Pin (-) is P5 and Input Pin (+) is P4, Reference Voltage = internal, Voltage Range = 0V to 10V.

- **Port 6 (P6)** -> DAC with ADC monitoring, Reference Voltage = internal, Voltage Output Level = 0V, Voltage Range = 0V to 10V.
- **Port 7 (P7)** -> GPI, Interrupt: Masked, Voltage Input Threshold: 2.5V.
- **Port 8 (P8)** -> GPO, Voltage output Level = 3.3V.
- **Port 18 (P18)** -> GPI, Interrupt: Masked, Voltage Input Threshold: 2.5V.
- **Port 19 (P19)** -> GPO, Voltage output Level = 3.3V.

And these are the general parameters used during the configuration of the MAX11300 device:



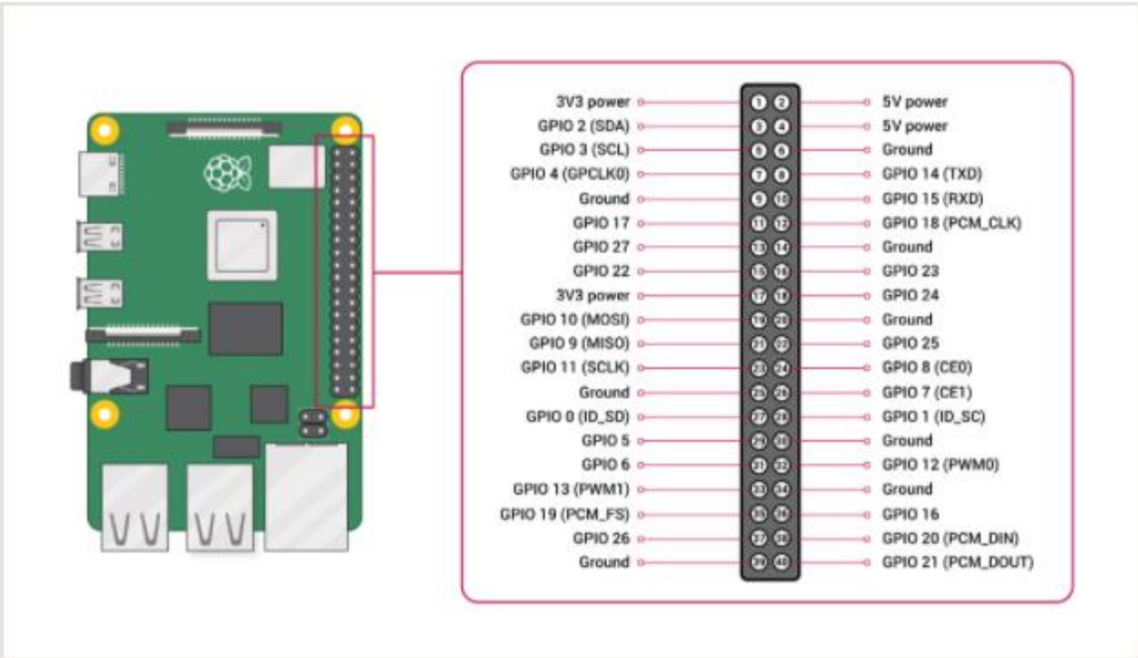
Not all the MAX11300´s specifications will be included during the development of this driver. These are the main specifications that will be included:

- Funcional modes for ports: Mode 1, Mode 3, Mode 5, Mode 6, Mode 7, Mode 8, Mode 9.

- DAC Update Mode: Sequential.
- ADC Conversion Mode: Continuous Sweep.
- Default ADC Conversion Rate of 200Ksps.
- Interrupts are masked.

LAB 11.5 hardware description

In this lab, you will use the SPI pins of the Raspberry Pi 4 Model B 40-pin GPIO header, which is found on all current Raspberry Pi boards, to connect to the PIXI™ CLICK mikroBUS™ socket. See below the Raspberry Pi 4 Model B connector:



And the PIXI™ CLICK mikroBUS™ socket:

Notes	Pin	mikroBUS				Pin	Notes
	NC	1	AN	PWM	16	**CNV**	ADC trigger control
	NC	2	RST	INT	15	**INT**	Interrupt output
Chip select	**CS**	3	CS	RX	14	NC	
SPI clock	**SCK**	4	SCK	TX	13	NC	
SPI data output	**SDO**	5	MISO	SCL	12	NC	
SPI data input	**SDI**	6	MOSI	SDA	11	NC	
Power supply	**+3.3V**	7	3.3V	5V	10	**+5V**	Power supply
Ground	**GND**	8	GND	GND	9	**GND**	Ground

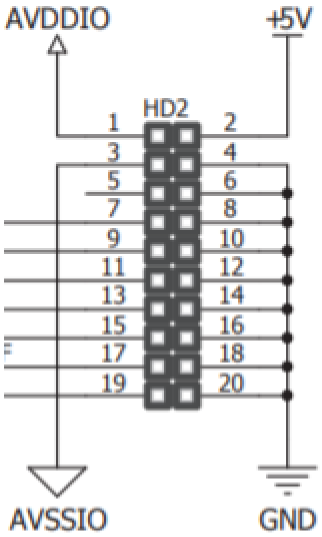
Connect the Raspberry Pi´s SPI pins to the SPI ones of the MAX11300 device, obtained from the PIXI™ CLICK mikroBUS™ socket:

- Connect Raspberry Pi 4 Model B **GPIO 8** to MAX11300 **CS** (Pin 3 of Mikrobus)
- Connect Raspberry Pi 4 Model B **SCLK** to MAX11300 **SCK** (Pin 4 of Mikrobus)
- Connect Raspberry Pi 4 Model B **MOSI** to MAX11300 **MOSI** (Pin 6 of Mikrobus)
- Connect Raspberry Pi 4 Model B **MISO** to MAX11300 **MISO** (Pin 5 of Mikrobus)

Also connect the next power pins between the two boards:

- Connect Raspberry Pi 4 Model B **3.3V** to MAX11300 **3.3V** (Pin 7 of Mikrobus)
- Connect Raspberry Pi 4 Model B **5V** to MAX11300 **5V** (Pin 10 of Mikrobus)
- Connect Raspberry Pi 4 Model B **GNDs** to MAX11300 **GNDs** (Pin 9 and Pin 8 of Mikrobus)

Finally, find the HD2 connector in the schematic of the PIXI™ CLICK board:
https://download.mikroe.com/documents/add-on-boards/click/pixi/pixi-click-schematic-v100.pdf



And connect the following pins:

- Connect the Pin 2 of HD2 (+5V) to the Pin 1 of HD2 (AVDDIO)
- Connect the Pin 4 of HD2 (GND) to the Pin 3 of HD2 (AVSSIO)

The hardware setup between the two boards is already done!!

LAB 11.5 device tree description

Open the bcm2711-rpi-4-b.dts DT file and find the spi0 controller master node. Inside the spi0 node, you can see the pinctrl-0 property, which configures the pins in SPI mode. Both spi0_pins and spi0_cs_pins are already defined in the bcm2711-rpi-4-b.dts file inside the gpio node.

The cs-gpios property specifies the gpio pins to be used for chip selects. In the spi0 node, you can see that there are two chip selects enabled. You will only use the first chip select <&gpio 8 1> during the development of this lab. Comment out all the sub-nodes included in the spi0 node coming from previous labs.

Now, you will add to the spi0 controller node the max11300 node, which includes twenty sub-nodes representing the different ports of the MAX11300 device. The first two properties inside the max11300 node are #size-cells and #address-cells. The #address-cells property defines the number of <u32> cells used to encode the address field in the child node's reg properties. The #size-cells property defines the number of <u32> cells used to encode the size field in the child node's reg properties. In this driver, the #address-cells property of the max11300 node is set to 1 and the #size-cells property is set to 0. This setting specifies that one cell is required to represent an address and there is no a required cell to represent the size of the nodes that are children of the max11300 node. The reg property included in all the channel nodes follows this specification that was set in the parent max11300 node.

There must be a DT device node´s compatible property identical to the compatible string stored in one of the driver´s of_device_id structures.

The spi-max-frequency specifies the maximum SPI clocking speed of device in Hz.

Each of the twenty child nodes can include the following properties:

- **reg** -> this property sets the port number of the MAX11300 device.
- **port-mode** -> this property sets the port configuration for the selected port.
- **AVR** -> this property selects the ADC voltage reference: 0: Internal, 1: External.
- **adc-range** -> this property selects the voltage range for ADC related modes.
- **dac-range** -> this property selects the voltage range for DAC related modes.
- **adc-samples** -> this property selects the number of samples for ADC related modes.
- **negative-input** -> this property sets the negative port number for ports configured in mode 8.

The channel sub-nodes will be configured with the same parameters that were used during the configuration of the MAX11300 GUI software:

```
&spi0 {
   pinctrl-names = "default";
   pinctrl-0 = <&spi0_pins &spi0_cs_pins>;
   cs-gpios = <&gpio 8 1>, <&gpio 7 1>;

   /* CE0 */
   /*spidev0: spidev@0{
          compatible = "spidev";
          reg = <0>;
          #address-cells = <1>;
          #size-cells = <0>;
          spi-max-frequency = <125000000>;
   };*/

   /* CE1 */
   /*spidev1: spidev@1{
          compatible = "spidev";
          reg = <1>;
```

```
        #address-cells = <1>;
        #size-cells = <0>;
        spi-max-frequency = <125000000>;
};*/

/*ADC: ltc2422@0 {
                compatible = "arrow,ltc2422";
                spi-max-frequency = <2000000>;
                reg = <0>;
                pinctrl-0 = <&key_pin>;
                int-gpios = <&gpio 23 0>;
};*/

max11300@0 {
        #size-cells = <0>;
        #address-cells = <1>;
        compatible = "maxim,max11300";
        reg = <0>;

        spi-max-frequency = <10000000>;

        channel@0 {
                reg = <0>;
                port-mode = <PORT_MODE_7>;
                AVR = <0>;
                adc-range = <ADC_VOLTAGE_RANGE_PLUS10>;
                adc-samples = <ADC_SAMPLES_1>;
        };
        channel@1 {
                reg = <1>;
                port-mode = <PORT_MODE_7>;
                AVR = <0>;
                adc-range = <ADC_VOLTAGE_RANGE_PLUS10>;
                adc-samples = <ADC_SAMPLES_128>;
        };
        channel@2 {
                reg = <2>;
                port-mode = <PORT_MODE_5>;
                dac-range = <DAC_VOLTAGE_RANGE_PLUS10>;
        };
        channel@3 {
                reg = <3>;
                port-mode = <PORT_MODE_5>;
                dac-range = <DAC_VOLTAGE_RANGE_PLUS10>;
        };
        channel@4 {
                reg = <4>;
                port-mode = <PORT_MODE_8>;
```

```
		AVR = <0>;
		adc-range = <ADC_VOLTAGE_RANGE_PLUS10>;
		adc-samples = <ADC_SAMPLES_1>;
		negative-input = <5>;
	};
	channel@5 {
		reg = <5>;
		port-mode = <PORT_MODE_9>;
		AVR = <0>;
		adc-range = <ADC_VOLTAGE_RANGE_PLUS10>;
	};
	channel@6 {
		reg = <6>;
		port-mode = <PORT_MODE_6>;
		AVR = <0>;
		dac-range = <DAC_VOLTAGE_RANGE_PLUS10>;
	};
	channel@7 {
		reg = <7>;
		port-mode = <PORT_MODE_1>;
	};
	channel@8 {
		reg = <8>;
		port-mode = <PORT_MODE_3>;
	};
	channel@9 {
		reg = <9>;
		port-mode = <PORT_MODE_0>;
	};
	channel@10 {
		reg = <10>;
		port-mode = <PORT_MODE_0>;
	};
	channel@11 {
		reg = <11>;
		port-mode = <PORT_MODE_0>;
	};
	channel@12 {
		reg = <12>;
		port-mode = <PORT_MODE_0>;
	};
	channel@13 {
		reg = <13>;
		port-mode = <PORT_MODE_0>;
	};
	channel@14 {
		reg = <14>;
		port-mode = <PORT_MODE_0>;
```

```
		};
		channel@15 {
			reg = <15>;
			port-mode = <PORT_MODE_0>;
		};
		channel@16 {
			reg = <16>;
			port-mode = <PORT_MODE_0>;
		};
		channel@17 {
			reg = <17>;
			port-mode = <PORT_MODE_0>;
		};
		channel@18 {
			reg = <18>;
			port-mode = <PORT_MODE_1>;
		};
		channel@19 {
			reg = <19>;
			port-mode = <PORT_MODE_3>;
		};
	};

	/*Accel: ADXL345@0 {
			compatible = "arrow,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>;
	};*/
};
```

You also have to include the next header file in bold inside the bcm2711-rpi-4-b.dts DT file.

```
// ==============================================
// Downstream rpi- changes

#include "bcm270x.dtsi"
#include "bcm271x-rpi-bt.dtsi"
#include <dt-bindings/iio/maxim,max11300.h>
```

The maxim,max11300.h file includes the values of the DT binding properties that will be used for the channel DT children nodes. You have to place the maxim,max11300.h file under the next iio folder in the kernel sources:

```
~/linux/include/dt-bindings/iio/
```

This is the content of the maxim,max11300.h file:

```
#ifndef _DT_BINDINGS_MAXIM_MAX11300_H
#define _DT_BINDINGS_MAXIM_MAX11300_H

#define    PORT_MODE_0    0
#define    PORT_MODE_1    1
#define    PORT_MODE_2    2
#define    PORT_MODE_3    3
#define    PORT_MODE_4    4
#define    PORT_MODE_5    5
#define    PORT_MODE_6    6
#define    PORT_MODE_7    7
#define    PORT_MODE_8    8
#define    PORT_MODE_9    9
#define    PORT_MODE_10   10
#define    PORT_MODE_11   11
#define    PORT_MODE_12   12


#define    ADC_SAMPLES_1    0
#define    ADC_SAMPLES_2    1
#define    ADC_SAMPLES_4    2
#define    ADC_SAMPLES_8    3
#define    ADC_SAMPLES_16   4
#define    ADC_SAMPLES_32   5
#define    ADC_SAMPLES_64   6
#define    ADC_SAMPLES_128   7

/* ADC voltage ranges */
#define    ADC_VOLTAGE_RANGE_NOT_SELECTED      0
#define    ADC_VOLTAGE_RANGE_PLUS10            1       // 0 to +5V range
#define    ADC_VOLTAGE_RANGE_PLUSMINUS5        2       // -5V to +5V range
#define    ADC_VOLTAGE_RANGE_MINUS10           3       // -10V to 0 range
#define    ADC_VOLTAGE_RANGE_PLUS25            4       // 0 to +2.5 range

/* DAC voltage ranges mode 5*/
#define    DAC_VOLTAGE_RANGE_NOT_SELECTED      0
#define    DAC_VOLTAGE_RANGE_PLUS10            1
#define    DAC_VOLTAGE_RANGE_PLUSMINUS5        2
#define    DAC_VOLTAGE_RANGE_MINUS10           3
```

```
#endif /* _DT_BINDINGS_MAXIM_MAX11300_H */
```

LAB 11.5 driver description

The main code sections of the driver will be described using three different categories: Industrial framework as an SPI interaction, Industrial framework as an IIO device and GPIO driver interface. The MAX11300 driver is based on Paul Cercueil´s AD5592R driver (https://elixir.bootlin.com/linux/latest/source/drivers/iio/dac/ad5592r.c)

Industrial framework as an SPI interaction

These are the main code sections:

1. Include the required header files:

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

2. Create a spi_driver structure:

   ```
   static struct spi_driver max11300_spi_driver = {
   	.driver = {
   		.name = "max11300",
   		.of_match_table = of_match_ptr(max11300_of_match),
   	},
   	.probe = max11300_spi_probe,
   	.remove = max11300_spi_remove,
   	.id_table = max11300_spi_ids,
   };
   module_spi_driver(max11300_spi_driver);
   ```

3. Register to the SPI bus as a driver:

   ```
   module_spi_driver(max11300_spi_driver);
   ```

4. Add "maxim,max11300" to the list of devices supported by the driver. The compatible variable matches with the compatible property of the max11300 DT node:

   ```
   static const struct of_device_id max11300_of_match[] = {
   	{ .compatible = "maxim,max11300", },
   	{},
   };
   MODULE_DEVICE_TABLE(of, max11300_of_match);
   ```

5. Define an array of spi_device_id structures:

   ```
   static const struct spi_device_id max11300_spi_ids[] = {
   	{ .name = "max11300", },
   	{}
   };
   MODULE_DEVICE_TABLE(spi, max11300_spi_ids);
   ```

6. Initialize the max11300_rw_ops structure with read and write callbacks that will access via SPI to the registers of the MAX11300 device. See below the code of these callbacks:

```
/* Initialize the struct max11300_rw_ops with read and write callback functions
to write/read via SPI from MAX11300 registers */
static const struct max11300_rw_ops max11300_rw_ops = {
	.reg_write = max11300_reg_write,
	.reg_read = max11300_reg_read,
	.reg_read_differential = max11300_reg_read_differential,
};

/* function to write MAX11300 registers */
static int max11300_reg_write(struct max11300_state *st, u8 reg, u16 val)
{
	struct spi_device *spi = container_of(st->dev, struct spi_device, dev);

	struct spi_transfer t[] = {
		{
			.tx_buf = &st->tx_cmd,
			.len = 1,
		}, {
			.tx_buf = &st->tx_msg,
			.len = 2,
		},
	};

	/* to transmit via SPI the LSB bit of the command byte must be 0 */
	st->tx_cmd = (reg << 1);

	/*
	 * In little endian CPUs the byte stored in the higher address of the
	 * "val" variable (MSB of the DAC) is stored in the lower address of the
	 * "st->tx_msg" variable using cpu_to_be16()
	 */
	st->tx_msg = cpu_to_be16(val);

	return spi_sync_transfer(spi, t, ARRAY_SIZE(t));
}
/* function to read MAX11300 registers in SE mode */
static int max11300_reg_read(struct max11300_state *st, u8 reg, u16 *value)
{
	struct spi_device *spi = container_of(st->dev, struct spi_device, dev);
	int ret;

	struct spi_transfer t[] = {
		{
			.tx_buf = &st->tx_cmd,
			.len = 1,
```

```
		}, {
			.rx_buf = &st->rx_msg,
			.len = 2,
		},
	};

	dev_info(st->dev, "read SE channel\n");

	/* to receive via SPI the LSB bit of the command byte must be 1 */
	st->tx_cmd = ((reg << 1) | 1);

	ret = spi_sync_transfer(spi, t, ARRAY_SIZE(t));
	if (ret < 0)
		return ret;

	/*
	 * In little endian CPUs the first byte (MSB of the ADC) received via
	 * SPI (in BE format) is stored in the lower address of "st->rx_msg"
	 * variable. This byte is copied to the higher address of the "value"
	 * variable using be16_to_cpu(). The second byte received via SPI is
	 * copied from the higher address of "st->rx_msg" to the lower address
	 * of the "value" variable in little endian CPUs.
	 * In big endian CPUs the addresses are not swapped.
	 */

	*value = be16_to_cpu(st->rx_msg);

	return 0;
}

/* function to read MAX11300 registers in differential mode (2's complement) */
static int max11300_reg_read_differential(struct max11300_state *st, u8 reg,
					  int *value)
{
	struct spi_device *spi = container_of(st->dev, struct spi_device, dev);
	int ret;

	struct spi_transfer t[] = {
		{
			.tx_buf = &st->tx_cmd,
			.len = 1,
		}, {
			.rx_buf = &st->rx_msg,
			.len = 2,
		},
	};

	dev_info(st->dev, "read differential channel\n");
```

```
	/* to receive LSB of command byte has to be 1 */
	st->tx_cmd = ((reg << 1) | 1);

	ret = spi_sync_transfer(spi, t, ARRAY_SIZE(t));
	if (ret < 0)
		return ret;



	/*
	 * extend to an int 2's complement value the received SPI value in 2's
	 * complement value, which is stored in the "st->rx_msg" variable
	 */

	*value = sign_extend32(be16_to_cpu(st->rx_msg), 11);

	return 0;
}
```

Industrial framework as an IIO device

These are the main code sections:

1. Include the required header files:

```
#include <linux/iio/iio.h> /* devm_iio_device_alloc(), iio_priv() */
```

2. Create a global private data structure to manage the device from any function of the driver:

```
struct max11300_state {
	struct device *dev; // pointer to SPI device
	const struct max11300_rw_ops *ops; // pointer to spi callback functions
	struct gpio_chip gpiochip; // gpio_chip controller
	struct mutex gpio_lock;
	u8 num_ports;  // number of ports of the MAX11300 device = 20
	u8 num_gpios;  // number of ports declared in the DT as GPIOs
	u8 gpio_offset[20]; // gpio port numbers (0 to 19) for the "offset"
values in the range 0..(@ngpio - 1)
	u8 gpio_offset_mode[20]; // gpio port modes (1 and 3) for the "offset"
values in the range 0..(@ngpio - 1)
	u8 port_modes[20]; // port modes for the 20 ports of the MAX11300
	u8 adc_range[20]; // voltage range for ADC related modes
	u8 dac_range[20]; // voltage range for DAC related modes
	u8 adc_reference[20]; // ADC voltage reference: 0: Internal, 1: External
	u8 adc_samples[20]; // number of samples for ADC related modes
	u8 adc_negative_port[20]; // negative port number for ports configured
in mode 8
	u8 tx_cmd; // command byte for SPI transactions
	__be16 tx_msg; // transmit value for SPI transactions in BE format
```

```
        __be16 rx_msg; // value received in SPI transactions in BE format
};
```

3. In the max11300_probe() function, declare an instance of the private structure and allocate the iio_dev structure.

```
struct iio_dev *indio_dev;
struct max11300_state *st;
indio_dev = devm_iio_device_alloc(dev, sizeof(*st));
```

4. Initialize the iio_device and the data private structure within the max11300_probe() function. The data private structure will be previously allocated by using the iio_priv() function. Keep pointers between physical devices (devices as handled by the physical bus, SPI in this case) and logical devices:

```
st = iio_priv(indio_dev); /* To be able to access the private data structure in
other parts of the driver you need to attach it to the iio_dev structure using
the iio_priv() function.You will retrieve the pointer "data" to the private
structure using the same function iio_priv() */

st->dev = dev; /* Keep pointer to the SPI device, needed for exchanging data
with the MAX11300 device */

dev_set_drvdata(dev, iio_dev); /* link the spi device with the iio device */

iio_dev->name = name; /* Store the iio_dev name. Before doing this within
your probe() function, you will get the spi_device_id that triggered the match
using spi_get_device_id() */

iio_dev->dev.parent = dev; /* keep pointers between physical devices
(devices as handled by the physical bus, SPI in this case) and logical devices
*/

indio_dev->info = &max11300_info; /* store the address of the iio_info
structure which contains a pointer variable to the IIO raw reading/writing
callbacks */

max11300_alloc_ports(st); /* configure the IIO channels of the device to
generate the IIO sysfs entries. This function will be described in more detail
in the next point */
```

5. The max11300_alloc_ports() function will read the properties from the channel DT children nodes of the max11300 DT node by using the fwnode_property_read_u32() function, and will store the values of these properties into the variables of the data global structure. The function max11300_set_port_modes() will use these variables to configure the ports of the MAX11300 device. The max11300_alloc_ports() function will also generate the different IIO sysfs entries using the max11300_setup_port_*_mode() functions:

```
/*
```

```
 * this function will allocate and configure the iio channels of the iio device
 * It will also read the DT properties of each port (channel) and will store
 * them in the global structure of the device
 */
static int max11300_alloc_ports(struct max11300_state *st)
{
	unsigned int i, curr_port = 0, num_ports = st->num_ports,
port_mode_6_count = 0, offset = 0;
	st->num_gpios = 0;

	/* recover the iio device from the global structure */
	struct iio_dev *iio_dev = iio_priv_to_dev(st);

	/* pointer to the storage of the specs of all the iio channels */
	struct iio_chan_spec *ports;

	/* pointer to struct fwnode_handle allowing device description object */
	struct fwnode_handle *child;

	u32 reg, tmp;
	int ret;

	/*
	 * walks for each MAX11300 child node from the DT,
	 * if an error is found in the node then walks to
	 * the following one (continue)
	 */
	device_for_each_child_node(st->dev, child) {
		ret = fwnode_property_read_u32(child, "reg", &reg);
		if (ret || reg >= ARRAY_SIZE(st->port_modes))
			continue;

		/* store the value of the DT "port,mode" property
		 * in the global structure to know the mode of each port in
		 * other functions of the driver
		 */
		ret = fwnode_property_read_u32(child, "port-mode", &tmp);
		if (!ret)
			st->port_modes[reg] = tmp;

		/* all the DT nodes should include the port-mode property */
		else {
			dev_info(st->dev, "port mode is not found\n");
			continue;
		}

		/*
		 * you will store other DT properties
```

```
 * depending of the used "port,mode" property
 */
switch (st->port_modes[reg]) {
case PORT_MODE_7:
        ret = fwnode_property_read_u32(child, "adc-range", &tmp);
        if (!ret)
                st->adc_range[reg] = tmp;
        else
                dev_info(st->dev, "Get default ADC range\n");

        ret = fwnode_property_read_u32(child, "AVR", &tmp);
        if (!ret)
                st->adc_reference[reg] = tmp;
        else
                dev_info(st->dev, "Get default internal ADC
                         reference\n");

        ret = fwnode_property_read_u32(child, "adc-samples",
                                       &tmp);
        if (!ret)
                st->adc_samples[reg] = tmp;
        else
                dev_info(st->dev, "Get default internal ADC
                         sampling\n");

        break;

case PORT_MODE_8:
        ret = fwnode_property_read_u32(child, "adc-range", &tmp);
        if (!ret)
                st->adc_range[reg] = tmp;
        else
                dev_info(st->dev, "Get default ADC range\n");

        ret = fwnode_property_read_u32(child, "AVR", &tmp);
        if (!ret)
                st->adc_reference[reg] = tmp;
        else
                dev_info(st->dev, "Get default internal ADC
                         reference\n");

        ret = fwnode_property_read_u32(child, "adc-samples",
                                       &tmp);
        if (!ret)
                st->adc_samples[reg] = tmp;
        else
                dev_info(st->dev, "Get default internal ADC
                         sampling\n");
```

```
                ret = fwnode_property_read_u32(child, "negative-input",
                                               &tmp);
                if (!ret)
                        st->adc_negative_port[reg] = tmp;
                else {
                        dev_info(st->dev, "Bad value for negative ADC
                                 channel\n");
                        return -EINVAL;
                }

                break;

        case PORT_MODE_9: case PORT_MODE_10:
                ret = fwnode_property_read_u32(child, "adc-range", &tmp);
                if (!ret)
                        st->adc_range[reg] = tmp;
                else
                        dev_info(st->dev, "Get default ADC range\n");

                ret = fwnode_property_read_u32(child, "AVR", &tmp);
                if (!ret)
                        st->adc_reference[reg] = tmp;
                else
                        dev_info(st->dev, "Get default internal ADC
                                 reference\n");

                break;

        case PORT_MODE_5: case PORT_MODE_6:
                ret = fwnode_property_read_u32(child, "dac-range", &tmp);
                if (!ret)
                st->dac_range[reg] = tmp;
                else
                        dev_info(st->dev, "Get default DAC range\n");

                /*
                 * A port in mode 6 will generate two IIO sysfs entries,
                 * one for writing the DAC port, and another for reading
                 * the ADC port
                 */
                if ((st->port_modes[reg]) == PORT_MODE_6) {
                        ret = fwnode_property_read_u32(child, "AVR",
                                                       &tmp);
                        if (!ret)
                                st->adc_reference[reg] = tmp;
                        else
```

```
                        dev_info(st->dev, "Get default internal
                                 ADC reference\n");

                /*
                 * get the number of ports set in mode_6 to
                 * allocate space for the realated iio channels
                 */
                port_mode_6_count++;
        }

        break;

/* The port is configured as a GPI in the DT */
case PORT_MODE_1:
        /*
         * link the gpio offset with the port number,
         * starting with offset = 0
         */
        st->gpio_offset[offset] = reg;



        /*
         * store the port_mode for each gpio offset,
         * starting with offset = 0
         */
        st->gpio_offset_mode[offset] = PORT_MODE_1;

        /*
         * increment the gpio offset and number of configured
         * ports as GPIOs
         */
        offset++;
        st->num_gpios++;
        break;

/* The port is configured as a GPO in the DT */
case PORT_MODE_3:
        /*
         * link the gpio offset with the port number,
         * starting with offset = 0
         */
        st->gpio_offset[offset] = reg;

        /*
         * store the port_mode for each gpio offset,
         * starting with offset = 0
         */
```

```
                        st->gpio_offset_mode[offset] = PORT_MODE_3;

                        /*
                         * increment the gpio offset and
                         * number of configured ports as GPIOs
                         */
                        offset++;
                        st->num_gpios++;
                        break;

                case PORT_MODE_0:
                        dev_info(st->dev, "the channel %d is set in default port
                                mode_0\n", reg);
                        break;

                default:
                        dev_info(st->dev, "bad port mode for channel %d\n", reg);
                }

        }

        /*
         * Allocate space for the storage of all the IIO channels specs.
         * Returns a pointer to this storage
         */
        devm_kcalloc(st->dev, num_ports + port_mode_6_count,
                      sizeof(*ports), GFP_KERNEL);

        /*
         * i is the number of the channel, &ports[curr_port] is a pointer
         * variable that will store the "iio_chan_spec structure" address of
         * each port
         */
        for (i = 0; i < num_ports; i++) {
                switch (st->port_modes[i]) {
                case PORT_MODE_5:
                        max11300_setup_port_5_mode(iio_dev, &ports[curr_port],
                                                   true, i, PORT_MODE_5);
                        curr_port++;
                        break;

                case PORT_MODE_6:
                        max11300_setup_port_6_mode(iio_dev, &ports[curr_port],
                                                   true, i, PORT_MODE_6);
                        curr_port++;
                        max11300_setup_port_6_mode(iio_dev, &ports[curr_port],
                                                  false, i, PORT_MODE_6);
                        curr_port++;
```

```
                break;

        case PORT_MODE_7:
                max11300_setup_port_7_mode(iio_dev, &ports[curr_port],
                                           false, i, PORT_MODE_7);
                curr_port++;
                break;

        case PORT_MODE_8:
                max11300_setup_port_8_mode(iio_dev, &ports[curr_port],
                        false, i, st->adc_negative_port[i], PORT_MODE_8);

                curr_port++;
                break;

        case PORT_MODE_0:
                dev_info(st->dev, "the channel is set in default port
                         mode_0\n");
                break;

        case PORT_MODE_1:
                dev_info(st->dev, "the channel %d is set in port
                         mode_1\n", i);
                break;

        case PORT_MODE_3:
                dev_info(st->dev, "the channel %d is set in port
                         mode_3\n", i);
                break;

        default:
                dev_info(st->dev, "bad port mode for channel %d\n", i);
        }
    }

    iio_dev->num_channels = curr_port;
    iio_dev->channels = ports;

    return 0;
}
```

6. Write the iio_info structure. The read/write user space operations to sysfs data channel access attributes are mapped to the following kernel callbacks:

```
static const struct iio_info max11300_info = {
        .read_raw = max11300_read_adc,
        .write_raw = max11300_write_dac,
};
```

The max11300_write_dac() function contains a switch(mask) that sets different tasks depending of the received parameter values. If the received info_mask value is [IIO_CHAN_INFO_RAW] = "raw", the max11300_reg_write() function is called, which writes a DAC value (using an SPI transaction) to the selected port DAC data register.

When the max11300_read_adc() function receives the info_mask value [IIO_CHAN_INFO_RAW] = "raw", it first reads the value of the ADC channel address to select the ADC port mode. Once the ADC port mode has been set up, then max11300_reg_read() or max11300_reg_read_differential() functions are called, which get via an SPI transaction, the value of the selected port ADC data register. The returned ADC value is stored into the val variable and this value is returned to the user space through the IIO_VAL_INT identifier.

GPIO driver interface

The MAX11300 driver will also include a GPIO controller, which will configure and control the MAX11300 ports selected as GPIOs (Port 1 and Port 3 modes) in the DT node of the device.

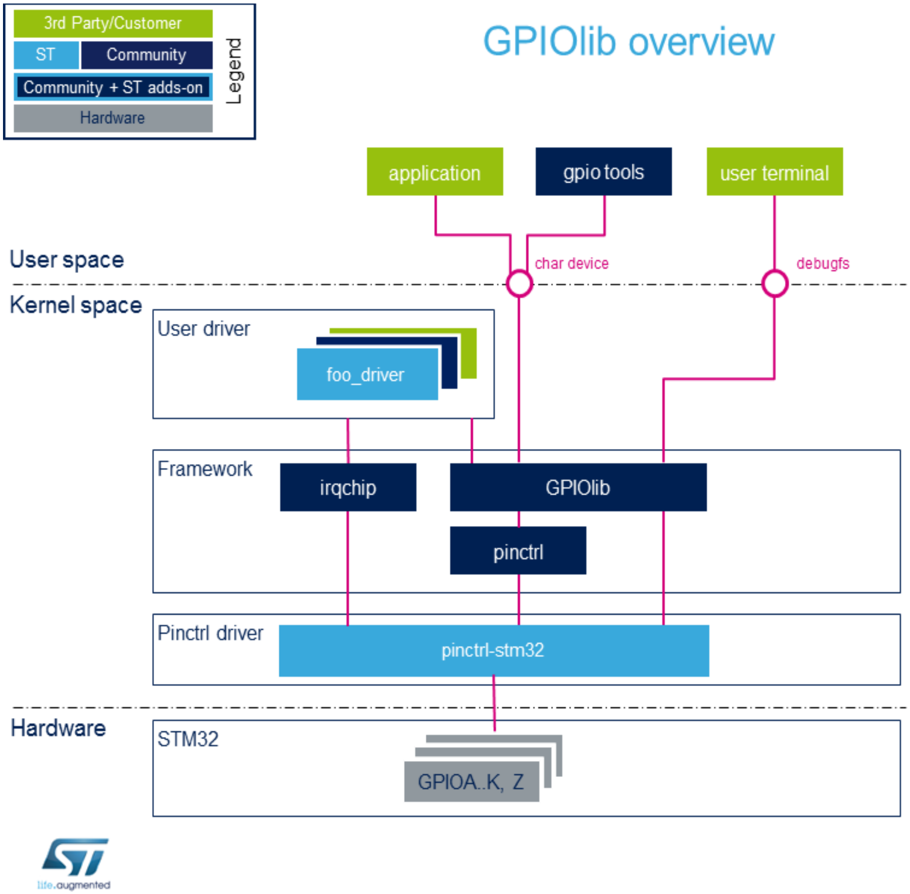
In the Chapter 5 of this book , you saw how to control GPIOs from kernel space using the GPIO descriptor consumer interface of the GPIOLib framework.

Most processors today use composite pin controllers. These composite pin controllers will control the GPIOs of the processor, generate interrupts on top of the GPIO functionality and allow pin multiplexing using the I/O pins of the processor as GPIOs or as one of several peripheral functions. The composite pin controllers are configured using a pinctrl driver.

The pinctrl driver will register the gpio_chip structures with the kernel, the irq_chip structures with the IRQ system and the pinctrl_desc structures with the Pinctrl subsystem. The gpio and pin controllers are associated with each other within the pinctrl driver through the pinctrl_add_gpio_range() function, which adds a range of GPIOs to be handled by a certain pin controller. In the section 2.1 of the gpio device tree binding document at https://elixir.bootlin.com/linux/latest/source/Documentation/devicetree/bindings/gpio/gpio.txt , you can see the gpio and pin controller interaction within the DT sources.

The GPIOLib framework will provide the kernel and user space APIs to control the GPIOs.

In the next image, taken from the STM32MP1 wiki article at https://wiki.st.com/stm32mpu/wiki/GPIOLib_overview, you can see the interaction between different kernel drivers and frameworks to control the GPIO chips. You can also see in this article a description of the blocks shown in the image below.



Our MAX11300 IIO driver will include a basic GPIO controller, which configures the ports of the MAX11300 device as GPIOs, sets the direction of the GPIOs (input or output) and controls the output level of the GPIO lines (low or high output level).

These are the main steps to create the GPIO controller in our MAX11300 IIO driver:

1. Include the following header, which defines the structures used to define a GPIO driver:

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

2. Initialize the gpio_chip structure with the different callbacks that will control the gpio lines of the GPIO controller and register the gpio chip with the kernel using the gpiochip_add_data() function:

```
static int max11300_gpio_init(struct max11300_state *st)
{

	st->gpiochip.label = "gpio-max11300";
	st->gpiochip.base = -1;
	st->gpiochip.ngpio = st->num_gpios;
	st->gpiochip.parent = st->dev;
	st->gpiochip.can_sleep = true;
	st->gpiochip.direction_input = max11300_gpio_direction_input;
	st->gpiochip.direction_output = max11300_gpio_direction_output;
	st->gpiochip.get = max11300_gpio_get;
	st->gpiochip.set = max11300_gpio_set;
	st->gpiochip.owner = THIS_MODULE;

	/* register a gpio_chip */
	return gpiochip_add_data(&st->gpiochip, st);
}
```

3. These are the callback functions that will control the GPIO lines of the MAX11300 GPIO controller:

```
/*
 * struct gpio_chip get callback function.
 * It gets the input value of the GPIO line (0=low, 1=high)
 * accessing to the GPI_DATA registers of the MAX11300
 */
static int max11300_gpio_get(struct gpio_chip *chip, unsigned int offset)
{
	struct max11300_state *st = gpiochip_get_data(chip);
	int ret = 0;
	u16 read_val;
	u8 reg;
	int val;

	mutex_lock(&st->gpio_lock);

	if (st->gpio_offset_mode[offset] == PORT_MODE_3)
	dev_info(st->dev, "the gpio %d cannot be configured in input mode\n",
		 offset);

	/* for GPIOs from 16 to 19 ports */
	if (st->gpio_offset[offset] > 0x0F) {
		reg = GPI_DATA_19_TO_16_ADDRESS;
```

```
		ret = st->ops->reg_read(st, reg, &read_val);
		if (ret)
			goto err_unlock;

		val = (int) (read_val);
		val = val << 16;

		if (val & BIT(st->gpio_offset[offset]))
			val = 1;
		else
			val = 0;

		mutex_unlock(&st->gpio_lock);
		return val;
	}
	else {
		reg = GPI_DATA_15_TO_0_ADDRESS;
		ret = st->ops->reg_read(st, reg, &read_val);
		if (ret)
			goto err_unlock;

		val = (int) read_val;

		if(val & BIT(st->gpio_offset[offset]))
			val = 1;
		else
			val = 0;

		mutex_unlock(&st->gpio_lock);
		return val;
	}

err_unlock:
	mutex_unlock(&st->gpio_lock);
	return ret;
}

/*
 * struct gpio_chip set callback function.
 * It sets the output value of the GPIO line with
 * GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the GPO_DATA registers of the max11300
 */
static void max11300_gpio_set(struct gpio_chip *chip, unsigned int offset,
				int value)
{
	struct max11300_state *st = gpiochip_get_data(chip);
	u8 reg;
```

```
	unsigned int val = 0;

	mutex_lock(&st->gpio_lock);

	if (st->gpio_offset_mode[offset] == PORT_MODE_1)
	dev_info(st->dev, "the gpio %d cannot accept this output\n", offset);

	if (value == 1 && (st->gpio_offset[offset] > 0x0F)) {
		dev_info(st->dev, "The GPIO ouput is set high and port_number is
			%d. Pin is > 0x0F\n", st->gpio_offset[offset]);
		val |= BIT(st->gpio_offset[offset]);
		val = val >> 16;
		reg = GPO_DATA_19_TO_16_ADDRESS;
		st->ops->reg_write(st, reg, val);
	}
	else if (value == 0 && (st->gpio_offset[offset] > 0x0F)) {
		dev_info(st->dev, "The GPIO ouput is set low and port_number is
			%d. Pin is > 0x0F\n", st->gpio_offset[offset]);
		val &= ~BIT(st->gpio_offset[offset]);
		val = val >> 16;
		reg = GPO_DATA_19_TO_16_ADDRESS;
		st->ops->reg_write(st, reg, val);
	}
	else if (value == 1 && (st->gpio_offset[offset] < 0x0F)) {
		dev_info(st->dev, "The GPIO ouput is set high and port_number is
			%d. Pin is < 0x0F\n", st->gpio_offset[offset]);
		val |= BIT(st->gpio_offset[offset]);
		reg = GPO_DATA_15_TO_0_ADDRESS;
		st->ops->reg_write(st, reg, val);
	}
	else if (value == 0 && (st->gpio_offset[offset] < 0x0F)) {
		dev_info(st->dev, "The GPIO ouput is set low and port_number is
			%d. Pin is < 0x0F\n", st->gpio_offset[offset]);
		val &= ~BIT(st->gpio_offset[offset]);
		reg = GPO_DATA_15_TO_0_ADDRESS;
		st->ops->reg_write(st, reg, val);
	}
	else
		dev_info(st->dev, "the gpio %d cannot accept this value\n",
			offset);

	mutex_unlock(&st->gpio_lock);
}

/*
 * struct gpio_chip direction_input callback function.
 * It configures the GPIO port as an input (GPI)
 * writing to the PORT_CFG register of the max11300
```

```
 */
static int max11300_gpio_direction_input(struct gpio_chip *chip,
                                         unsigned int offset)
{
        struct max11300_state *st = gpiochip_get_data(chip);
        int ret;
        u8 reg;
        u16 port_mode, val;

        mutex_lock(&st->gpio_lock);


        /* get the port number stored in the GPIO offset */
        if (st->gpio_offset_mode[offset] == PORT_MODE_3)
                dev_info(st->dev, "Error.The gpio %d only can be set in output
                        mode\n", offset);

        /* Set the logic 1 input above 2.5V level */
        val = 0x0fff;

        /* store the GPIO threshold value in the port DAC register */
        reg = PORT_DAC_DATA_BASE_ADDRESS + st->gpio_offset[offset];
        ret = st->ops->reg_write(st, reg, val);
        if (ret)
                goto err_unlock;

        /* Configure the port as GPI */
        reg = PORT_CFG_BASE_ADDRESS + st->gpio_offset[offset];
        port_mode = (1 << 12);
        ret = st->ops->reg_write(st, reg, port_mode);
        if (ret)
                goto err_unlock;

        mdelay(1);

err_unlock:
        mutex_unlock(&st->gpio_lock);

        return ret;
}

/*
 * struct gpio_chip direction_output callback function.
 * It configures the GPIO port as an output (GPO) writing to
 * the PORT_CFG register of the max11300 and sets output value of the
 * GPIO line with GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the GPO data registers of the max11300
 */
```

```
static int max11300_gpio_direction_output(struct gpio_chip *chip,
                                          unsigned int offset, int value)
{
        struct max11300_state *st = gpiochip_get_data(chip);
        int ret;
        u8 reg;
        u16 port_mode, val;

        mutex_lock(&st->gpio_lock);

        dev_info(st->dev, "The GPIO is set as an output\n");

        if (st->gpio_offset_mode[offset] == PORT_MODE_1)
                dev_info(st->dev, "the gpio %d only can be set in input mode\n",
                         offset);

        /* GPIO output high is 3.3V */
        val = 0x0547;

        reg = PORT_DAC_DATA_BASE_ADDRESS + st->gpio_offset[offset];
        ret = st->ops->reg_write(st, reg, val);
        if (ret) {
                mutex_unlock(&st->gpio_lock);
                return ret;
        }
        mdelay(1);
        reg = PORT_CFG_BASE_ADDRESS + st->gpio_offset[offset];
        port_mode = (3 << 12);
        ret = st->ops->reg_write(st, reg, port_mode);
        if (ret) {
                mutex_unlock(&st->gpio_lock);
                return ret;
        }
        mdelay(1);

        mutex_unlock(&st->gpio_lock);

        max11300_gpio_set(chip, offset, value);

        return ret;
}
```

See in the next **Listings** the complete " IIO Mixed-Signal I/O Device" driver source code for the Raspberry Pi 4 Model B processor.

Note: The " IIO Mixed-Signal I/O Device" driver source code developed for the Raspberry Pi 4 Model B board is included in the linux_5.4_rpi4_drivers.zip file inside the linux_5.4_max11300_driver folder and can be downloaded from the GitHub repository at https://github.com/ALIBERA/linux_book_2nd_edition

Listing 11-6: max11300-base.h

```
#ifndef __DRIVERS_IIO_DAC_max11300_BASE_H__
#define __DRIVERS_IIO_DAC_max11300_BASE_H__

#include <linux/types.h>
#include <linux/cache.h>
#include <linux/mutex.h>
#include <linux/gpio/driver.h>

struct max11300_state;

/* masks for the Device Control (DCR) Register */
#define DCR_ADCCTL_CONTINUOUS_SWEEP (BIT(0) | BIT(1))
#define DCR_DACREF BIT(6)
#define BRST BIT(14)
#define RESET BIT(15)

/* define register addresses */
#define DCR_ADDRESS 0x10
#define PORT_CFG_BASE_ADDRESS 0x20
#define PORT_ADC_DATA_BASE_ADDRESS 0x40
#define PORT_DAC_DATA_BASE_ADDRESS 0x60
#define DACPRSTDAT1_ADDRESS 0x16
#define GPO_DATA_15_TO_0_ADDRESS 0x0D
#define GPO_DATA_19_TO_16_ADDRESS 0x0E
#define GPI_DATA_15_TO_0_ADDRESS 0x0B
#define GPI_DATA_19_TO_16_ADDRESS 0x0C

/*
 * declare the struct with pointers to the functions that will read and write
 * via SPI the registers of the MAX11300 device
 */
struct max11300_rw_ops {
   int (*reg_write)(struct max11300_state *st, u8 reg, u16 value);
   int (*reg_read)(struct max11300_state *st, u8 reg, u16 *value);
   int (*reg_read_differential)(struct max11300_state *st, u8 reg, int *value);
};
```

```
/* declare the global structure that will store the info of the device */
struct max11300_state {
   struct device *dev;
   const struct max11300_rw_ops *ops;
   struct gpio_chip gpiochip;
   struct mutex gpio_lock;
   u8 num_ports;
   u8 num_gpios;
   u8 gpio_offset[20];
   u8 gpio_offset_mode[20];
   u8 port_modes[20];
   u8 adc_range[20];
   u8 dac_range[20];
   u8 adc_reference[20];
   u8 adc_samples[20];
   u8 adc_negative_port[20];
   u8 tx_cmd;
   __be16 tx_msg;
   __be16 rx_msg;
};

int max11300_probe(struct device *dev, const char *name,
            const struct max11300_rw_ops *ops);
int max11300_remove(struct device *dev);

#endif /* __DRIVERS_IIO_DAC_max11300_BASE_H__ */
```

Listing 11-7: maxim,max11300.h

```
#ifndef _DT_BINDINGS_MAXIM_MAX11300_H
#define _DT_BINDINGS_MAXIM_MAX11300_H

#define    PORT_MODE_0          0
#define    PORT_MODE_1          1
#define    PORT_MODE_2          2
#define    PORT_MODE_3          3
#define    PORT_MODE_4          4
#define    PORT_MODE_5          5
#define    PORT_MODE_6          6
#define    PORT_MODE_7          7
#define    PORT_MODE_8          8
#define    PORT_MODE_9          9
#define    PORT_MODE_10         10
#define    PORT_MODE_11         11
#define    PORT_MODE_12         12

#define    ADC_SAMPLES_1        0
#define    ADC_SAMPLES_2        1
```

```
#define    ADC_SAMPLES_4         2
#define    ADC_SAMPLES_8         3
#define    ADC_SAMPLES_16        4
#define    ADC_SAMPLES_32        5
#define    ADC_SAMPLES_64        6
#define    ADC_SAMPLES_128       7



/* ADC voltage ranges */
#define    ADC_VOLTAGE_RANGE_NOT_SELECTED      0
#define    ADC_VOLTAGE_RANGE_PLUS10            1  // 0 to +5V range
#define    ADC_VOLTAGE_RANGE_PLUSMINUS5        2  // -5V to +5V range
#define    ADC_VOLTAGE_RANGE_MINUS10           3  // -10V to 0 range
#define    ADC_VOLTAGE_RANGE_PLUS25            4  // 0 to +2.5 range

/* DAC voltage ranges mode 5*/
#define    DAC_VOLTAGE_RANGE_NOT_SELECTED      0
#define    DAC_VOLTAGE_RANGE_PLUS10            1
#define    DAC_VOLTAGE_RANGE_PLUSMINUS5        2
#define    DAC_VOLTAGE_RANGE_MINUS10           3

#endif /* _DT_BINDINGS_MAXIM_MAX11300_H */
```

Listing 11-8: max11300.c

```
#include "max11300-base.h"

#include <linux/bitops.h>
#include <linux/module.h>
#include <linux/of.h>
#include <linux/spi/spi.h>

/* function to write MAX11300 registers */
static int max11300_reg_write(struct max11300_state *st, u8 reg, u16 val)
{
   struct spi_device *spi = container_of(st->dev, struct spi_device, dev);

   struct spi_transfer t[] = {
          {
                 .tx_buf = &st->tx_cmd,
                 .len = 1,
          }, {
                 .tx_buf = &st->tx_msg,
                 .len = 2,
          },
   };
```

```
	/* to transmit via SPI the LSB bit of the command byte must be 0 */
	st->tx_cmd = (reg << 1);

	/*
	 * In little endian CPUs the byte stored in the higher address of
	 * the "val" variable (MSB of the DAC) is stored in the lower address
	 * of the "st->tx_msg" variable using cpu_to_be16()
	 */

	st->tx_msg = cpu_to_be16(val);

	return spi_sync_transfer(spi, t, ARRAY_SIZE(t));
}

/* function to read MAX11300 registers in SE mode */
static int max11300_reg_read(struct max11300_state *st, u8 reg, u16 *value)
{
	struct spi_device *spi = container_of(st->dev, struct spi_device, dev);
	int ret;

	struct spi_transfer t[] = {
		{
			.tx_buf = &st->tx_cmd,
			.len = 1,
		}, {
			.rx_buf = &st->rx_msg,
			.len = 2,
		},
	};

	dev_info(st->dev, "read SE channel\n");

	/* to receive via SPI the LSB bit of the command byte must be 1 */
	st->tx_cmd = ((reg << 1) | 1);

	ret = spi_sync_transfer(spi, t, ARRAY_SIZE(t));
	if (ret < 0)
		return ret;

	/*
	 * In little endian CPUs the first byte (MSB of the ADC) received via
	 * SPI (in BE format) is stored in the lower address of "st->rx_msg"
	 * variable. This byte is copied to the higher address of the "value"
	 * variable using be16_to_cpu(). The second byte received via SPI is
	 * copied from the higher address of "st->rx_msg" to the lower address
	 * of the "value" variable in little endian CPUs.
	 * In big endian CPUs the addresses are not swapped.
	 */
```

```
    *value = be16_to_cpu(st->rx_msg);

    return 0;
}

/* function to read MAX11300 registers in differential mode (2's complement) */
static int max11300_reg_read_differential(struct max11300_state *st, u8 reg,
                                          int *value)
{
    struct spi_device *spi = container_of(st->dev, struct spi_device, dev);
    int ret;

    struct spi_transfer t[] = {
            {
                    .tx_buf = &st->tx_cmd,
                    .len = 1,
            }, {
                    .rx_buf = &st->rx_msg,
                    .len = 2,
            },
    };

    dev_info(st->dev, "read differential channel\n");

    /* to receive LSB of command byte has to be 1 */
    st->tx_cmd = ((reg << 1) | 1);

    ret = spi_sync_transfer(spi, t, ARRAY_SIZE(t));
    if (ret < 0)
            return ret;

    /*
     * extend to an int 2's complement value the received SPI value in 2's
     * complement value, which is stored in the "st->rx_msg" variable
     */
    *value = sign_extend32(be16_to_cpu(st->rx_msg), 11);

    return 0;
}

/*
 * Initialize the struct max11300_rw_ops with read and write
 * callback functions to write/read via SPI from MAX11300 registers
 */
static const struct max11300_rw_ops max11300_rw_ops = {
    .reg_write = max11300_reg_write,
    .reg_read = max11300_reg_read,
    .reg_read_differential = max11300_reg_read_differential,
```

```
};

static int max11300_spi_probe(struct spi_device *spi)
{
   const struct spi_device_id *id = spi_get_device_id(spi);

   return max11300_probe(&spi->dev, id->name, &max11300_rw_ops);
}

static int max11300_spi_remove(struct spi_device *spi)
{
   return max11300_remove(&spi->dev);
}

static const struct spi_device_id max11300_spi_ids[] = {
   { .name = "max11300", },
   {}
};
MODULE_DEVICE_TABLE(spi, max11300_spi_ids);

static const struct of_device_id max11300_of_match[] = {
   { .compatible = "maxim,max11300", },
   {},
};
MODULE_DEVICE_TABLE(of, max11300_of_match);

static struct spi_driver max11300_spi_driver = {
   .driver = {
          .name = "max11300",
          .of_match_table = of_match_ptr(max11300_of_match),
   },
   .probe = max11300_spi_probe,
   .remove = max11300_spi_remove,
   .id_table = max11300_spi_ids,
};
module_spi_driver(max11300_spi_driver);

MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("Maxim max11300 multi-port converters");
MODULE_LICENSE("GPL v2");
```

Listing 11-9: max11300-base.c

```
#include <linux/bitops.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>
#include <linux/module.h>
#include <linux/mutex.h>
#include <linux/of.h>
#include <linux/property.h>

#include <dt-bindings/iio/maxim,max11300.h>

#include "max11300-base.h"

/*
 * struct gpio_chip get callback function.
 * It gets the input value of the GPIO line (0=low, 1=high)
 * accessing to the GPI_DATA registers of max11300
 */
static int max11300_gpio_get(struct gpio_chip *chip, unsigned int offset)
{
   struct max11300_state *st = gpiochip_get_data(chip);
   int ret = 0;
   u16 read_val;
   u8 reg;
   int val;

   mutex_lock(&st->gpio_lock);

   dev_info(st->dev, "The GPIO input is get\n");

   if (st->gpio_offset_mode[offset] == PORT_MODE_3)
   dev_info(st->dev, "the gpio %d cannot be configured in input mode\n",
            offset);

   /* for GPIOs from 16 to 19 ports */
   if (st->gpio_offset[offset] > 0x0F) {
           reg = GPI_DATA_19_TO_16_ADDRESS;
           ret = st->ops->reg_read(st, reg, &read_val);
           if (ret)
                   goto err_unlock;

           val = (int) (read_val);
           val = val << 16;

           if (val & BIT(st->gpio_offset[offset]))
                   val = 1;
           else
```

```
		val = 0;

		mutex_unlock(&st->gpio_lock);
		return val;
	}
	else {
		reg = GPI_DATA_15_TO_0_ADDRESS;
		ret = st->ops->reg_read(st, reg, &read_val);
		if (ret)
			goto err_unlock;

		val = (int) read_val;

		if(val & BIT(st->gpio_offset[offset]))
			val = 1;
		else
			val = 0;

		mutex_unlock(&st->gpio_lock);
		return val;
	}

err_unlock:
	mutex_unlock(&st->gpio_lock);
	return ret;
}

/*
 * struct gpio_chip set callback function.
 * It sets the output value of the GPIO line in
 * GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the GPO_DATA registers of max11300
 */
static void max11300_gpio_set(struct gpio_chip *chip, unsigned int offset,
			     int value)
{
	struct max11300_state *st = gpiochip_get_data(chip);
	u8 reg;
	unsigned int val = 0;

	mutex_lock(&st->gpio_lock);

	dev_info(st->dev, "The GPIO ouput is set\n");

	if (st->gpio_offset_mode[offset] == PORT_MODE_1)
	dev_info(st->dev, "the gpio %d cannot accept this output\n", offset);

	if (value == 1 && (st->gpio_offset[offset] > 0x0F)) {
```

```
        dev_info(st->dev,
            "The GPIO ouput is set high and port_number is %d. Pin is > 0x0F\n",
                st->gpio_offset[offset]);
        val |= BIT(st->gpio_offset[offset]);
        val = val >> 16;
        reg = GPO_DATA_19_TO_16_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 0 && (st->gpio_offset[offset] > 0x0F)) {
        dev_info(st->dev,
            "The GPIO ouput is set low and port_number is %d. Pin is > 0x0F\n",
                st->gpio_offset[offset]);
        val &= ~BIT(st->gpio_offset[offset]);
        val = val >> 16;
        reg = GPO_DATA_19_TO_16_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 1 && (st->gpio_offset[offset] < 0x0F)) {
        dev_info(st->dev,
            "The GPIO ouput is set high and port_number is %d. Pin is < 0x0F\n",
                st->gpio_offset[offset]);
        val |= BIT(st->gpio_offset[offset]);
        reg = GPO_DATA_15_TO_0_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 0 && (st->gpio_offset[offset] < 0x0F)) {
        dev_info(st->dev,
             "The GPIO ouput is set low and port_number is %d. Pin is < 0x0F\n",
                st->gpio_offset[offset]);
        val &= ~BIT(st->gpio_offset[offset]);
        reg = GPO_DATA_15_TO_0_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else
        dev_info(st->dev, "the gpio %d cannot accept this value\n", offset);

    mutex_unlock(&st->gpio_lock);
}

/*
 * struct gpio_chip direction_input callback function.
 * It configures the GPIO port as an input (GPI)
 * writing to the PORT_CFG register of max11300
 */
static int max11300_gpio_direction_input(struct gpio_chip *chip,
                                          unsigned int offset)
{
    struct max11300_state *st = gpiochip_get_data(chip);
```

```
	int ret;
	u8 reg;
	u16 port_mode, val;

	mutex_lock(&st->gpio_lock);

	dev_info(st->dev, "The GPIO is set as an input\n");

	/* get the port number stored in the GPIO offset */
	if (st->gpio_offset_mode[offset] == PORT_MODE_3)
		dev_info(st->dev,
			 "Error.The gpio %d only can be set in output mode\n",
			 offset);

	/* Set the logic 1 input above 2.5V level*/
	val = 0x0fff;

	/* store the GPIO threshold value in the port DAC register */
	reg = PORT_DAC_DATA_BASE_ADDRESS + st->gpio_offset[offset];
	ret = st->ops->reg_write(st, reg, val);
	if (ret)
		goto err_unlock;

	/* Configure the port as GPI */
	reg = PORT_CFG_BASE_ADDRESS + st->gpio_offset[offset];
	port_mode = (1 << 12);
	ret = st->ops->reg_write(st, reg, port_mode);
	if (ret)
		goto err_unlock;

	mdelay(1);

err_unlock:
	mutex_unlock(&st->gpio_lock);

	return ret;
}

/*
 * struct gpio_chip direction_output callback function.
 * It configures the GPIO port as an output (GPO) writing to
 * the PORT_CFG register of max11300 and sets output value of the
 * GPIO line in GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the GPO data registers of max11300
 */
static int max11300_gpio_direction_output(struct gpio_chip *chip,
					  unsigned int offset, int value)
{
```

```
	struct max11300_state *st = gpiochip_get_data(chip);
	int ret;
	u8 reg;
	u16 port_mode, val;

	mutex_lock(&st->gpio_lock);

	dev_info(st->dev, "The GPIO is set as an output\n");

	if (st->gpio_offset_mode[offset] == PORT_MODE_1)
		dev_info(st->dev,
			 "the gpio %d only can be set in input mode\n",
			 offset);

	/* GPIO output high is 3.3V */
	val = 0x0547;

	reg = PORT_DAC_DATA_BASE_ADDRESS + st->gpio_offset[offset];
	ret = st->ops->reg_write(st, reg, val);
	if (ret) {
		mutex_unlock(&st->gpio_lock);
		return ret;
	}
	mdelay(1);
	reg = PORT_CFG_BASE_ADDRESS + st->gpio_offset[offset];
	port_mode = (3 << 12);
	ret = st->ops->reg_write(st, reg, port_mode);
	if (ret) {
		mutex_unlock(&st->gpio_lock);
		return ret;
	}
	mdelay(1);

	mutex_unlock(&st->gpio_lock);

	max11300_gpio_set(chip, offset, value);

	return ret;
}

/*
 * Initialize the MAX11300 gpio controller (struct gpio_chip)
 * and register it to the kernel
 */
static int max11300_gpio_init(struct max11300_state *st)
{
	if (!st->num_gpios)
		return 0;
```

```
    st->gpiochip.label = "gpio-max11300";
    st->gpiochip.base = -1;
    st->gpiochip.ngpio = st->num_gpios;
    st->gpiochip.parent = st->dev;
    st->gpiochip.can_sleep = true;
    st->gpiochip.direction_input = max11300_gpio_direction_input;
    st->gpiochip.direction_output = max11300_gpio_direction_output;
    st->gpiochip.get = max11300_gpio_get;
    st->gpiochip.set = max11300_gpio_set;
    st->gpiochip.owner = THIS_MODULE;

    mutex_init(&st->gpio_lock);

    /* register a gpio_chip */
    return gpiochip_add_data(&st->gpiochip, st);
}

/*
 * Configure the port configuration registers of each port with the values
 * retrieved from the DT properties.These DT values were read and stored in
 * the device global structure using the max11300_alloc_ports() function.
 * The ports in GPIO mode will be configured in the gpiochip.direction_input
 * and gpiochip.direction_output callback functions.
 */
static int max11300_set_port_modes(struct max11300_state *st)
{
    const struct max11300_rw_ops *ops = st->ops;
    int ret;
    unsigned int i;
    u8 reg;
    u16 adc_range, dac_range, adc_reference, adc_samples, adc_negative_port;
    u16 val, port_mode;
    struct iio_dev *iio_dev = iio_priv_to_dev(st);

    mutex_lock(&iio_dev->mlock);

    for (i = 0; i < st->num_ports; i++) {
            switch (st->port_modes[i]) {
            case PORT_MODE_5: case PORT_MODE_6:
                    reg = PORT_CFG_BASE_ADDRESS + i;
                    adc_reference = st->adc_reference[i];
                    port_mode = (st->port_modes[i] << 12);
                    dac_range = (st->dac_range[i] << 8);

                    dev_info(st->dev,
                  "the value of adc cfg addr for channel %d in port mode %d is %x\n",
                            i, st->port_modes[i], reg);
```

```
		if ((st->port_modes[i]) == PORT_MODE_5)
			val = (port_mode | dac_range);
		else
			val = (port_mode | dac_range | adc_reference);

		dev_info(st->dev, "the channel %d is set in port mode %d\n",
			 i, st->port_modes[i]);
		dev_info(st->dev,
		"the value of adc cfg val for channel %d in port mode %d is %x\n",
			 i, st->port_modes[i], val);

		ret = ops->reg_write(st, reg, val);
		if (ret)
			goto err_unlock;

		mdelay(1);
		break;
	case PORT_MODE_7:
		reg = PORT_CFG_BASE_ADDRESS + i;
		port_mode = (st->port_modes[i] << 12);
		adc_range = (st->adc_range[i] << 8);
		adc_reference = st->adc_reference[i];
		adc_samples = (st->adc_samples[i] << 5);

		dev_info(st->dev,
		"the value of adc cfg addr for channel %d in port mode %d is %x\n",
			 i, st->port_modes[i], reg);

		val = (port_mode | adc_range | adc_reference | adc_samples);

		dev_info(st->dev,
			 "the channel %d is set in port mode %d\n",
			 i, st->port_modes[i]);
		dev_info(st->dev,
		 "the value of adc cfg val for channel %d in port mode %d is %x\n",
			 i, st->port_modes[i], val);

		ret = ops->reg_write(st, reg, val);
		if (ret)
			goto err_unlock;

		mdelay(1);

		break;
	case PORT_MODE_8:
		reg = PORT_CFG_BASE_ADDRESS + i;
		port_mode = (st->port_modes[i] << 12);
```

```
                        adc_range = (st->adc_range[i] << 8);
                        adc_reference = st->adc_reference[i];
                        adc_samples = (st->adc_samples[i] << 5);
                        adc_negative_port = st->adc_negative_port[i];

                        dev_info(st->dev,
                     "the value of adc cfg addr for channel %d in port mode %d is %x\n",
                                i, st->port_modes[i], reg);

                        val = (port_mode | adc_range | adc_reference | adc_samples |
adc_negative_port);

                        dev_info(st->dev,
                                "the channel %d is set in port mode %d\n",
                                i, st->port_modes[i]);
                        dev_info(st->dev,
                     "the value of adc cfg val for channel %d in port mode %d is %x\n",
                                i, st->port_modes[i], val);

                        ret = ops->reg_write(st, reg, val);
                        if (ret)
                               goto err_unlock;

                        mdelay(1);
                        break;
                case PORT_MODE_9: case PORT_MODE_10:
                        reg = PORT_CFG_BASE_ADDRESS + i;
                        port_mode = (st->port_modes[i] << 12);
                        adc_range = (st->adc_range[i] << 8);
                        adc_reference = st->adc_reference[i];

                        dev_info(st->dev,
                     "the value of adc cfg addr for channel %d in port mode %d is %x\n",
                                i, st->port_modes[i], reg);

                        val = (port_mode | adc_range | adc_reference);

                        dev_info(st->dev,
                                "the channel %d is set in port mode %d\n",
                                i, st->port_modes[i]);
                        dev_info(st->dev,
                      "the value of adc cfg val for channel %d in port mode %d is %x\n",
                                i, st->port_modes[i], val);

                        ret = ops->reg_write(st, reg, val);
                        if (ret)
                               goto err_unlock;
```

```
			mdelay(1);
			break;
		case PORT_MODE_0:
			dev_info(st->dev,
				"the port %d is set in default port mode_0\n", i);
			break;
		case PORT_MODE_1:
			dev_info(st->dev, "the port %d is set in port mode_1\n", i);
			break;
		case PORT_MODE_3:
			dev_info(st->dev, "the port %d is set in port mode_3\n", i);
			break;
		default:
			dev_info(st->dev, "bad port mode is selected\n");
			return -EINVAL;
		}
	}

err_unlock:
	mutex_unlock(&iio_dev->mlock);
	return ret;
}

/* IIO writing callback function */
static int max11300_write_dac(struct iio_dev *iio_dev,
			      struct iio_chan_spec const *chan,
			      int val, int val2, long mask)
{
	struct max11300_state *st = iio_priv(iio_dev);
	u8 reg;
	int ret;

	reg = (PORT_DAC_DATA_BASE_ADDRESS + chan->channel);

	dev_info(st->dev, "the DAC data register is %x\n", reg);
	dev_info(st->dev, "the value in the DAC data register is %x\n", val);

	switch (mask) {
	case IIO_CHAN_INFO_RAW:
		if (!chan->output)
			return -EINVAL;

		mutex_lock(&iio_dev->mlock);
		ret = st->ops->reg_write(st, reg, val);
		mutex_unlock(&iio_dev->mlock);
		break;
	default:
		return -EINVAL;
```

```
	}

	return ret;
}

/* IIO reading callback function */
static int max11300_read_adc(struct iio_dev *iio_dev,
			     struct iio_chan_spec const *chan,
			     int *val, int *val2, long m)
{
	struct max11300_state *st = iio_priv(iio_dev);
	u16 read_val_se;
	int read_val_dif;
	u8 reg;
	int ret;

	reg = PORT_ADC_DATA_BASE_ADDRESS + chan->channel;

	switch (m) {
	case IIO_CHAN_INFO_RAW:
		mutex_lock(&iio_dev->mlock);

		if (!chan->output && ((chan->address == PORT_MODE_7) || (chan->address
== PORT_MODE_6))) {
			ret = st->ops->reg_read(st, reg, &read_val_se);
			if (ret)
				goto unlock;
			*val = (int) read_val_se;
		}
		else if (!chan->output && (chan->address == PORT_MODE_8)) {
			ret = st->ops->reg_read_differential(st, reg, &read_val_dif);
			if (ret)
				goto unlock;
			*val = read_val_dif;
		}
		else {
			ret = -EINVAL;
			goto unlock;
		}

		ret = IIO_VAL_INT;
		break;
	default:
		ret = -EINVAL;
	}

unlock:
	mutex_unlock(&iio_dev->mlock);
```

```
    return ret;
}

/* Create kernel hooks to read/write IIO sysfs attributes from user space */
static const struct iio_info max11300_info = {
    .read_raw = max11300_read_adc,
    .write_raw = max11300_write_dac,
};

/* DAC with positive voltage range */
static void max11300_setup_port_5_mode(struct iio_dev *iio_dev,
                                       struct iio_chan_spec *chan, bool output,
                                       unsigned int id, unsigned long port_mode)
{
    chan->type = IIO_VOLTAGE;
    chan->indexed = 1;
    chan->address = port_mode;
    chan->output = output;
    chan->channel = id;
    chan->info_mask_separate = BIT(IIO_CHAN_INFO_RAW);
    chan->scan_type.sign = 'u';
    chan->scan_type.realbits = 12;
    chan->scan_type.storagebits = 16;
    chan->scan_type.endianness = IIO_BE;
    chan->extend_name = "mode_5_DAC";
}

/* DAC with positive voltage range */
static void max11300_setup_port_6_mode(struct iio_dev *iio_dev,
                                       struct iio_chan_spec *chan, bool output,
                                       unsigned int id, unsigned long port_mode)
{
    chan->type = IIO_VOLTAGE;
    chan->indexed = 1;
    chan->address = port_mode;
    chan->output = output;
    chan->channel = id;
    chan->info_mask_separate = BIT(IIO_CHAN_INFO_RAW);
    chan->scan_type.sign = 'u';
    chan->scan_type.realbits = 12;
    chan->scan_type.storagebits = 16;
    chan->scan_type.endianness = IIO_BE;
    chan->extend_name = "mode_6_DAC_ADC";
}

/* ADC in SE mode with positive voltage range and straight binary */
static void max11300_setup_port_7_mode(struct iio_dev *iio_dev,
                                       struct iio_chan_spec *chan, bool output,
```

```
                                      unsigned int id, unsigned long port_mode)
{
   chan->type = IIO_VOLTAGE;
   chan->indexed = 1;
   chan->address = port_mode;
   chan->output = output;
   chan->channel = id;
   chan->info_mask_separate = BIT(IIO_CHAN_INFO_RAW);
   chan->scan_type.sign = 'u';
   chan->scan_type.realbits = 12;
   chan->scan_type.storagebits = 16;
   chan->scan_type.endianness = IIO_BE;
   chan->extend_name = "mode_7_ADC";
}

/* ADC in differential mode with 2's complement value */
static void max11300_setup_port_8_mode(struct iio_dev *iio_dev,
                                      struct iio_chan_spec *chan, bool output,
                                      unsigned id, unsigned id2,
                                      unsigned int port_mode)
{
   chan->type = IIO_VOLTAGE;
   chan->differential = 1,
   chan->address = port_mode;
   chan->indexed = 1;
   chan->output = output;
   chan->channel = id;
   chan->channel2 = id2;
   chan->info_mask_separate = BIT(IIO_CHAN_INFO_RAW);
   chan->scan_type.sign = 's';
   chan->scan_type.realbits = 12;
   chan->scan_type.storagebits = 16;
   chan->scan_type.endianness = IIO_BE;
   chan->extend_name = "mode_8_ADC";
}

/*
 * this function will allocate and configure the iio channels of the iio device.
 * It will also read the DT properties of each port (channel) and will store them
 * in the device global structure
 */
static int max11300_alloc_ports(struct max11300_state *st)
{
   unsigned int i, curr_port = 0, num_ports = st->num_ports, port_mode_6_count =
0, offset = 0;
   st->num_gpios = 0;

   /* recover the iio device from the global structure */
```

```
struct iio_dev *iio_dev = iio_priv_to_dev(st);

/* pointer to the storage of the specs of all the iio channels */
struct iio_chan_spec *ports;

/* pointer to struct fwnode_handle that allows a device description object */
struct fwnode_handle *child;

u32 reg, tmp;
int ret;

/*
 * walks for each MAX11300 child node from the DT, if there is an error
 * then walks to the following one (continue)
 */
device_for_each_child_node(st->dev, child) {
	ret = fwnode_property_read_u32(child, "reg", &reg);
	if (ret || reg >= ARRAY_SIZE(st->port_modes))
		continue;

	/*
	 * store the value of the DT "port,mode" property in the global struct
	 * to know the mode of each port in other functions of the driver
	 */
	ret = fwnode_property_read_u32(child, "port-mode", &tmp);
	if (!ret)
		st->port_modes[reg] = tmp;

	/* all the DT nodes should include the port-mode property */
	else {
		dev_info(st->dev, "port mode is not found\n");
		continue;
	}

	/*
	 * you will store other DT properties depending
	 * of the used "port,mode" property
	 */
	switch (st->port_modes[reg]) {
	case PORT_MODE_7:
		ret = fwnode_property_read_u32(child, "adc-range", &tmp);
		if (!ret)
			st->adc_range[reg] = tmp;
		else
			dev_info(st->dev, "Get default ADC range\n");

		ret = fwnode_property_read_u32(child, "AVR", &tmp);
		if (!ret)
```

```
                st->adc_reference[reg] = tmp;
        else
                dev_info(st->dev,
                         "Get default internal ADC reference\n");

        ret = fwnode_property_read_u32(child, "adc-samples", &tmp);
        if (!ret)
                st->adc_samples[reg] = tmp;
        else
                dev_info(st->dev, "Get default internal ADC sampling\n");

        dev_info(st->dev, "the channel %d is set in port mode %d\n",
                 reg, st->port_modes[reg]);
        break;
case PORT_MODE_8:
        ret = fwnode_property_read_u32(child, "adc-range", &tmp);
        if (!ret)
                st->adc_range[reg] = tmp;
        else
                dev_info(st->dev, "Get default ADC range\n");

        ret = fwnode_property_read_u32(child, "AVR", &tmp);
        if (!ret)
                st->adc_reference[reg] = tmp;
        else
                dev_info(st->dev,
                         "Get default internal ADC reference\n");

        ret = fwnode_property_read_u32(child, "adc-samples", &tmp);
        if (!ret)
                st->adc_samples[reg] = tmp;
        else
                dev_info(st->dev, "Get default internal ADC sampling\n");

        ret = fwnode_property_read_u32(child, "negative-input", &tmp);
        if (!ret)
                st->adc_negative_port[reg] = tmp;
        else {
                dev_info(st->dev,
                         "Bad value for negative ADC channel\n");
                return -EINVAL;
        }

        dev_info(st->dev, "the channel %d is set in port mode %d\n",
                 reg, st->port_modes[reg]);
        break;
case PORT_MODE_9: case PORT_MODE_10:
        ret = fwnode_property_read_u32(child, "adc-range", &tmp);
```

```
        if (!ret)
                st->adc_range[reg] = tmp;
        else
                dev_info(st->dev, "Get default ADC range\n");

        ret = fwnode_property_read_u32(child, "AVR", &tmp);
        if (!ret)
                st->adc_reference[reg] = tmp;
        else
                dev_info(st->dev,
                         "Get default internal ADC reference\n");
        dev_info(st->dev, "the channel %d is set in port mode %d\n",
                 reg, st->port_modes[reg]);
        break;
case PORT_MODE_5: case PORT_MODE_6:
        ret = fwnode_property_read_u32(child, "dac-range", &tmp);
        if (!ret)
        st->dac_range[reg] = tmp;
        else
                dev_info(st->dev, "Get default DAC range\n");

        /*
         * A port in mode 6 will generate two IIO sysfs entries,
         * one for writing the DAC port, and another for reading
         * the ADC port
         */
        if ((st->port_modes[reg]) == PORT_MODE_6) {
                ret = fwnode_property_read_u32(child, "AVR", &tmp);
                if (!ret)
                        st->adc_reference[reg] = tmp;
                else
                        dev_info(st->dev,
                                 "Get default internal ADC reference\n");

                /*
                 * get the number of ports set in mode_6 to allocate
                 * space for the related iio channels
                 */
                port_mode_6_count++;
                dev_info(st->dev, "there are %d channels in mode_6\n",
                         port_mode_6_count);
        }

        dev_info(st->dev, "the channel %d is set in port mode %d\n",
                 reg, st->port_modes[reg]);
        break;
/* The port is configured as a GPI in the DT */
case PORT_MODE_1:
```

```
        dev_info(st->dev, "the channel %d is set in port mode %d\n",
                reg, st->port_modes[reg]);

        /*
         * link the gpio offset with the port number,
         * starting with offset = 0
         */
        st->gpio_offset[offset] = reg;

        /*
         * store the port_mode for each gpio offset,
         * starting with offset = 0
         */
        st->gpio_offset_mode[offset] = PORT_MODE_1;

        dev_info(st->dev,
            "the gpio number %d is using the gpio offset number %d\n",
                st->gpio_offset[offset], offset);

        /*
         * increment the gpio offset and number
         * of configured ports as GPIOs
        */
        offset++;
        st->num_gpios++;
        break;
/* The port is configured as a GPO in the DT */
case PORT_MODE_3:
        dev_info(st->dev, "the channel %d is set in port mode %d\n",
                reg, st->port_modes[reg]);

        /*
         * link the gpio offset with the port number,
         * starting with offset = 0
         */
        st->gpio_offset[offset] = reg;

        /*
         * store the port_mode for each gpio offset,
         * starting with offset = 0
        */
        st->gpio_offset_mode[offset] = PORT_MODE_3;

        dev_info(st->dev,
             "the gpio number %d is using the gpio offset number %d\n",
                st->gpio_offset[offset], offset);
```

```
			/*
			 * increment the gpio offset and
			 * number of configured ports as GPIOs
			 */
			offset++;
			st->num_gpios++;
			break;
		case PORT_MODE_0:
			dev_info(st->dev,
				 "the channel %d is set in default port mode_0\n", reg);
			break;
		default:
			dev_info(st->dev, "bad port mode for channel %d\n", reg);
		}

	}

	/*
	 * Allocate space for the storage of all the IIO channels specs.
	 * Returns a pointer to this storage
	 */
	ports = devm_kcalloc(st->dev, num_ports + port_mode_6_count,
			     sizeof(*ports), GFP_KERNEL);
	if (!ports)
		return -ENOMEM;

	/*
	 * i is the number of the channel, &ports[curr_port] is a pointer variable that
	 * will store the "iio_chan_spec structure" address of each port
	 */
	for (i = 0; i < num_ports; i++) {
		switch (st->port_modes[i]) {
		case PORT_MODE_5:
			dev_info(st->dev, "the port %d is configured as MODE 5\n", i);
			max11300_setup_port_5_mode(iio_dev, &ports[curr_port],
						   true, i, PORT_MODE_5); // true = out
			curr_port++;
			break;
		case PORT_MODE_6:
			dev_info(st->dev, "the port %d is configured as MODE 6\n", i);
			max11300_setup_port_6_mode(iio_dev, &ports[curr_port],
						   true, i, PORT_MODE_6); // true = out
			curr_port++;
			max11300_setup_port_6_mode(iio_dev, &ports[curr_port],
						   false, i, PORT_MODE_6); // false = in
			curr_port++;
			break;
		case PORT_MODE_7:
```

```
                        dev_info(st->dev, "the port %d is configured as MODE 7\n", i);
                        max11300_setup_port_7_mode(iio_dev, &ports[curr_port],
                                        false, i, PORT_MODE_7); // false = in
                        curr_port++;
                        break;
                case PORT_MODE_8:
                        dev_info(st->dev, "the port %d is configured as MODE 8\n", i);
                        max11300_setup_port_8_mode(iio_dev, &ports[curr_port],
                                        false, i, st->adc_negative_port[i],
                                        PORT_MODE_8); // false = in
                        curr_port++;
                        break;
                case PORT_MODE_0:
                        dev_info(st->dev,
                                "the channel is set in default port mode_0\n");
                        break;
                case PORT_MODE_1:
                        dev_info(st->dev, "the channel %d is set in port mode_1\n", i);
                        break;
                case PORT_MODE_3:
                        dev_info(st->dev, "the channel %d is set in port mode_3\n", i);
                        break;
                default:
                        dev_info(st->dev, "bad port mode for channel %d\n", i);
                }
        }

        iio_dev->num_channels = curr_port;
        iio_dev->channels = ports;

        return 0;
}

int max11300_probe(struct device *dev, const char *name,
                const struct max11300_rw_ops *ops)
{

        /* create an iio device */
        struct iio_dev *iio_dev;

        /* create the global structure that will store the info of the device */
        struct max11300_state *st;

        u16 write_val;
        u16 read_val;
        u8 reg;
        int ret;
```

```
write_val = 0;

dev_info(dev, "max11300_probe() function is called\n");

/* allocates memory fot the IIO device */
iio_dev = devm_iio_device_alloc(dev, sizeof(*st));
if (!iio_dev)
       return -ENOMEM;

/* link the global data structure with the iio device */
st = iio_priv(iio_dev);

/* store in the global structure the spi device */
st->dev = dev;

/*
 * store in the global structure the pointer to the
 * MAX11300 SPI read and write functions
 */
st->ops = ops;

/* setup the number of ports of the MAX11300 device */
st->num_ports = 20;

/* link the spi device with the iio device */
dev_set_drvdata(dev, iio_dev);


iio_dev->dev.parent = dev;
iio_dev->name = name;

/*
 * store the address of the iio_info structure,
 * which contains pointer variables
 * to IIO write/read callbacks
 */
iio_dev->info = &max11300_info;
iio_dev->modes = INDIO_DIRECT_MODE;

/* reset the MAX11300 device */
reg = DCR_ADDRESS;
dev_info(st->dev, "the value of DCR_ADDRESS is %x\n", reg);
write_val = RESET;
dev_info(st->dev, "the value of reset is %x\n", write_val);
ret = ops->reg_write(st, reg, write_val);
if (ret != 0)
       goto error;
```

```
	/* return MAX11300 Device ID */
	reg = 0x00;
	ret = ops->reg_read(st, reg, &read_val);
	if (ret != 0)
		goto error;
	dev_info(st->dev, "the value of device ID is %x\n", read_val);

	/* Configure DACREF and ADCCTL */
	reg = DCR_ADDRESS;
	write_val = (DCR_ADCCTL_CONTINUOUS_SWEEP | DCR_DACREF);
	dev_info(st->dev, "the value of DACREF_CONT_SWEEP is %x\n", write_val);
	ret = ops->reg_write(st, reg, write_val);
	udelay(200);
	if (ret)
		goto error;
	dev_info(dev, "the setup of the device is done\n");

	/* Configure the IIO channels of the device */
	ret = max11300_alloc_ports(st);
	if (ret)
		goto error;

	ret = max11300_set_port_modes(st);
	if (ret)
		goto error_reset_device;

	ret = iio_device_register(iio_dev);
	if (ret)
		goto error;

	ret = max11300_gpio_init(st);
	if (ret)
		goto error_dev_unregister;

	return 0;

error_dev_unregister:
	iio_device_unregister(iio_dev);

error_reset_device:
	/* reset the device */
	reg = DCR_ADDRESS;
	write_val = RESET;
	ret = ops->reg_write(st, reg, write_val);
	if (ret != 0)
		return ret;

error:
```

```
   return ret;
}
EXPORT_SYMBOL_GPL(max11300_probe);

int max11300_remove(struct device *dev)
{
   struct iio_dev *iio_dev = dev_get_drvdata(dev);

   iio_device_unregister(iio_dev);

   return 0;
}
EXPORT_SYMBOL_GPL(max11300_remove);

MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("Maxim max11300 multi-port converters");
MODULE_LICENSE("GPL v2");
```

LAB 11.5 driver demonstration

Libgpiod provides a C library and simple tools for interacting with the Linux GPIO character devices. The GPIO sysfs interface is deprecated from Linux 4.8 for these libgpiod tools. The C library encapsulates the ioctl() calls and data structures using a straightforward API. For more information, see: https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/about/

Connect you Raspberry Pi 4 to the Internet and download libgpiod library and tools:

```
root@raspberrypi:/home# sudo apt-get install gpiod libgpiod-dev libgpiod-doc
```

The tools provided with libgpiod allow accessing the GPIO driver from the command line. There are six commands in libgpiod tools:

- **gpiodetect**: list all gpiochips present on the system, their names, labels, and number of GPIO lines. In the lab, the MAX11300 gpio chip will appear with the name of gpiochip10.
- **gpioinfo:** list all lines of specified gpiochips, their names, consumers, direction, active state, and additional flags.
- **gpioget:** read values of specified GPIO lines. This tool will call to the gpiochip.direction_input and gpiochip.get callback functions declared in the struct gpio_chip of the driver.
- **gpioset:** set values of specified GPIO lines, potentially keep the lines exported and wait until timeout, user input or signal. This tool will call to the gpiochip.direction_output callback function declared in the struct gpio_chip of the driver.
- **gpiofind:** find the gpiochip name and line offset given the line name.

- **gpiomon:** wait for events on GPIO lines, specify which events to watch, how many events to process before exiting or if the events should be reported to the console.

Download the linux_5.4_rpi4_drivers.zip file from the github of the book and unzip it in the home folder of your Linux host:

```
PC:~$ cd ~/linux_5.4_rpi4_drivers/linux_5.4_max11300_driver/
```

Compile and deploy the drivers to the Raspberry Pi 4 Model B board:

```
~/linux_5.4_rpi4_drivers/linux_5.4_max11300_driver$ make
```

```
~/linux_5.4_rpi4_drivers/linux_5.4_max11300_driver$ make deploy
```

Follow the next instructions to test the driver:

```
/* load the module */
root@raspberrypi:/home# insmod max11300-base.ko
[   49.513538] max11300_base: loading out-of-tree module taints kernel.
root@raspberrypi:/home# insmod max11300.ko
[   52.983020] max11300 spi0.0: max11300_probe() function is called
[   52.989221] max11300 spi0.0: the value of DCR_ADDRESS is 10
[   52.994896] max11300 spi0.0: the value of reset is 8000
[   53.000313] max11300 spi0.0: read SE channel
[   53.004977] max11300 spi0.0: the value of device ID is 424
[   53.010607] max11300 spi0.0: the value of DACREF_CONT_SWEEP is 43
[   53.017255] max11300 spi0.0: the setup of the device is done
[   53.023122] max11300 spi0.0: the channel 0 is set in port mode 7
[   53.029286] max11300 spi0.0: the channel 1 is set in port mode 7
[   53.035409] max11300 spi0.0: the channel 2 is set in port mode 5
[   53.041572] max11300 spi0.0: the channel 3 is set in port mode 5
[   53.047735] max11300 spi0.0: the channel 4 is set in port mode 8
[   53.053858] max11300 spi0.0: the channel 5 is set in port mode 9
[   53.060011] max11300 spi0.0: there are 1 channels in mode_6
[   53.065680] max11300 spi0.0: the channel 6 is set in port mode 6
[   53.071829] max11300 spi0.0: the channel 7 is set in port mode 1
[   53.077972] max11300 spi0.0: the gpio number 7 is using the gpio offset number
0
[   53.085503] max11300 spi0.0: the channel 8 is set in port mode 3
[   53.091644] max11300 spi0.0: the gpio number 8 is using the gpio offset number
1
[   53.099205] max11300 spi0.0: the channel 9 is set in default port mode_0
[   53.106030] max11300 spi0.0: the channel 10 is set in default port mode_0
[   53.112975] max11300 spi0.0: the channel 11 is set in default port mode_0
[   53.119919] max11300 spi0.0: the channel 12 is set in default port mode_0
[   53.126832] max11300 spi0.0: the channel 13 is set in default port mode_0
[   53.133777] max11300 spi0.0: the channel 14 is set in default port mode_0
[   53.140721] max11300 spi0.0: the channel 15 is set in default port mode_0
[   53.147666] max11300 spi0.0: the channel 16 is set in default port mode_0
[   53.154583] max11300 spi0.0: the channel 17 is set in default port mode_0
```

```
[   53.161529] max11300 spi0.0: the channel 18 is set in port mode 1
[   53.167762] max11300 spi0.0: the gpio number 18 is using the gpio offset number
2
[   53.175385] max11300 spi0.0: the channel 19 is set in port mode 3
[   53.181617] max11300 spi0.0: the gpio number 19 is using the gpio offset number
3
[   53.189305] max11300 spi0.0: the port 0 is configured as MODE 7
[   53.195330] max11300 spi0.0: the port 1 is configured as MODE 7
[   53.201367] max11300 spi0.0: the port 2 is configured as MODE 5
[   53.207389] max11300 spi0.0: the port 3 is configured as MODE 5
[   53.213394] max11300 spi0.0: the port 4 is configured as MODE 8
[   53.219415] max11300 spi0.0: bad port mode for channel 5
[   53.224804] max11300 spi0.0: the port 6 is configured as MODE 6
[   53.230823] max11300 spi0.0: the channel 7 is set in port mode_1
[   53.236917] max11300 spi0.0: the channel 8 is set in port mode_3
[   53.243024] max11300 spi0.0: the channel is set in default port mode_0
[   53.249660] max11300 spi0.0: the channel is set in default port mode_0
[   53.256284] max11300 spi0.0: the channel is set in default port mode_0
[   53.262919] max11300 spi0.0: the channel is set in default port mode_0
[   53.269555] max11300 spi0.0: the channel is set in default port mode_0
[   53.276177] max11300 spi0.0: the channel is set in default port mode_0
[   53.282813] max11300 spi0.0: the channel is set in default port mode_0
[   53.289449] max11300 spi0.0: the channel is set in default port mode_0
[   53.296071] max11300 spi0.0: the channel is set in default port mode_0
[   53.302707] max11300 spi0.0: the channel 18 is set in port mode_1
[   53.308901] max11300 spi0.0: the channel 19 is set in port mode_3
[   53.315085] max11300 spi0.0: the value of adc cfg addr for channel 0 in port
mode 7 is 20
[   53.323408] max11300 spi0.0: the channel 0 is set in port mode 7
[   53.329521] max11300 spi0.0: the value of adc cfg val for channel 0 in port
mode 7 is 7100
[   53.338958] max11300 spi0.0: the value of adc cfg addr for channel 1 in port
mode 7 is 21
[   53.347259] max11300 spi0.0: the channel 1 is set in port mode 7
[   53.353373] max11300 spi0.0: the value of adc cfg val for channel 1 in port
mode 7 is 71e0
[   53.362803] max11300 spi0.0: the value of adc cfg addr for channel 2 in port
mode 5 is 22
[   53.371116] max11300 spi0.0: the channel 2 is set in port mode 5
[   53.377210] max11300 spi0.0: the value of adc cfg val for channel 2 in port
mode 5 is 5100
[   53.386634] max11300 spi0.0: the value of adc cfg addr for channel 3 in port
mode 5 is 23
[   53.394949] max11300 spi0.0: the channel 3 is set in port mode 5
[   53.401056] max11300 spi0.0: the value of adc cfg val for channel 3 in port
mode 5 is 5100
[   53.410478] max11300 spi0.0: the value of adc cfg addr for channel 4 in port
mode 8 is 24
```

```
[   53.418791] max11300 spi0.0: the channel 4 is set in port mode 8
[   53.424886] max11300 spi0.0: the value of adc cfg val for channel 4 in port
mode 8 is 8105
[   53.434309] max11300 spi0.0: the value of adc cfg addr for channel 5 in port
mode 9 is 25
[   53.442621] max11300 spi0.0: the channel 5 is set in port mode 9
[   53.448728] max11300 spi0.0: the value of adc cfg val for channel 5 in port
mode 9 is 9100
[   53.458140] max11300 spi0.0: the value of adc cfg addr for channel 6 in port
mode 6 is 26
[   53.466438] max11300 spi0.0: the channel 6 is set in port mode 6
[   53.472543] max11300 spi0.0: the value of adc cfg val for channel 6 in port
mode 6 is 6100
[   53.481969] max11300 spi0.0: the port 7 is set in port mode_1
[   53.487818] max11300 spi0.0: the port 8 is set in port mode_3
[   53.493647] max11300 spi0.0: the port 9 is set in default port mode_0
[   53.500195] max11300 spi0.0: the port 10 is set in default port mode_0
[   53.506819] max11300 spi0.0: the port 11 is set in default port mode_0
[   53.513454] max11300 spi0.0: the port 12 is set in default port mode_0
[   53.520090] max11300 spi0.0: the port 13 is set in default port mode_0
[   53.526712] max11300 spi0.0: the port 14 is set in default port mode_0
[   53.533347] max11300 spi0.0: the port 15 is set in default port mode_0
[   53.539983] max11300 spi0.0: the port 16 is set in default port mode_0
[   53.546605] max11300 spi0.0: the port 17 is set in default port mode_0
[   53.553241] max11300 spi0.0: the port 18 is set in port mode_1
[   53.559171] max11300 spi0.0: the port 19 is set in port mode_3
root@raspberrypi:/home#

root@raspberrypi:/home# cd /sys/bus/iio/devices/iio:device0

/* check the IIO sysfs entries under the IIO MAX11300 device */
root@raspberrypi:/sys/bus/iio/devices/iio:device0# ls
dev                                   name                             power
in_voltage0_mode_7_ADC_raw            of_node                          subsystem
in_voltage1_mode_7_ADC_raw            out_voltage2_mode_5_DAC_raw      uevent
in_voltage4-voltage5_mode_8_ADC_raw   out_voltage3_mode_5_DAC_raw
in_voltage6_mode_6_DAC_ADC_raw        out_voltage6_mode_6_DAC_ADC_raw
```

Connect port2 (DAC) to port0 (ADC)

```
/* write to the port2 (DAC) */
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 500 >
out_voltage2_mode_5_DAC_raw
[  262.167664] max11300 spi0.0: the DAC data register is 62
[  262.173083] max11300 spi0.0: the value in the DAC data register is 1f4

/* read the port0 (ADC) */
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_voltage0_mode_7_ADC_raw
[  272.073718] max11300 spi0.0: read SE channel
```

connect port2 (DAC) to port4 (ADC differential positive) & port3 (DAC) to port 5 (ADC differential negative)

```
/* set 5V output in the port2 (DAC) */
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 2047 >
out_voltage2_mode_5_DAC_raw
[  402.617682] max11300 spi0.0: the DAC data register is 62
[  402.623100] max11300 spi0.0: the value in the DAC data register is 7ff

/* set 2.5V in the port3 (DAC) */
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 1024 >
out_voltage2_mode_5_DAC_raw
[  426.497655] max11300 spi0.0: the DAC data register is 62
[  426.503071] max11300 spi0.0: the value in the DAC data register is 400

/* read differential input (port4_port5): 2.5V */
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_voltage4-
voltage5_mode_8_ADC_raw
[  455.593738] max11300 spi0.0: read differential channel
512

/* set DAC and read ADC in port mode 6 */
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 1024 >
out_voltage6_mode_6_DAC_ADC_raw
[  535.557710] max11300 spi0.0: the DAC data register is 66
[  535.563129] max11300 spi0.0: the value in the DAC data register is 400
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat
in_voltage6_mode_6_DAC_ADC_raw
[  545.983702] max11300 spi0.0: read SE channel
1022

/* check the gpio chip controllers */
root@raspberrypi:/home# ls -l /dev/gpiochip*
crw-rw---- 1 root gpio 254, 0 nov 22 10:40 /dev/gpiochip0
crw-rw---- 1 root gpio 254, 1 nov 22 10:40 /dev/gpiochip1
crw-rw---- 1 root gpio 254, 2 nov 22 10:59 /dev/gpiochip2

/* Print information of all the lines of the gpiochip2 */
root@raspberrypi:/home# gpioinfo gpiochip2
gpiochip2 - 4 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
        line   2:      unnamed       unused   input  active-high
        line   3:      unnamed       unused   input  active-high
```

connect port19 (GPO) to port 18 (GPI)

```
/* Set port19 (GPO) to high */
root@raspberrypi:/home# gpioset gpiochip2 3=1
```

```
[ 1300.382362] max11300 spi0.0: The GPIO is set as an output
[ 1300.390173] max11300 spi0.0: The GPIO ouput is set
[ 1300.395099] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F

/* Read port 18 (GPI) */
root@raspberrypi:/home# gpioget gpiochip2 2
[ 1351.003501] max11300 spi0.0: The GPIO is set as an input
[ 1351.010218] max11300 spi0.0: The GPIO input is get
[ 1351.015100] max11300 spi0.0: read SE channel
1

/* Set port19 (GPO) to low */
root@raspberrypi:/home# gpioset gpiochip2 3=0
[ 1371.353884] max11300 spi0.0: The GPIO is set as an output
[ 1371.361644] max11300 spi0.0: The GPIO ouput is set
[ 1371.366573] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F

/* Read port 18 (GPI) */
root@raspberrypi:/home# gpioget gpiochip2 2
[ 1375.553853] max11300 spi0.0: The GPIO is set as an input
[ 1375.560577] max11300 spi0.0: The GPIO input is get
[ 1375.565458] max11300 spi0.0: read SE channel
0
```

connect port19 (GPO) to port 7 (GPI)

```
/* Set port19 (GPO) to high */
root@raspberrypi:/home# gpioset gpiochip2 3=1
[ 1466.426732] max11300 spi0.0: The GPIO is set as an output
[ 1466.434538] max11300 spi0.0: The GPIO ouput is set
[ 1466.439463] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F

/* Read port7 (GPI) */
root@raspberrypi:/home# gpioget gpiochip2 0
[ 1487.107109] max11300 spi0.0: The GPIO is set as an input
[ 1487.113730] max11300 spi0.0: The GPIO input is get
[ 1487.118612] max11300 spi0.0: read SE channel
1

/* Set port19 (GPO) to low */
root@raspberrypi:/home# gpioset gpiochip2 3=0
[ 1511.977771] max11300 spi0.0: The GPIO is set as an output
[ 1511.985530] max11300 spi0.0: The GPIO ouput is set
[ 1511.990454] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F

/* Read port7 (GPI) */
root@raspberrypi:/home# gpioget gpiochip2 0
```

```
[ 1516.137865] max11300 spi0.0: The GPIO is set as an input
[ 1516.144490] max11300 spi0.0: The GPIO input is get
[ 1516.149372] max11300 spi0.0: read SE channel
0

connect port8 (GPO) to port 7 (GPI)

/* Set port8 (GPO) to high */
root@raspberrypi:/home# gpioset gpiochip2 1=1
[   91.824390] max11300 spi0.0: The GPIO is set as an output
[   91.832066] max11300 spi0.0: The GPIO ouput is set
[   91.836948] max11300 spi0.0: The GPIO ouput is set high and port_number is 8.
Pin is < 0x0F

/* Read port7 (GPI) */
root@raspberrypi:/home# gpioget gpiochip2 0
[  106.667646] max11300 spi0.0: The GPIO is set as an input
[  106.674198] max11300 spi0.0: The GPIO input is get
[  106.679131] max11300 spi0.0: read SE channel
1

/* Set port8 (GPO) to low */
root@raspberrypi:/home# gpioset gpiochip2 1=0
[  127.445175] max11300 spi0.0: The GPIO is set as an output
[  127.452866] max11300 spi0.0: The GPIO ouput is set
[  127.457816] max11300 spi0.0: The GPIO ouput is set low and port_number is 8.
Pin is < 0x0F

/* Read port7 (GPI) */
root@raspberrypi:/home# gpioget gpiochip2 0
[  130.235012] max11300 spi0.0: The GPIO is set as an input
[  130.241708] max11300 spi0.0: The GPIO input is get
[  130.246590] max11300 spi0.0: read SE channel
0

/* check the new direction of the gpio lines */
root@Raspberry Pi 4 Model B :~# gpioinfo gpiochip2
gpiochip2 - 4 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused  output  active-high
        line   2:      unnamed       unused   input  active-high
        line   3:      unnamed       unused   input  active-high

/* remove the module */
root@raspberrypi:/home# rmmod max11300.ko
root@raspberrypi:/home# rmmod max11300-base.ko
```

In this section, you have seen how to control GPIOs using the tools provided with libgpiod. In the next section, you will see how to write applications to control GPIOs by using two different methods. The first method will control the GPIO by using a device node and the second method will control the GPIO by using the functions of the libgpiod library.

GPIO control through character device

The Chapter 5 of this book explains how to write GPIO user drivers that control GPIOs using the new GPIO descriptor interface included in the GPIOlib framework. This descriptor interface identifies each GPIO through a gpio_desc structure.

GPIOlib is a framework that provides an internal Linux kernel API for managing and configuring GPIOs, acting as a bridge between the Linux GPIO controller drivers and the Linux GPIO user drivers. Writing Linux drivers for devices using GPIOs is a good practice, but you can prefer to control the GPIOs from user space. GPIOlib also provides access to the user space APIs that control the GPIOs through ioctl calls on /dev/gpiochipX char device files, where X is the number of the GPIO bank.

Until the launching of Linux kernel 4.8, the GPIOs were accessed via sysfs (/sys/class/gpio) method, but after this release, there are new interfaces based on a char device. The sysfs interface is deprecated, and is highly recommended to use the new interface. These are some of the advantages of using the new character device user API:

- One device file for each gpiochip:

  /dev/gpiochip0, /dev/gpiochip1, /dev/gpiochipX…

- Similar to other kernel interfaces: ioctl() + poll() + read() + close()
- Possible to set/read multiple GPIOs at once.
- Possible to find GPIO lines by name.

The following application toggles ten times the port19 of the PIXI™ CLICK board. The port19 GPIO can be connected to the red LED on the Color click eval board (https://www.mikroe.com/color-click), to see the red LED blinking.

Send the application to the Raspberry Pi 4 Model B board and compile the application on the Pi:

```
~/linux_5.4_rpi4_drivers/linux_5.4_max11300_driver/application_code$ scp
gpio_device_app.c root@10.0.0.10:/home/
```

```
root@raspberrypi:/home# gcc -o gpio_device_app gpio_device_app.c
```

Finally, execute the application on the target. You can see the red LED flashes!

```
root@raspberrypi:/home# ./gpio_device_app
[  387.963017] max11300 spi0.0: The GPIO is set as an output
[  387.970755] max11300 spi0.0: The GPIO ouput is set
```

```
[  387.975638] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[  387.985031] max11300 spi0.0: The GPIO ouput is set
[  387.989977] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[  388.998930] max11300 spi0.0: The GPIO ouput is set
[  389.003817] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[  390.012625] max11300 spi0.0: The GPIO ouput is set
[  390.017547] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[  391.026219] max11300 spi0.0: The GPIO ouput is set
[  391.031142] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[  392.039912] max11300 spi0.0: The GPIO ouput is set
[  392.044797] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[  393.053507] max11300 spi0.0: The GPIO ouput is set
[  393.058435] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[  394.067208] max11300 spi0.0: The GPIO ouput is set
[  394.072145] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[  395.080982] max11300 spi0.0: The GPIO ouput is set
[  395.085867] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[  396.094677] max11300 spi0.0: The GPIO ouput is set
[  396.099601] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[  397.108285] max11300 spi0.0: The GPIO ouput is set
[  397.113168] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
```

Listing 11-10: gpio_device_app.c

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <linux/gpio.h>
#include <sys/ioctl.h>

/* configure port19 as an output and flash an LED */

#define DEVICE_GPIO "/dev/gpiochip2"
```

```
int main(int argc, char *argv[])
{
    int fd;
    int ret;
    int flash = 10;

    struct gpiohandle_data data;
    struct gpiohandle_request req;

    /* open gpio device */
    fd = open(DEVICE_GPIO, 2);
    if (fd < 0) {
        fprintf(stderr, "Failed to open %s\n", DEVICE_GPIO);
        return -1;
    }

    /* request GPIO line 3 as an output (red LED) */
    req.lineoffsets[0] = 3;
    req.lines = 1;
    req.flags = GPIOHANDLE_REQUEST_OUTPUT;
    strcpy(req.consumer_label, "led_gpio_port19");

    ret = ioctl(fd, GPIO_GET_LINEHANDLE_IOCTL, &req);
    if (ret < 0) {
        printf("ERROR get line handle IOCTL (%d)\n", ret);
        if (close(fd) == -1)
          perror("Failed to close GPIO char device");
        return ret;
    }

    /* start the led_red with off state */
    data.values[0] = 1;

    for (int i=0; i < flash; i++) {
        /* toggle LED */
        data.values[0] = !data.values[0];
        ret = ioctl(req.fd, GPIOHANDLE_SET_LINE_VALUES_IOCTL, &data);
        if (ret < 0) {
          fprintf(stderr, "Failed to issue %s (%d)\n",
"GPIOHANDLE_SET_LINE_VALUES_IOCTL", ret);
          if (close(req.fd) == -1)
                 perror("Failed to close GPIO line");
          if (close(fd) == -1)
                 perror("Failed to close GPIO char device");
          return ret;

        }
        sleep(1);
```

```
    }

    /* close gpio line */
    ret = close(req.fd);
    if (ret == -1)
          perror("Failed to close GPIO line");

    /* close gpio device */
    ret = close(fd);
        if (ret == -1)
          perror("Failed to close GPIO char device");

    return ret;
}
```

GPIO control through gpiolibd library

In this section, you will see how to control GPIOs using the functions of the libgpiod library.

The following libgpiod_app application has the same behaviour than the gpio_device_app one, toggling ten times the port19 connected to the red LED on the Color click eval board, but this time you will use the libgpiod library instead of the "gpio char device" method to control the red LED.

Send the application to the Raspberry Pi 4 Model B board:

```
~/linux_5.4_rpi4_drivers/linux_5.4_max11300_driver/application_code$ scp
libgpiod_max11300_app.c root@10.0.0.10:/home/
```

Compile the application in the Raspberry Pi 4 Model B board:

```
root@raspberrypi:/home# gcc -o libgpiod_max11300_app -lgpiod
libgpiod_max11300_app.c
```

Finally, execute the compiled application on the target. You can see the red LED flashes!

```
root@raspberrypi:/home# ./libgpiod_max11300_app
[  897.034026] max11300 spi0.0: The GPIO is set as an output
[  897.041828] max11300 spi0.0: The GPIO ouput is set
[  897.046711] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[  897.060675] max11300 spi0.0: The GPIO ouput is set
[  897.065562] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[  898.077778] max11300 spi0.0: The GPIO ouput is set
[  898.082668] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[  899.094982] max11300 spi0.0: The GPIO ouput is set
[  899.099920] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
```

```
[  900.112112] max11300 spi0.0: The GPIO ouput is set
[  900.117002] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[  901.129335] max11300 spi0.0: The GPIO ouput is set
[  901.134223] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[  902.146373] max11300 spi0.0: The GPIO ouput is set
[  902.151310] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[  903.160406] max11300 spi0.0: The GPIO ouput is set
[  903.165292] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[  904.174362] max11300 spi0.0: The GPIO ouput is set
[  904.179291] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[  905.191664] max11300 spi0.0: The GPIO ouput is set
[  905.196553] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[  906.210534] max11300 spi0.0: The GPIO ouput is set
[  906.215423] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
```

Listing 11-11: libgpiod_max11300_app.c

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <gpiod.h>

int main(int argc, char *argv[])
{
   struct gpiod_chip *output_chip;
   struct gpiod_line *output_line;
   int line_value = 1;
   int flash = 10;
   int ret;

   /* open /dev/gpiochip2 */
   output_chip = gpiod_chip_open_by_number(2);
   if (!output_chip)
          return -1;

   /* get line 3 (port19) of the gpiochip2 device */
   output_line = gpiod_chip_get_line(output_chip, 3);
   if(!output_line) {
          gpiod_chip_close(output_chip);
          return -1;
   }
```

```
    /* config port19 (GPO) as output and set ouput to high level */
    if (gpiod_line_request_output(output_line, "Port19_GPO",
                            GPIOD_LINE_ACTIVE_STATE_HIGH) == -1) {
            gpiod_line_release(output_line);
            gpiod_chip_close(output_chip);
            return -1;
    }

    /* toggle 10 times the port19 (GPO) of the max11300 device */
    for (int i=0; i < flash; i++) {
            line_value = !line_value;
            ret = gpiod_line_set_value(output_line, line_value);
            if (ret == -1) {
                    ret = -errno;
                    gpiod_line_release(output_line);
                    gpiod_chip_close(output_chip);
                    return ret;
            }
            sleep(1);
    }

    gpiod_line_release(output_line);
    gpiod_chip_close(output_chip);

    return 0;
}
```

Note: The source code of the applications developed during this lab is included in the linux_5.4_rpi4_drivers.zip file inside the linux_5.4_max11300_driver folder under application_code folder, and can be downloaded from the GitHub repository at https://github.com/ALIBERA/linux_book_2nd_edition

LAB 7.4: "GPIO expander device" module

This new LAB 7.4 has been added to the labs of Chapter 7 to reinforce the concepts of creating **NESTED THREADED GPIO irqchips** drivers, which were explained during the chapter seven of this book, and apply in a practical way how to create a gpio controller with interrupt capabilities. You will also develop a user application that request GPIO interrupts from user space using the GPIOlib APIs.

A new low cost evaluation board based on the CY8C9520A device will be used, thus expanding the number of evaluation boards that can be acquired to practice with the theory explained in Chapter 7.

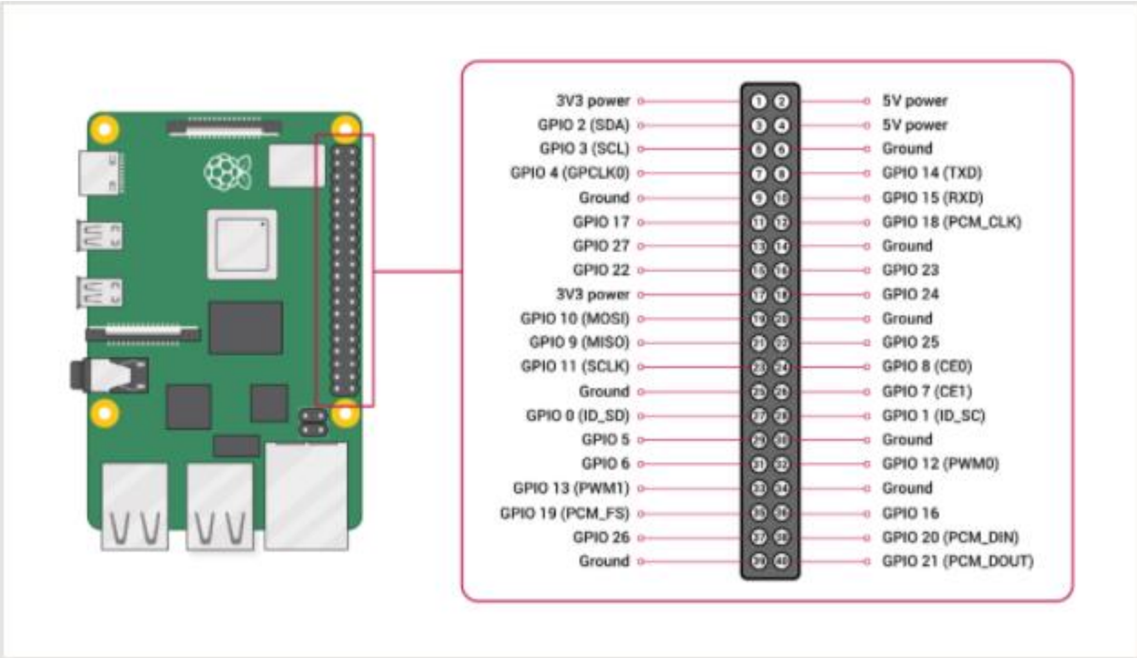
This new kernel module will control the Cypress CY8C9520A device. The CY8C9520A is a multi-port IO expander with on board user available EEPROM and several PWM outputs. The IO expander's data pins can be independently assigned as inputs, outputs, quasi-bidirectional input/outputs or PWM outputs. The individual data pins can be configured as open drain or collector, strong drive (10 mA source, 25 mA sink), resistively pulled up or down, or high impedance. The factory default configuration is pulled up internally. You can check all the info related to this device at https://www.cypress.com/products/cy8c95xx

The hardware platforms used in this lab are the Raspberry Pi 4 Model B board and the EXPAND 6 Click from MIKROE. The documentation of these boards can be found at https://www.raspberrypi.org/products/raspberry-pi-4-model-b/?resellerType=home and https://www.mikroe.com/expand-6-click

Not all the CY8C9520A features are included in this driver. The driver will configure the CY8C9520A port pins as input and outputs and will handle GPIO interrupts.

LAB 7.4 hardware description

In this lab, you will use the I2C pins of the Raspberry Pi 4 Model B 40-pin GPIO header, which is found on all current Raspberry Pi boards, to connect to the EXPAND 6 Click mikroBUS™ socket. See below the Raspberry Pi 4 Model B connector:



And the EXPAND 6 Click mikroBUS™ socket:

Notes	Pin	mikroBUS				Pin	Notes
	NC	1	AN	PWM	16	NC	
Reset	**RST**	2	RST	INT	15	**INT**	Interrupt
	NC	3	CS	RX	14	NC	
	NC	4	SCK	TX	13	NC	
	NC	5	MISO	SCL	12	**SCL**	I2C Clock
	NC	6	MOSI	SDA	11	**SDA**	I2C Data
Power Supply	**3.3V**	7	3.3V	5V	10	**5V**	Power Supply
Ground	**GND**	8	GND	GND	9	**GND**	Ground

Connect the Raspberry Pi´s I2C pins to the I2C ones of the CY8C9520A device, obtained from the EXPAND 6 Click mikroBUS™ socket:

- Connect Raspberry Pi 4 Model B **SCL** to CY8C9520A **SCL** (Pin 12 of Mikrobus)
- Connect Raspberry Pi 4 Model B **SDA** to CY8C9520A **SDA** (Pin 11 of Mikrobus)
- Connect Raspberry Pi 4 Model B **GPIO 23** to CY8C9520A **INT** (Pin 15 of Mikrobus)

Also connect the next power pins between the two boards:

- Connect Raspberry Pi 4 Model B **3.3V** to CY8C9520A **3.3V** (Pin 7 of Mikrobus)
- Connect Raspberry Pi 4 Model B **GND** to CY8C9520A **GND** (Pin 8 of Mikrobus)

The hardware setup between the two boards is already done!!

LAB 7.4 device tree description

Open the bcm2711-rpi-4-b.dts DT file and find the i2c1 controller master node. Inside the i2c1 node, you can see the pinctrl-0 property which configures the pins in I2C mode. The i2c1_pins are already defined in the bcm2711-rpi-4-b.dts file inside the gpio node property.

The i2c1 controller is enabled by writing "okay" to the status property. You will set to 100KHz the clock-frequency property. EXPAND 6 Click communicates with MPU using an I2C bus interface with a maximum frequency of 100kHz.

Now, you will add to the i2c1 controller node the cy8c9520a node. There must be a DT device node´s compatible property identical to the compatible string stored in one of the driver´s of_device_id structures. The reg property includes the I2C address of the device.

The interrupt-controller property is an empty property, which declares a node as a device that receives interrupt signals. The interrupt-cells property is a property of the interrupt controller, and defines how many cells are needed to specify a single interrupt in an interrupt client node. In our device node the interrupt-cells property is set to two, the first cell defines the index of the interrupt within the controller, while the second cell is used to specify the trigger and level flags of the interrupt.

Every GPIO controller node must contain both an empty gpio-controller property, and a gpio-cells integer property, which indicates the number of cells in a gpio-specifier for a gpio client device.

The interrupt-parent is a property containing a phandle to the interrupt controller that it is attached to. Nodes that do not have an interrupt-parent property can also inherit the property of their parent node. The CY8C9520A Interrupt pin (INT) is connected to the GPIO 23 pin of the

Raspberry Pi 4 Model B processor, so the interrupt parent of our device is the gpio peripheral of the Raspberry Pi 4 Model B processor.

The interrupts property is a property containing a list of interrupt specifiers, one for each interrupt output signal on the device. In our driver there is one output interrupt, so only one interrupt specifier containing the interrupted line number of the GPIO peripheral is needed.

See below in bold the device-tree configuration of our cy8c9520a device:

```
&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>;
   };

   ioexp@38 {
           compatible = "arrow,ioexp";
           reg = <0x38>;
   };

   ioexp@39 {
           compatible = "arrow,ioexp";
           reg = <0x39>;
   };

   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";
        };
};

adxl345@1d {
        compatible = "arrow,adxl345";
        reg = <0x1d>;
};

cy8c9520a: cy8c9520a@20 {
        compatible = "cy8c9520a";
        reg = <0x20>;
        interrupt-controller;
        #interrupt-cells = <2>;
        gpio-controller;
        #gpio-cells = <2>;

        interrupts = <23 1>;
        interrupt-parent = <&gpio>;
};

};
```

LAB 7.4 GPIO controller driver description

The main code sections of the driver will be described using two different categories: I2C driver setup, and GPIO driver interface. The CY8C9520A driver is based on the CY8C9540A driver from Intel Corporation.

I2C driver setup

These are the main code sections:

1. Include the required header files:

   ```
   #include <linux/i2c.h>
   ```

2. Create a i2c_driver structure:

   ```
   static struct i2c_driver cy8c9520a_driver = {
   ```

```
		.driver = {
			.name = DRV_NAME,
			.of_match_table = my_of_ids,
			.owner = THIS_MODULE,
			},
		.probe = cy8c9520a_probe,
		.remove = cy8c9520a_remove,
		.id_table = cy8c9520a_id,
};
```

3. Register to the I2C bus as a driver:

```
module_i2c_driver(cy8c9520a_driver);
```

4. Add "cy8c9520a" to the list of devices supported by the driver. The compatible variable matches with the compatible property of the cy8c9520a DT node:

```
static const struct of_device_id my_of_ids[] = {
	{ .compatible = "cy8c9520a"},
	{},
};
MODULE_DEVICE_TABLE(of, my_of_ids);
```

5. Define an array of i2c_device_id structures:

```
static const struct i2c_device_id cy8c9520a_id[] = {
	{DRV_NAME, 0},
	{}
};
MODULE_DEVICE_TABLE(i2c, cy8c9520a_id);
```

GPIO driver interface

The CY8C9520A driver will control the I/O expander's data pins as inputs and outputs. In this driver each and every GPIO pin can be used as an external interrupt. Whenever there is an input change on a specific GPIO pin, the IRQ interrupt will be asserted by the CY8C9520A GPIO controller.

The CY8C9520A driver will register its gpio_chip structure with the kernel, and its irq_chip structure with the IRQ system.

Our GPIO irqchip will fall in the category of NESTED THREADED GPIO IRQCHIPS, which are off-chip GPIO expanders that reside on the other side of a sleeping bus, such as I2C or SPI.

The GPIOlib framework will provide the kernel and user space APIs to control the GPIOs and handle their interrupts.

These are the main steps to create our CY8C9520A driver, which includes a GPIO controller with interrupt capabilities:

1. Include the following header, which defines the structures used to define a GPIO driver:

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

2. Initialize the gpio_chip structure with the different callbacks that will control the gpio lines of the GPIO controller, and register the gpiochip with the kernel using the devm_gpiochip_add_data() function. In the Listing 7-4 you can check the source code of these callback functions. Comments have been added before the main lines of the code to understand the meaning of the same.

```
static int cy8c9520a_gpio_init(struct cy8c9520a *cygpio)
{
	struct gpio_chip *gpiochip = &cygpio->gpio_chip;
	int err;

	gpiochip->label = cygpio->client->name;
	gpiochip->base = -1;
	gpiochip->ngpio = NGPIO;
	gpiochip->parent = &cygpio->client->dev;
	gpiochip->of_node = gpiochip->parent->of_node;
	gpiochip->can_sleep = true;
	gpiochip->direction_input = cy8c9520a_gpio_direction_input;
	gpiochip->direction_output = cy8c9520a_gpio_direction_output;
	gpiochip->get = cy8c9520a_gpio_get;
	gpiochip->set = cy8c9520a_gpio_set;
	gpiochip->owner = THIS_MODULE;

	/* register a gpio_chip */
	err = devm_gpiochip_add_data(gpiochip->parent, gpiochip, cygpio);
	if (err)
		return err;
	return 0;
}
```

3. Initialize the irq_chip structure with the different callbacks that will handle the GPIO interrupts flow. In the Listing 7-4, you can check the source code of these callback functions. Comments have been added before the main lines of the code to understand the meaning of the same.

```
static struct irq_chip cy8c9520a_irq_chip = {
	.name                 = "cy8c9520a-irq",
	.irq_mask             = cy8c9520a_irq_mask,
	.irq_unmask           = cy8c9520a_irq_unmask,
	.irq_bus_lock         = cy8c9520a_irq_bus_lock,
	.irq_bus_sync_unlock  = cy8c9520a_irq_bus_sync_unlock,
	.irq_set_type         = cy8c9520a_irq_set_type,
};
```

4. Write the interrupt setup function for the CY8C9520A device. The gpiochip_irqchip_add_nested() function adds a nested irqchip to a gpiochip. The gpiochip_irqchip_add_nested() function takes as a parameter the handle_simple_irq flow handler, which handles simple interrupts sent from a demultiplexing interrupt handler or coming from hardware where no interrupt hardware control is necessary. You can find all the complete information about irq-flow methods at https://www.kernel.org/doc/html/latest/core-api/genericirq.html

   The interrupt handler for the GPIO child device will be called inside of a new thread created by the handle_nested_irq() function, which is called inside the interrupt handler of the driver.

   The devm_request_threaded_irq() function inside cy8c9520a_irq_setup() will allocate the interrupt line taking as parameters the driver´s interrupt handler, the linux IRQ number (client->irq), flags that will instruct the kernel about the desired behaviour (IRQF_ONESHOT | IRQF_TRIGGER_HIGH), and a pointer to the cygpio global structure that will be recovered in the interrupt handler of the driver.

```
static int cy8c9520a_irq_setup(struct cy8c9520a *cygpio)
{
	struct i2c_client *client = cygpio->client;
	struct gpio_chip *chip = &cygpio->gpio_chip;
	u8 dummy[NPORTS];
	int ret, i;

	mutex_init(&cygpio->irq_lock);

	/*
	 * Clear interrupt state registers by reading the three registers
	 * Interrupt Status Port0, Interrupt Status Port1,
	 * Interrupt Status Port2,
	 * and store the values in a dummy array
	 */
	i2c_smbus_read_i2c_block_data(client, REG_INTR_STAT_PORT0,
				      NPORTS, dummy);

	/*
	 * Initialise Interrupt Mask Port Register (19h) for each port
	 * Disable the activation of the INT lines. Each 1 in this
	 * register masks (disables) the int from the corresponding GPIO
	 */
	memset(cygpio->irq_mask_cache, 0xff, sizeof(cygpio->irq_mask_cache));
	memset(cygpio->irq_mask, 0xff, sizeof(cygpio->irq_mask));

	/* Disable interrupts in all the gpio lines */
	for (i = 0; i < NPORTS; i++) {
```

```
		i2c_smbus_write_byte_data(client, REG_PORT_SELECT, i);

		i2c_smbus_write_byte_data(client, REG_INTR_MASK,
					  cygpio->irq_mask[i]);
	}

	/* add a nested irqchip to the gpiochip */
	gpiochip_irqchip_add_nested(chip,
				    &cy8c9520a_irq_chip,
				    0,
				    handle_simple_irq,
				    IRQ_TYPE_NONE);

	/*
	 * Request interrupt on a GPIO pin of the external processor
	 * this processor pin is connected to the INT pin of the cy8c9520a
	 */
	devm_request_threaded_irq(&client->dev, client->irq, NULL,
				  cy8c9520a_irq_handler,
				  IRQF_ONESHOT | IRQF_TRIGGER_HIGH,
				  dev_name(&client->dev), cygpio);

	/*
	 * set up a nested irq handler for a gpio_chip from a parent IRQ
	 * you can now request interrupts from GPIO child drivers nested
	 * to the cy8c9520a driver
	 */
	gpiochip_set_nested_irqchip(chip,
				    &cy8c9520a_irq_chip,
				    cygpio->irq);

	return 0;
err:
	mutex_destroy(&cygpio->irq_lock);
	return ret;
}
```

5. Write the interrupt handler for the CY8C9520A device. Inside this handler the pending GPIO interrupts are checked by reading the pending variable value, then the position of the first bit set in the variable is returned; the _ffs() function is used to perform this task. For each pending interrupt that is found, there is a call to the handle_nested_irq() wrapper function, which in turn calls the interrupt handler of the GPIO child driver that has requested a CY8C9520A GPIO interrupt by using the devm_request_threaded_irq() function. The parameter of the handle_nested_irq() function is the Linux IRQ number previously returned by using the irq_find_mapping() function, which receives the hwirq of the input pin as a parameter (gpio_irq variable). The pending interrupt is cleared by

doing pending &= ~BIT(gpio), and the same process is repeated until all the pending interrupts are being managed.

```
static irqreturn_t cy8c9520a_irq_handler(int irq, void *devid)
{
	struct cy8c9520a *cygpio = devid;
	u8 stat[NPORTS], pending;
	unsigned port, gpio, gpio_irq;
	int ret;

	/*
	 * store in stat and clear (to enable ints)
	 * the three interrupt status registers by reading them
	 */
	i2c_smbus_read_i2c_block_data(cygpio->client,
				      REG_INTR_STAT_PORT0,
				      NPORTS, stat);

	ret = IRQ_NONE;

	for (port = 0; port < NPORTS; port ++) {
		mutex_lock(&cygpio->irq_lock);

		/*
		 * In every port check the GPIOs that have their int unmasked
		 * and whose bits have been enabled in their REG_INTR_STAT_PORT
		 * register due to an interrupt in the GPIO, and store the new
		 * value in the pending register
		 */
		pending = stat[port] & (~cygpio->irq_mask[port]);
		mutex_unlock(&cygpio->irq_lock);

		while (pending) {
			ret = IRQ_HANDLED;
			/* get the first gpio that has got an int */
			gpio = __ffs(pending);

			/* clears the gpio in the pending register */
			pending &= ~BIT(gpio);

			/* gets the int number associated to this gpio */
			gpio_irq = cy8c9520a_port_offs[port] + gpio;

			/* launch the ISR of the GPIO child driver */
			handle_nested_irq(irq_find_mapping(cygpio-
>gpio_chip.irq.domain, gpio_irq));

		}
```

```
		}

		return ret;
	}
```

See in the next **Listing 7-4** the complete "GPIO expander device" driver source code for the Raspberry Pi 4 Model B processor.

Note: The "GPIO expander device" driver source code developed for the Raspberry Pi 4 Model B board is included in the linux_5.4_rpi4_drivers.zip file, in the linux_5.4_CY8C9520A_driver folder, and can be downloaded from the GitHub repository at https://github.com/ALIBERA/linux_book_2nd_edition

Listing 7-4: CY8C9520A_rpi4.c

```
#include <linux/i2c.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/gpio/driver.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>

#define DRV_NAME                        "cy8c9520a"

/* cy8c9520a settings */
#define NGPIO                           20
#define DEVID_CY8C9520A                 0x20
#define NPORTS                          3

/* Register offset  */
#define REG_INPUT_PORT0                 0x00
#define REG_OUTPUT_PORT0                0x08
#define REG_INTR_STAT_PORT0             0x10
#define REG_PORT_SELECT                 0x18
#define REG_SELECT_PWM                  0x1a
#define REG_INTR_MASK                   0x19
#define REG_PIN_DIR                     0x1c
#define REG_DRIVE_PULLUP                0x1d
#define REG_DRIVE_PULLDOWN              0x1e
#define REG_DEVID_STAT                  0x2e

/* definition of the global structure for the driver */
struct cy8c9520a {
   struct i2c_client *client;
   struct gpio_chip gpio_chip;
   struct gpio_desc *gpio;
```

```
	int irq;
	struct mutex lock;
	/* protect serialized access to the interrupt controller bus */
	struct mutex irq_lock;
	/* cached output registers */
	u8 outreg_cache[NPORTS];
	/* cached IRQ mask */
	u8 irq_mask_cache[NPORTS];
	/* IRQ mask to be applied */
	u8 irq_mask[NPORTS];
};

/* Per-port GPIO offset */
static const u8 cy8c9520a_port_offs[] = {
	0,
	8,
	16,
};

/* return the port of the gpio */
static inline u8 cypress_get_port(unsigned int gpio)
{
	u8 i = 0;
	for (i = 0; i < sizeof(cy8c9520a_port_offs) - 1; i ++) {
		if (! (gpio / cy8c9520a_port_offs[i + 1]))
			break;
	}
	return i;
}

/* get the gpio offset inside its respective port */
static inline u8 cypress_get_offs(unsigned gpio, u8 port)
{
	return gpio - cy8c9520a_port_offs[port];
}

/*
 * struct gpio_chip get callback function.
 * It gets the input value of the GPIO line (0=low, 1=high)
 * accessing to the REG_INPUT_PORT register
 */
static int cy8c9520a_gpio_get(struct gpio_chip *chip,
			      unsigned int gpio)
{
	int ret;
	u8 port, in_reg;

	struct cy8c9520a *cygpio = gpiochip_get_data(chip);
```

```
    dev_info(chip->parent, "cy8c9520a_gpio_get function is called\n");

    /* get the input port address address (in_reg) for the GPIO */
    port = cypress_get_port(gpio);
    in_reg = REG_INPUT_PORT0 + port;

    dev_info(chip->parent, "the in_reg address is %u\n", in_reg);

    mutex_lock(&cygpio->lock);

    ret = i2c_smbus_read_byte_data(cygpio->client, in_reg);
    if (ret < 0) {
            dev_err(chip->parent, "can't read input port %u\n", in_reg);
    }

    dev_info(chip->parent,
            "cy8c9520a_gpio_get function with %d value is returned\n",
            ret);

    mutex_unlock(&cygpio->lock);

    /*
     * check the status of the GPIO in its input port register
     * and return it. If expression is not 0 returns 1
     */
    return !!(ret & BIT(cypress_get_offs(gpio, port)));
}

/*
 * struct gpio_chip set callback function.
 * It sets the output value of the GPIO line in
 * GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the REG_OUTPUT_PORT register
 */
static void cy8c9520a_gpio_set(struct gpio_chip *chip,
                              unsigned int gpio, int val)
{
    int ret;
    u8 port, out_reg;
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    dev_info(chip->parent,
             "cy8c9520a_gpio_set_value func with %d value is called\n",
             val);

    /* get the output port address address (out_reg) for the GPIO */
    port = cypress_get_port(gpio);
```

```
	out_reg = REG_OUTPUT_PORT0 + port;

	mutex_lock(&cygpio->lock);

	/*
	 * if val is 1, gpio output level is high
	 * if val is 0, gpio output level is low
	 * the output registers were previously cached in cy8c9520a_setup()
	 */
	if (val) {
		cygpio->outreg_cache[port] |= BIT(cypress_get_offs(gpio, port));
	} else {
		cygpio->outreg_cache[port] &= ~BIT(cypress_get_offs(gpio, port));
	}

	ret = i2c_smbus_write_byte_data(cygpio->client, out_reg,
					cygpio->outreg_cache[port]);
	if (ret < 0) {
		dev_err(chip->parent, "can't write output port %u\n", port);
	}

	mutex_unlock(&cygpio->lock);
}

/*
 * struct gpio_chip direction_output callback function.
 * It configures the GPIO as an output writing to
 * the REG_PIN_DIR register of the selected port
 */
static int cy8c9520a_gpio_direction_output(struct gpio_chip *chip,
					   unsigned int gpio, int val)
{
	int ret;
	u8 pins, port;

	struct cy8c9520a *cygpio = gpiochip_get_data(chip);

	/* gets the port number of the gpio */
	port = cypress_get_port(gpio);

	dev_info(chip->parent, "cy8c9520a_gpio_direction output is called\n");

	mutex_lock(&cygpio->lock);

	/* select the port where we want to config the GPIO as output */
	ret = i2c_smbus_write_byte_data(cygpio->client, REG_PORT_SELECT, port);
	if (ret < 0) {
		dev_err(chip->parent, "can't select port %u\n", port);
```

```
		goto err;
	}

	ret = i2c_smbus_read_byte_data(cygpio->client, REG_PIN_DIR);
	if (ret < 0) {
		dev_err(chip->parent, "can't read pin direction\n");
		goto err;
	}

	/* simply transform int to u8 */
	pins = (u8)ret & 0xff;

	/* add the direction of the new pin. Set 1 if input and set 0 is output */
	pins &= ~BIT(cypress_get_offs(gpio, port));

	ret = i2c_smbus_write_byte_data(cygpio->client, REG_PIN_DIR, pins);
	if (ret < 0) {
		dev_err(chip->parent, "can't write pin direction\n");
	}

err:
	mutex_unlock(&cygpio->lock);
	cy8c9520a_gpio_set(chip, gpio, val);
	return ret;
}

/*
 * struct gpio_chip direction_input callback function.
 * It configures the GPIO as an input writing to
 * the REG_PIN_DIR register of the selected port
 */
static int cy8c9520a_gpio_direction_input(struct gpio_chip *chip,
					   unsigned int gpio)
{
	int ret;
	u8 pins, port;

	struct cy8c9520a *cygpio = gpiochip_get_data(chip);

	/* gets the port number of the gpio */
	port = cypress_get_port(gpio);

	dev_info(chip->parent, "cy8c9520a_gpio_direction input is called\n");

	mutex_lock(&cygpio->lock);

	/* select the port where we want to config the GPIO as input */
	ret = i2c_smbus_write_byte_data(cygpio->client, REG_PORT_SELECT, port);
```

```
    if (ret < 0) {
            dev_err(chip->parent, "can't select port %u\n", port);
            goto err;
    }

    ret = i2c_smbus_read_byte_data(cygpio->client, REG_PIN_DIR);
    if (ret < 0) {
            dev_err(chip->parent, "can't read pin direction\n");
            goto err;
    }

    /* simply transform int to u8 */
    pins = (u8)ret & 0xff;

    /*
     * add the direction of the new pin.
     * Set 1 if input (out == 0) and set 0 is ouput (out == 1)
     */
    pins |= BIT(cypress_get_offs(gpio, port));

    ret = i2c_smbus_write_byte_data(cygpio->client, REG_PIN_DIR, pins);
    if (ret < 0) {
            dev_err(chip->parent, "can't write pin direction\n");
            goto err;
    }

err:
    mutex_unlock(&cygpio->lock);
    return ret;
}

/* function to lock access to slow bus (i2c) chips */
static void cy8c9520a_irq_bus_lock(struct irq_data *d)
{
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    dev_info(chip->parent, "cy8c9520a_irq_bus_lock is called\n");
    mutex_lock(&cygpio->irq_lock);
}

/*
 * function to sync and unlock slow bus (i2c) chips
 * REG_INTR_MASK register is accessed via I2C
 * write 0 to the interrupt mask register line to
 * activate the interrupt on the GPIO
 */
static void cy8c9520a_irq_bus_sync_unlock(struct irq_data *d)
{
```

```
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    int ret, i;
    unsigned int gpio;
    u8 port;
    dev_info(chip->parent, "cy8c9520a_irq_bus_sync_unlock is called\n");

    gpio = d->hwirq;
    port = cypress_get_port(gpio);

    /* irq_mask_cache stores the last value of irq_mask for each port */
    for (i = 0; i < NPORTS; i++) {
            /*
             * check if some of the bits have changed from the last cached value
             * irq_mask registers were initialized in cy8c9520a_irq_setup()
             */
            if (cygpio->irq_mask_cache[i] ^ cygpio->irq_mask[i]) {
                    dev_info(chip->parent, "gpio %u is unmasked\n", gpio);
                    cygpio->irq_mask_cache[i] = cygpio->irq_mask[i];
                    ret = i2c_smbus_write_byte_data(cygpio->client,
                                                    REG_PORT_SELECT, i);
                    if (ret < 0) {
                            dev_err(chip->parent, "can't select port %u\n", port);
                            goto err;
                    }

                    /* enable the interrupt for the GPIO unmasked */
                    ret = i2c_smbus_write_byte_data(cygpio->client, REG_INTR_MASK,
                                                    cygpio->irq_mask[i]);
                    if (ret < 0) {
                            dev_err(chip->parent,
                                    "can't write int mask on port %u\n", port);
                            goto err;
                    }

                    ret = i2c_smbus_read_byte_data(cygpio->client, REG_INTR_MASK);
                    dev_info(chip->parent, "the REG_INTR_MASK value is %d\n", ret);

            }
    }

err:
    mutex_unlock(&cygpio->irq_lock);
}

/*
 * mask (disable) the GPIO interrupt.
 * In the initial setup all the int lines are masked
```

```
 */
static void cy8c9520a_irq_mask(struct irq_data *d)
{
   u8 port;
   struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
   struct cy8c9520a *cygpio = gpiochip_get_data(chip);
   unsigned gpio = d->hwirq;
   port = cypress_get_port(gpio);
   dev_info(chip->parent, "cy8c9520a_irq_mask is called\n");

   cygpio->irq_mask[port] |= BIT(cypress_get_offs(gpio, port));
}

/*
 * unmask (enable) the GPIO interrupt.
 * In the initial setup all the int lines are masked
 */
static void cy8c9520a_irq_unmask(struct irq_data *d)
{
   u8 port;
   struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
   struct cy8c9520a *cygpio = gpiochip_get_data(chip);
   unsigned gpio = d->hwirq;
   port = cypress_get_port(gpio);
   dev_info(chip->parent, "cy8c9520a_irq_unmask is called\n");

   cygpio->irq_mask[port] &= ~BIT(cypress_get_offs(gpio, port));
}

/* set the flow type (IRQ_TYPE_LEVEL/etc.) of the IRQ */
static int cy8c9520a_irq_set_type(struct irq_data *d, unsigned int type)
{
   int ret = 0;
   struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
   struct cy8c9520a *cygpio = gpiochip_get_data(chip);

   dev_info(chip->parent, "cy8c9520a_irq_set_type is called\n");

   if ((type != IRQ_TYPE_EDGE_BOTH) && (type != IRQ_TYPE_EDGE_FALLING)) {
           dev_err(&cygpio->client->dev, "irq %d: unsupported type %d\n",
                   d->irq, type);
           ret = -EINVAL;
           goto err;
   }

err:
   return ret;
}
```

```
/* Iinitialization of the irq_chip structure with callback functions */
static struct irq_chip cy8c9520a_irq_chip = {
    .name                 = "cy8c9520a-irq",
    .irq_mask             = cy8c9520a_irq_mask,
    .irq_unmask           = cy8c9520a_irq_unmask,
    .irq_bus_lock         = cy8c9520a_irq_bus_lock,
    .irq_bus_sync_unlock  = cy8c9520a_irq_bus_sync_unlock,
    .irq_set_type         = cy8c9520a_irq_set_type,
};

/*
 * interrupt handler for the cy8c9520a. It is called when
 * there is a rising or falling edge in the unmasked GPIO
 */
static irqreturn_t cy8c9520a_irq_handler(int irq, void *devid)
{
    struct cy8c9520a *cygpio = devid;
    u8 stat[NPORTS], pending;
    unsigned port, gpio, gpio_irq;
    int ret;

    pr_info ("the interrupt ISR has been entered\n");

    /*
     * store in stat and clear (to enable ints)
     * the three interrupt status registers by reading them
     */
    ret = i2c_smbus_read_i2c_block_data(cygpio->client,
                                        REG_INTR_STAT_PORT0,
                                        NPORTS, stat);
    if (ret < 0) {
            memset(stat, 0, sizeof(stat));
    }

    ret = IRQ_NONE;

    for (port = 0; port < NPORTS; port ++) {
            mutex_lock(&cygpio->irq_lock);

            /*
             * In every port check the GPIOs that have their int unmasked
             * and whose bits have been enabled in their REG_INTR_STAT_PORT
             * register due to an interrupt in the GPIO, and store the new
             * value in the pending register
             */
            pending = stat[port] & (~cygpio->irq_mask[port]);
            mutex_unlock(&cygpio->irq_lock);
```

```
		/* Launch the ISRs of all the gpios that requested an interrupt */
		while (pending) {
			ret = IRQ_HANDLED;
			/* get the first gpio that has got an int */
			gpio = __ffs(pending);

			/* clears the gpio in the pending register */
			pending &= ~BIT(gpio);

			/* gets the int number associated to this gpio */
			gpio_irq = cy8c9520a_port_offs[port] + gpio;

			/* launch the ISR of the GPIO child driver */
			handle_nested_irq(irq_find_mapping(cygpio->gpio_chip.irq.domain,
							gpio_irq));

		}
	}

	return ret;
}

/* Initial setup for the cy8c9520a */
static int cy8c9520a_setup(struct cy8c9520a *cygpio)
{
	int ret, i;
	struct i2c_client *client = cygpio->client;

	/* Disable PWM, set all GPIOs as input.  */
	for (i = 0; i < NPORTS; i ++) {
		ret = i2c_smbus_write_byte_data(client, REG_PORT_SELECT, i);
		if (ret < 0) {
			dev_err(&client->dev, "can't select port %u\n", i);
			goto end;
		}

		ret = i2c_smbus_write_byte_data(client, REG_SELECT_PWM, 0x00);
		if (ret < 0) {
			dev_err(&client->dev, "can't write to SELECT_PWM\n");
			goto end;
		}

		ret = i2c_smbus_write_byte_data(client, REG_PIN_DIR, 0xff);
		if (ret < 0) {
			dev_err(&client->dev, "can't write to PIN_DIR\n");
			goto end;
		}
```

```
	}

	/* Cache the output registers (Output Port 0, Output Port 1, Output Port 2) */
	ret = i2c_smbus_read_i2c_block_data(client, REG_OUTPUT_PORT0,
					    sizeof(cygpio->outreg_cache),
					    cygpio->outreg_cache);
	if (ret < 0) {
		dev_err(&client->dev, "can't cache output registers\n");
		goto end;
	}

	dev_info(&client->dev, "the cy8c9520a_setup is done\n");

end:
	return ret;
}

/* Interrupt setup for the cy8c9520a */
static int cy8c9520a_irq_setup(struct cy8c9520a *cygpio)
{
	struct i2c_client *client = cygpio->client;
	struct gpio_chip *chip = &cygpio->gpio_chip;
	u8 dummy[NPORTS];
	int ret, i;

	mutex_init(&cygpio->irq_lock);

	dev_info(&client->dev, "the cy8c9520a_irq_setup function is entered\n");

	/*
	 * Clear interrupt state registers by reading the three registers
	 * Interrupt Status Port0, Interrupt Status Port1, Interrupt Status Port2,
	 * and store the values in a dummy array
	 */
	ret = i2c_smbus_read_i2c_block_data(client, REG_INTR_STAT_PORT0,
					    NPORTS, dummy);
	if (ret < 0) {
		dev_err(&client->dev, "couldn't clear int status\n");
		goto err;
	}

	dev_info(&client->dev, "the interrupt state registers are cleared\n");

	/*
	 * Initialise Interrupt Mask Port Register (19h) for each port
	 * Disable the activation of the INT lines. Each 1 in this
	 * register masks (disables) the int from the corresponding GPIO
	 */
```

```
memset(cygpio->irq_mask_cache, 0xff, sizeof(cygpio->irq_mask_cache));
memset(cygpio->irq_mask, 0xff, sizeof(cygpio->irq_mask));

/* Disable interrupts in all the gpio lines */
for (i = 0; i < NPORTS; i++) {
        ret = i2c_smbus_write_byte_data(client, REG_PORT_SELECT, i);
        if (ret < 0) {
                dev_err(&client->dev, "can't select port %u\n", i);
                goto err;
        }

        ret = i2c_smbus_write_byte_data(client, REG_INTR_MASK,
                                        cygpio->irq_mask[i]);
        if (ret < 0) {
                dev_err(&client->dev,
                        "can't write int mask on port %u\n", i);
                goto err;
        }
}

dev_info(&client->dev, "the interrupt mask port registers are set\n");

/* add a nested irqchip to the gpiochip */
ret =  gpiochip_irqchip_add_nested(chip,
                                  &cy8c9520a_irq_chip,
                                  0,
                                  handle_simple_irq,
                                  IRQ_TYPE_NONE);
if (ret) {
        dev_err(&client->dev,
                "could not connect irqchip to gpiochip\n");
        return ret;
}

/*
 * Request interrupt on a GPIO pin of the external processor
 * this processor pin is connected to the INT pin of the cy8c9520a
 */
ret = devm_request_threaded_irq(&client->dev, client->irq, NULL,
                               cy8c9520a_irq_handler,
                               IRQF_ONESHOT | IRQF_TRIGGER_HIGH,
                               dev_name(&client->dev), cygpio);
if (ret) {
        dev_err(&client->dev, "failed to request irq %d\n", cygpio->irq);
                return ret;
}

/*
```

```
	 * set up a nested irq handler for a gpio_chip from a parent IRQ
	 * you can now request interrupts from GPIO child drivers nested
	 * to the cy8c9520a driver
	 */
	gpiochip_set_nested_irqchip(chip,
				    &cy8c9520a_irq_chip,
				    cygpio->irq);

	dev_info(&client->dev, "the interrupt setup is done\n");

	return 0;
err:
	mutex_destroy(&cygpio->irq_lock);
	return ret;
}

/*
 * Initialize the cy8c9520a gpio controller (struct gpio_chip)
 * and register it to the kernel
 */
static int cy8c9520a_gpio_init(struct cy8c9520a *cygpio)
{
	struct gpio_chip *gpiochip = &cygpio->gpio_chip;
	int err;

	gpiochip->label = cygpio->client->name;
	gpiochip->base = -1;
	gpiochip->ngpio = NGPIO;
	gpiochip->parent = &cygpio->client->dev;
	gpiochip->of_node = gpiochip->parent->of_node;
	gpiochip->can_sleep = true;
	gpiochip->direction_input = cy8c9520a_gpio_direction_input;
	gpiochip->direction_output = cy8c9520a_gpio_direction_output;
	gpiochip->get = cy8c9520a_gpio_get;
	gpiochip->set = cy8c9520a_gpio_set;
	gpiochip->owner = THIS_MODULE;

	/* register a gpio_chip */
	err = devm_gpiochip_add_data(gpiochip->parent, gpiochip, cygpio);
	if (err)
		return err;
	return 0;
}

static int cy8c9520a_probe(struct i2c_client *client,
			   const struct i2c_device_id *id)
{
	struct cy8c9520a *cygpio;
```

```
int ret;
unsigned int dev_id;

dev_info(&client->dev, "cy8c9520a_probe() function is called\n");

if (!i2c_check_functionality(client->adapter,
                             I2C_FUNC_SMBUS_I2C_BLOCK |
                             I2C_FUNC_SMBUS_BYTE_DATA)) {
       dev_err(&client->dev, "SMBUS Byte/Block unsupported\n");
       return -EIO;
}

/* allocate global private structure for a new device */
cygpio = devm_kzalloc(&client->dev, sizeof(*cygpio), GFP_KERNEL);
if (!cygpio) {
       dev_err(&client->dev, "failed to alloc memory\n");
       return -ENOMEM;
}

cygpio->client = client;

mutex_init(&cygpio->lock);

/* Whoami */
dev_id = i2c_smbus_read_byte_data(client, REG_DEVID_STAT);
if (dev_id < 0) {
       dev_err(&client->dev, "can't read device ID\n");
       ret = dev_id;
       goto err;
}
dev_info(&client->dev, "dev_id=0x%x\n", dev_id & 0xff);

/* Initial setup for the cy8c9520a */
ret = cy8c9520a_setup(cygpio);
if (ret < 0) {
       goto err;
}

/* Initialize the cy8c9520a gpio controller */
ret = cy8c9520a_gpio_init(cygpio);
if (ret) {
       goto err;
}

/* Interrupt setup for the cy8c9520a */
ret = cy8c9520a_irq_setup(cygpio);
if (ret) {
       goto err;
```

```
    }

    /* link the I2C device with the cygpio device */
    i2c_set_clientdata(client, cygpio);

    return 0;
err:
    mutex_destroy(&cygpio->lock);

    return ret;
}

static int cy8c9520a_remove(struct i2c_client *client)
{
    dev_info(&client->dev, "cy8c9520a_remove() function is called\n");

    return 0;
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "cy8c9520a"},
    {},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static const struct i2c_device_id cy8c9520a_id[] = {
    {DRV_NAME, 0},
    {}
};
MODULE_DEVICE_TABLE(i2c, cy8c9520a_id);

static struct i2c_driver cy8c9520a_driver = {
    .driver = {
              .name = DRV_NAME,
              .of_match_table = my_of_ids,
              .owner = THIS_MODULE,
              },
    .probe = cy8c9520a_probe,
    .remove = cy8c9520a_remove,
    .id_table = cy8c9520a_id,
};
module_i2c_driver(cy8c9520a_driver);

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a driver that controls the \
                   cy8c9520a I2C GPIO expander");
```

LAB 7.4 GPIO child driver description

In this lab, you will develop a GPIO child driver (int_rpi4_gpio) which will request a GPIO IRQ from the CY8C9520A gpio controller. You will use the LAB 7.1: "button interrupt device" Module of this book as a starting point for the development of the driver. Whenever there is a change in the first input line of the CY8C9520A P0 port, the IRQ interrupt will be asserted by the CY8C9520A GPIO controller, and its interrupt handler cy8c9520a_irq_handler() will be called. The CY8C9520A driver's interrupt handler will call handle_nested_irq(), which in turn calls the interrupt handler P0_line0_isr() of our GPIO child driver.

The GPIO child driver will request the GPIO INT by using the devm_request_threaded_irq() function. Before calling this function, the driver will return the Linux IRQ number from the device tree by using the platform_get_irq() function.

See below the device-tree configuration for the int_rpi4_gpio device that should be included in the the bcm2711-rpi-4-b.dts DT file. Check the differences with the int_key node of the LAB 7.1: "button interrupt device" Module that was taken as a reference for the development of this driver.

In our new driver, the interrupt-parent is the cy8c9520a node of our CY8C9520A gpio controller driver and the GPIO interrupt line included in the interrupts property has the number 0, which matches with the first input line of the CY8C9520A P0 controller.

```
int_key {
        compatible = "arrow,intkey";
        pinctrl-names = "default";
        pinctrl-0 = <&key_pin>;
        gpios = <&gpio 23 0>;
        interrupts = <23 1>;
        interrupt-parent = <&gpio>;
   };

   int_gpio {
        compatible = "arrow,int_gpio_expand";
        pinctrl-names = "default";
        interrupt-parent = <&cy8c9520a>;
        interrupts = <0  IRQ_TYPE_EDGE_BOTH>;
   };
```

See in the next **Listing 7-5** the complete "GPIO child device" driver source code for the Raspberry Pi 4 Model B processor.

Note: The "GPIO child device" driver source code developed for the Raspberry Pi 4 Model B board is included in the linux_5.4_rpi4_drivers.zip file, in the linux_5.4_CY8C9520A_driver folder under the linux_5.4_gpio_int_driver folder, and can be downloaded from the GitHub repository at https://github.com/ALIBERA/linux_book_2nd_edition

Listing 7-5: int_rpi4_gpio.c

```
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/interrupt.h>
#include <linux/gpio/consumer.h>
#include <linux/miscdevice.h>
#include <linux/of_device.h>

static char *INT_NAME = "P0_line0_INT";

/* interrupt handler */
static irqreturn_t P0_line0_isr(int irq, void *data)
{
   struct device *dev = data;
   dev_info(dev, "interrupt received. key: %s\n", INT_NAME);
   return IRQ_HANDLED;
}

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

static int my_probe(struct platform_device *pdev)
{
   int ret_val, irq;
   struct device *dev = &pdev->dev;

   dev_info(dev, "my_probe() function is called.\n");

   /* Get the Linux IRQ number */
   irq = platform_get_irq(pdev, 0);
   if (irq < 0){
          dev_err(dev, "irq is not available\n");
          return -EINVAL;
   }
   dev_info(dev, "IRQ_using_platform_get_irq: %d\n", irq);

   /* Allocate the interrupt line */
   ret_val = devm_request_threaded_irq(dev, irq, NULL, P0_line0_isr,
                            IRQF_ONESHOT | IRQF_TRIGGER_FALLING |
IRQF_TRIGGER_RISING,
                            INT_NAME, dev);
```

```
	if (ret_val) {
		dev_err(dev, "Failed to request interrupt %d, error %d\n", irq,
ret_val);
		return ret_val;
	}

	ret_val = misc_register(&helloworld_miscdevice);
	if (ret_val != 0)
	{
		dev_err(dev, "could not register the misc device mydev\n");
		return ret_val;
	}

	dev_info(dev, "mydev: got minor %i\n",helloworld_miscdevice.minor);
	dev_info(dev, "my_probe() function is exited.\n");

	return 0;
}

static int my_remove(struct platform_device *pdev)
{
	dev_info(&pdev->dev, "my_remove() function is called.\n");
	misc_deregister(&helloworld_miscdevice);
	dev_info(&pdev->dev, "my_remove() function is exited.\n");
	return 0;
}

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

MODULE_DEVICE_TABLE(of, my_of_ids);

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

module_platform_driver(my_platform_driver);

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

```
MODULE_DESCRIPTION("This is a GPIO INT platform driver");
```

LAB 7.4 GPIO based IRQ application

In the previous section, you have seen how to request and handle a GPIO IRQ by using a GPIO child driver. In the following **Listing 7-6**, you will see how to request and handle an interrupt from the user space for the first line of the CY8C9520A P0 port. You will use the GPIOlib user space APIs, that will handle the GPIO INT through ioctl calls on the /dev/gpiochip2 char device file.

Note: The "GPIO based IRQ application" source code developed for the Raspberry Pi 4 Model B board is included in the linux_5.4_rpi4_drivers.zip file, in the linux_5.4_CY8C9520A_driver folder under the app folder, and can be downloaded from the GitHub repository at https://github.com/ALIBERA/linux_book_2nd_edition

Listing 7-6: gpio_int.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <poll.h>
#include <string.h>
#include <linux/gpio.h>
#include <sys/ioctl.h>

#define DEV_GPIO  "/dev/gpiochip2"

#define POLL_TIMEOUT -1 /* No timeout */

int main(int argc, char *argv[])
{
    int fd, fd_in;
    int ret;
    int flags;

    struct gpioevent_request req;
    struct gpioevent_data evdata;
    struct pollfd fdset;

    /* open gpio */
    fd = open(DEV_GPIO, O_RDWR);
    if (fd < 0) {
        printf("ERROR: open %s ret=%d\n", DEV_GPIO, fd);
        return -1;
    }
```

```
    /* Request GPIO P0 first line interrupt */
    req.lineoffset = 0;
    req.handleflags = GPIOHANDLE_REQUEST_INPUT;
    req.eventflags  = GPIOEVENT_REQUEST_BOTH_EDGES;
    strncpy(req.consumer_label, "gpio_irq", sizeof(req.consumer_label) - 1);

    /* requrest line event handle */
    ret = ioctl(fd, GPIO_GET_LINEEVENT_IOCTL, &req);
    if (ret) {
        printf("ERROR: ioctl get line event ret=%d\n", ret);
        return -1;
    }

    /* set event fd nonbloack read */
    fd_in = req.fd;
    flags = fcntl(fd_in, F_GETFL);
    flags |= O_NONBLOCK;
    ret = fcntl(fd_in, F_SETFL, flags);
    if (ret) {
        printf("ERROR: fcntl set nonblock read\n");
    }

    for (;;) {
        fdset.fd      = fd_in;
        fdset.events  = POLLIN;
        fdset.revents = 0;

        /* poll gpio line event */
        ret = poll(&fdset, 1, POLL_TIMEOUT);
        if (ret <= 0)
            continue;

        if (fdset.revents & POLLIN) {
            printf("irq received.\n");
            /* read event data */
            ret = read(fd_in, &evdata, sizeof(evdata));
            if (ret == sizeof(evdata))
                printf("id: %d, timestamp: %lld\n", evdata.id, evdata.timestamp);
        }
    }

    /* close gpio */
    close(fd);

    return 0;
}
```

LAB 7.4 driver demonstration

Download the linux_5.4_rpi4_drivers.zip file from the github of the book and unzip it in the home folder of your Linux host:

```
~/linux_5.4_rpi4_drivers$ cd linux_5.4_CY8C9520A_driver
```

Compile and deploy the drivers and the application to the **Raspberry Pi 4 Model B** board:

```
~/linux_5.4_rpi4_drivers/linux_5.4_CY8C9520A_driver$ make

~/linux_5.4_rpi4_drivers/linux_5.4_CY8C9520A_driver$ make deploy

~/linux_5.4_rpi4_drivers/linux_5.4_CY8C9520A_driver/linux_5.4_gpio_int_driver$
make

~/linux_5.4_rpi4_drivers/linux_5.4_CY8C9520A_driver/linux_5.4_gpio_int_driver$
make deploy

~/linux_5.4_rpi4_drivers/linux_5.4_CY8C9520A_driver/app$ scp gpio_int.c
root@10.0.0.10:/home

root@raspberrypi:/home# gcc -o gpio_int gpio_int.c
```

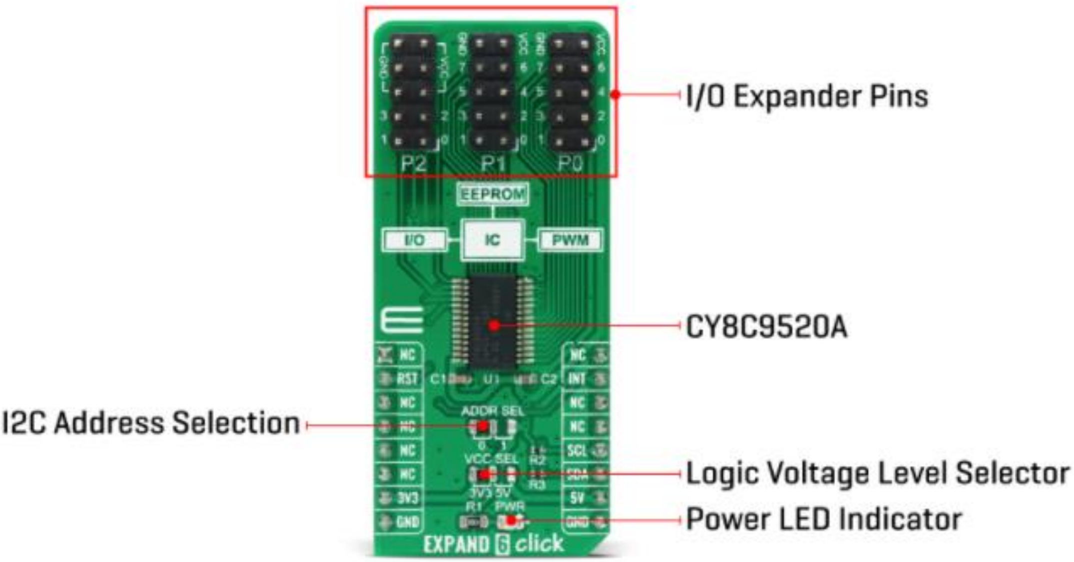
Follow the next instructions to test the drivers:

```
/* load the CY8C9520A module */
root@raspberrypi:/home# insmod CY8C9520A_rpi4.ko
[  157.763155] CY8C9520A_rpi4: loading out-of-tree module taints kernel.
[  157.773365] cy8c9520a 1-0020: cy8c9520a_probe() function is called
[  157.781876] cy8c9520a 1-0020: dev_id=0x20
[  157.804688] cy8c9520a 1-0020: the cy8c9520a_setup is done
[  157.813703] cy8c9520a 1-0020: the cy8c9520a_irq_setup function is entered
[  157.823866] cy8c9520a 1-0020: the interrupt state registers are cleared
[  157.840716] cy8c9520a 1-0020: the interrupt mask port registers are set
[  157.848193] cy8c9520a 1-0020: the interrupt setup is done

/* Print information of all the lines of the gpiochip2 */
root@raspberrypi:/home# gpioinfo gpiochip2
gpiochip2 - 20 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
        line   2:      unnamed       unused   input  active-high
        line   3:      unnamed       unused   input  active-high
        line   4:      unnamed       unused   input  active-high
        line   5:      unnamed       unused   input  active-high
        line   6:      unnamed       unused   input  active-high
        line   7:      unnamed       unused   input  active-high
        line   8:      unnamed       unused   input  active-high
        line   9:      unnamed       unused   input  active-high
```

```
line  10:      unnamed       unused   input  active-high
line  11:      unnamed       unused   input  active-high
line  12:      unnamed       unused   input  active-high
line  13:      unnamed       unused   input  active-high
line  14:      unnamed       unused   input  active-high
line  15:      unnamed       unused   input  active-high
line  16:      unnamed       unused   input  active-high
line  17:      unnamed       unused   input  active-high
line  18:      unnamed       unused   input  active-high
line  19:      unnamed       unused   input  active-high
```

Connect pin 0 to pin 1 on the P0 port of the I/O Expander board

/* the gpio lines of the gpiochip2 are configured with internal pull-up to Vcc */




```
/* set to high level the pin 1 of P0 */
root@raspberrypi:/home# gpioset gpiochip2 1=1
[  266.227650] cy8c9520a 1-0020: cy8c9520a_gpio_direction output is called
[  266.239696] cy8c9520a 1-0020: cy8c9520a_gpio_set_value func with 1 value is
called

/* check the value received in the pin 0 of P0 */
root@raspberrypi:/home# gpioget gpiochip2 0
[  285.287449] cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
[  285.299704] cy8c9520a 1-0020: cy8c9520a_gpio_get function is called
```

```
[  285.306084] cy8c9520a 1-0020: the in_reg address is 0
[  285.313172] cy8c9520a 1-0020: cy8c9520a_gpio_get function with 255 value is
returned
```
1

```
/* set to low level the pin 1 of P0 */
```
root@raspberrypi:/home# gpioset gpiochip2 1=0
```
[  325.605128] cy8c9520a 1-0020: cy8c9520a_gpio_direction output is called
[  325.617598] cy8c9520a 1-0020: cy8c9520a_gpio_set_value func with 0 value is
called
```

```
/* check the value received in the pin 0 of P0 */
```
root@raspberrypi:/home# gpioget gpiochip2 0
```
[  330.154964] cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
[  330.167169] cy8c9520a 1-0020: cy8c9520a_gpio_get function is called
[  330.173604] cy8c9520a 1-0020: the in_reg address is 0
[  330.180566] cy8c9520a 1-0020: cy8c9520a_gpio_get function with 252 value is
returned
```
0

Disconnect pin 0 and pin 1 on the P0 port of the I/O Expander pins. Handle GPIO INT in line 0 of P0 using the gpio interrupt driver

```
/* load the gpio interrupt module */
```
root@raspberrypi:/home# insmod int_rpi4_gpio.ko
```
[  374.403991] int_gpio_expand int_gpio: my_probe() function is called.
[  374.410657] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[  374.416604] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[  374.423205] cy8c9520a 1-0020: gpio 0 is unmasked
[  374.432014] the interrupt ISR has been entered
[  374.436629] cy8c9520a 1-0020: the REG_INTR_MASK value is 254
[  374.442456] int_gpio_expand int_gpio: IRQ_using_platform_get_irq: 61
[  374.448984] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[  374.454926] cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
[  374.460857] cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
[  374.466669] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[  374.473626] int_gpio_expand int_gpio: mydev: got minor 59
[  374.479168] int_gpio_expand int_gpio: my_probe() function is exited.
```

```
/* check the gpio interrupt with Linux IRQ number 61 */
```
root@raspberrypi:/home# cat /proc/interrupts
```
           CPU0       CPU1       CPU2       CPU3
 17:          1          0          0          0     GICv2  99 Level     timer
 18:          0          0          0          0     GICv2  29 Level
arch_timer
 19:       5696      13923       8410      17931     GICv2  30 Level
arch_timer
 26:        436          0          0          0     GICv2  65 Level
fe00b880.mailbox
```

```
 29:       6678          0          0          0     GICv2 153 Level     uart-
pl011
 30:          0          0          0          0     GICv2 150 Level
fe204000.spi
 31:        793          0          0          0     GICv2 125 Level     ttyS0
 32:         76          0          0          0     GICv2 149 Level
fe804000.i2c
 35:        352          0          0          0     GICv2 114 Level     DMA IRQ
 37:          0          0          0          0     GICv2 116 Level     DMA IRQ
 38:          0          0          0          0     GICv2 117 Level     DMA IRQ
 42:         59          0          0          0     GICv2  66 Level     VCHIQ
doorbell
 43:      10414          0          0          0     GICv2 158 Level     mmc1,
mmc0
 45:          0          0          0          0     GICv2  48 Level     arm-pmu
 46:          0          0          0          0     GICv2  49 Level     arm-pmu
 47:          0          0          0          0     GICv2  50 Level     arm-pmu
 48:          0          0          0          0     GICv2  51 Level     arm-pmu
 50:        823          0          0          0     GICv2 189 Level     eth0
 51:        155          0          0          0     GICv2 190 Level     eth0
 57:          0          0          0          0     GICv2 106 Level     v3d
 58:          0          0          0          0     GICv2 175 Level     PCIe PME,
aerdrv
 59:         38          0          0          0  BRCM STB PCIe MSI 524288 Edge
xhci_hcd
 60:          1          0          0          0  pinctrl-bcm2835  23 Level     1-
0020
 61:          0          0          0          0  cy8c9520a-irq   0 Edge
P0_line0_INT
IPI0:          0          0          0          0  CPU wakeup interrupts
IPI1:          0          0          0          0  Timer broadcast interrupts
IPI2:       1702       2513       4873       2247  Rescheduling interrupts
IPI3:        128        593        926        474  Function call interrupts
IPI4:          0          0          0          0  CPU stop interrupts
IPI5:        333        415        562        154  IRQ work interrupts
IPI6:          0          0          0          0  completion interrupts
Err:          0

/* Connect pin 0 of P0 to GND, then disconnect it from GND. Two interrupts are
fired */
root@raspberrypi:/home# [  472.674523] the interrupt ISR has been entered
[  472.681840] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  475.601337] the interrupt ISR has been entered
[  475.608693] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT

/* remove the gpio int module */
root@raspberrypi:/home# rmmod int_rpi4_gpio.ko
[  521.101163] int_gpio_expand int_gpio: my_remove() function is called.
[  521.110535] int_gpio_expand int_gpio: my_remove() function is exited.
```

```
[  521.117671] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[  521.123619] cy8c9520a 1-0020: cy8c9520a_irq_mask is called
[  521.129241] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[  521.135811] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[  521.141792] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called


/* remove the CY8C9520A module */
root@raspberrypi:/home# rmmod CY8C9520A_rpi4.ko
[  561.660986] cy8c9520a 1-0020: cy8c9520a_remove() function is called
```

Reboot the system. Handle GPIO INT in line 0 of P0 using a GPIO based interrupt application

```
/* load the CY8C9520A module */
root@raspberrypi:/home# insmod CY8C9520A_rpi4.ko
[   60.763246] CY8C9520A_rpi4: loading out-of-tree module taints kernel.
[   60.771424] cy8c9520a 1-0020: cy8c9520a_probe() function is called
[   60.779889] cy8c9520a 1-0020: dev_id=0x20
[   60.802703] cy8c9520a 1-0020: the cy8c9520a_setup is done
[   60.808670] cy8c9520a 1-0020: the cy8c9520a_irq_setup function is entered
[   60.821481] cy8c9520a 1-0020: the interrupt state registers are cleared
[   60.839226] cy8c9520a 1-0020: the interrupt mask port registers are set
[   60.846240] cy8c9520a 1-0020: the interrupt setup is done

/* Launch the gpiomon application */
root@raspberrypi:/home# gpiomon --falling-edge gpiochip2 0
[   41.094394] cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
[   41.106561] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[   41.112557] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[   41.119178] cy8c9520a 1-0020: gpio 0 is unmasked
[   41.129427] cy8c9520a 1-0020: the REG_INTR_MASK value is 254
[   41.135203] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[   41.141220] cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
[   41.147147] cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
[   41.152913] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called

/* Now connect pin 0 of P0 to GND. An interrupt is fired */
[   50.553632] the interrupt ISR has been entered
event: FALLING EDGE offset: 0 timestamp: [1606046305.553936360]

/* Disconnect pin 0 of P0 from GND. An interrupt is fired */
[   53.068682] the interrupt ISR has been entered
event: FALLING EDGE offset: 0 timestamp: [1606046308.068990655]

/* Exit application with Ctrl+C */
^C[   97.196572] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[   97.202658] cy8c9520a 1-0020: cy8c9520a_irq_mask is called
[   97.208274] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[   97.214839] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[   97.220825] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
```

```
/* Launch now the gpio_int application. Connect pin 0 of P0 to GND, then remove it
from GND. Two interrupts are fired */
root@raspberrypi:/home# ./gpio_int
[  135.605390] cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
[  135.617339] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[  135.623324] cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
[  135.629264] cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
[  135.635057] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[  146.377464] the interrupt ISR has been entered
[  146.384799] cy8c9520a 1-0020: cy8c9520a_gpio_get function is called
[  146.391236] cy8c9520a 1-0020: the in_reg address is 0
[  146.398250] cy8c9520a 1-0020: cy8c9520a_gpio_get function with 254 value is
returned
irq received.
id: 2, timestamp: 1606046401377764044
[  149.416517] the interrupt ISR has been entered
[  149.423884] cy8c9520a 1-0020: cy8c9520a_gpio_get function is called
[  149.430313] cy8c9520a 1-0020: the in_reg address is 0
[  149.437393] cy8c9520a 1-0020: cy8c9520a_gpio_get function with 255 value is
returned
irq received.
id: 1, timestamp: 1606046404416847616
```

LAB 7.5: "GPIO-PWM-PINCTRL expander device" module

The CY8C9520A_pwm_pinctrl driver that you will develop in this LAB 7.5, is an extension of the previous CY8C9520A_rpi4 driver, to which you will add new “pin controller” and “PWM controller” capabilities.

LAB 7.5 pin controller driver description

As described in Chapter 5 of this book, a pin controller is a peripheral of the processor that can configure pin hardware settings. It may be able to multiplex, bias, set load capacitance, set drive modes (pull up or down, open drain high/low, strong drive fast/slow, or high-impedance input), etc. for individual pins or groups of pins. The pin controller section of this driver will configure several drive modes for the CY8C9520A port data pins (pull up, pull down and strong drive).

On the software side, the Linux pinctrl framework configures and controls the microprocessor pins. There are two ways to use it:

- A pin (or group of pins) is controlled by a hardware block, then pinctrl will apply the pin configuration given by the device tree by calling specific vendor callback functions. This is the way that we will use in our lab driver.
- A pin needs to be controlled by software (typically a GPIO), then GPIOLib framework will be used to control this pin on top of pinctrl framework. For GPIOs that use pins known to the pinctrl subsystem, that subsystem should be informed of their use; a gpiolib driver's .request() operation may call pinctrl_request_gpio(), and a gpiolib driver's .free() operation may call pinctrl_free_gpio(). The pinctrl subsystem allows a pinctrl_request_gpio() to succeed concurrently with a pin or pingroup being "owned" by a device for pin multiplexing. The gpio and pin controllers are associated with each other through the pinctrl_add_gpio_range() function, which adds a range of GPIOs to be handled by a certain pin controller.

The first step during the development of our driver´s pinctrl code is to tell the pinctrl framework which pins the CY8C9520A device provides; that is a simple matter of enumerating their names and associating each with an integer pin number. You will create a pinctrl_pin_desc structure with the unique pin numbers from the global pin number space and the name for these pins. You have to use these names when you configure your device tree pin configuration nodes.

```
static const struct pinctrl_pin_desc cy8c9520a_pins[] = {
   PINCTRL_PIN(0, "gpio0"),
   PINCTRL_PIN(1, "gpio1"),
   PINCTRL_PIN(2, "gpio2"),
   PINCTRL_PIN(3, "gpio3"),
   PINCTRL_PIN(4, "gpio4"),
   PINCTRL_PIN(5, "gpio5"),
   PINCTRL_PIN(6, "gpio6"),
   PINCTRL_PIN(7, "gpio7"),
   PINCTRL_PIN(8, "gpio8"),
   PINCTRL_PIN(9, "gpio9"),
   PINCTRL_PIN(10, "gpio10"),
   PINCTRL_PIN(11, "gpio11"),
   PINCTRL_PIN(12, "gpio12"),
   PINCTRL_PIN(13, "gpio13"),
   PINCTRL_PIN(14, "gpio14"),
   PINCTRL_PIN(15, "gpio15"),
   PINCTRL_PIN(16, "gpio16"),
   PINCTRL_PIN(17, "gpio17"),
   PINCTRL_PIN(18, "gpio18"),
   PINCTRL_PIN(19, "gpio19"),
};
```

A pin controller is registered by filling in a pinctrl_desc structure and registering it to the pinctrl subsystem with the devm_pinctrl_register() function. See below the setup of the pintrl_desc structure, done in our driver´s probe() function.

```
cygpio->pinctrl_desc.name = "cy8c9520a-pinctrl";
cygpio->pinctrl_desc.pctlops = &cygpio_pinctrl_ops;
cygpio->pinctrl_desc.confops = &cygpio_pinconf_ops;
cygpio->pinctrl_desc.npins = cygpio->gpio_chip.ngpio;

cygpio->pinctrl_desc.pins = cy8c9520a_pins;
cygpio->pinctrl_desc.owner = THIS_MODULE;

cygpio->pctldev = devm_pinctrl_register(&client->dev,
                                        &cygpio->pinctrl_desc, cygpio);
```

The pctlops variable points to the custom cygpio_pinctrl_ops structure, which contains pointers to several callback functions. The pinconf_generic_dt_node_to_map_pin function will parse our device tree "pin configuration nodes", and creates mapping table entries for them. You will not implement the rest of the callback functions inside the pinctrl_ops structure.

```
static const struct pinctrl_ops cygpio_pinctrl_ops = {
   .get_groups_count = cygpio_pinctrl_get_groups_count,
   .get_group_name = cygpio_pinctrl_get_group_name,
   .get_group_pins = cygpio_pinctrl_get_group_pins,
#ifdef CONFIG_OF
   .dt_node_to_map = pinconf_generic_dt_node_to_map_pin,
   .dt_free_map = pinconf_generic_dt_free_map,
#endif
};
```

The confops variable points to the custom cygpio_pinconf_ops structure, which contains pointers to callback functions that perform pin config operations. You will only implement the cygpio_pinconf_set callback function, which sets the drive modes for all the gpios configured in our CY8C9520A´s device tree pin configuration nodes.

```
static const struct pinconf_ops cygpio_pinconf_ops = {
   .pin_config_set = cygpio_pinconf_set,
   .is_generic = true,
};
```

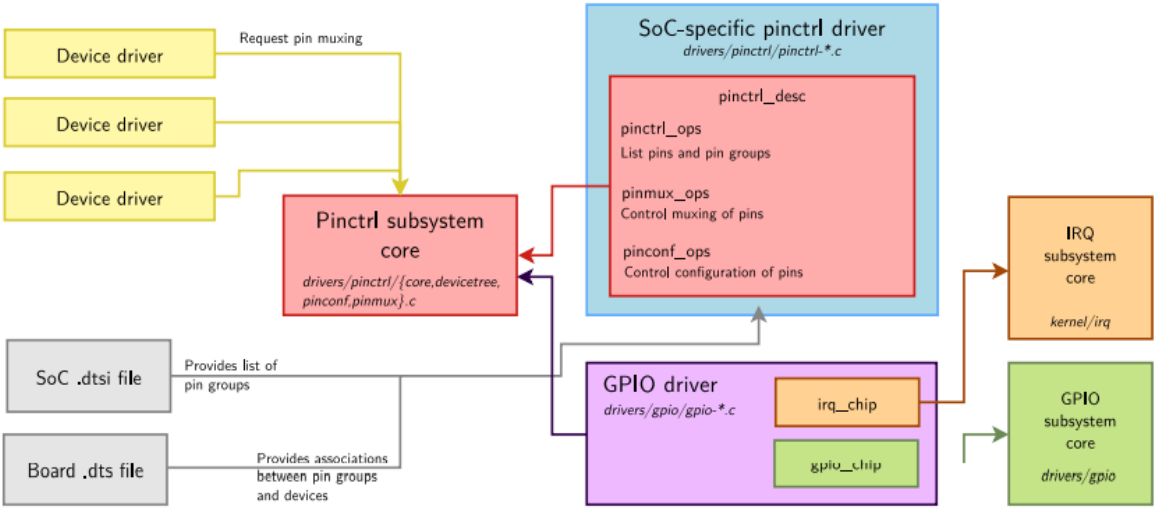
See below the code of the cygpio_pinconf_set callback function:

```
/* Configure the Drive Mode Register Settings */
static int cygpio_pinconf_set(struct pinctrl_dev *pctldev, unsigned int pin,
                        unsigned long *configs, unsigned int num_configs)
{
```

```
struct cy8c9520a *cygpio = pinctrl_dev_get_drvdata(pctldev);
struct i2c_client *client = cygpio->client;
enum pin_config_param param;
u32 arg;
int ret = 0;
int i;
u8 offs = 0;
u8 val = 0;
u8 port = cypress_get_port(pin);
u8 pin_offset = cypress_get_offs(pin, port);

mutex_lock(&cygpio->lock);

for (i = 0; i < num_configs; i++) {
        param = pinconf_to_config_param(configs[i]);
        arg = pinconf_to_config_argument(configs[i]);

        switch (param) {
        case PIN_CONFIG_BIAS_PULL_UP:
                offs = 0x0;
                break;
        case PIN_CONFIG_BIAS_PULL_DOWN:
                offs = 0x01;
                break;
        case PIN_CONFIG_DRIVE_STRENGTH:
                offs = 0x04;
                break;
        case PIN_CONFIG_BIAS_HIGH_IMPEDANCE:
                offs = 0x06;
                break;
        default:
                dev_err(&client->dev, "Invalid config param %04x\n", param);
                return -ENOTSUPP;
        }

        /* write to the REG_DRIVE registers of the CY8C9520A device */
        i2c_smbus_write_byte_data(client, REG_PORT_SELECT, port);

        i2c_smbus_read_byte_data(client, REG_DRIVE_PULLUP + offs);

        val = (u8)(ret | BIT(pin_offset));

        i2c_smbus_write_byte_data(client, REG_DRIVE_PULLUP + offs, val);

}

mutex_unlock(&cygpio->lock);
return ret;
```

```
}
```

In the following image, extracted from the Bootlin "Linux Kernel and Driver Development training" (https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf), you can see the pinctrl subsystem diagram. The image shows the location of the pinctrl´s main files and structures inside the kernel sources, and also the interaction between the pinctrl and GPIO drivers with the Pinctrl subsystem core. You can also see the interaction of the GPIO driver with the GPIO subsystem core and the IRQ subsystem core if the driver has interrupt capabilities, as is the case of our CY8C9520A driver.



Finally, you will add the following lines in bold to the device-tree configuration of our cy8c9520a device:

```
cy8c9520a: cy8c9520a@20 {
          compatible = "cy8c9520a";
          reg = <0x20>;
          interrupt-controller;
          #interrupt-cells = <2>;
          gpio-controller;
          #gpio-cells = <2>;

          interrupts = <23 1>;
          interrupt-parent = <&gpio>;
```

```
            pinctrl-names = "default";
            pinctrl-0 = <&accel_int_pin &cy8c9520apullups &cy8c9520apulldowns
&cy8c9520adrivestrength>;

            cy8c9520apullups: pinmux1 {
                    pins = "gpio0", "gpio1";
                    bias-pull-up;
            };

            cy8c9520apulldowns: pinmux2 {
                    pins = "gpio2";
                    bias-pull-down;
            };

            /* pwm channel */
            cy8c9520adrivestrength: pinmux3 {
                    pins = "gpio3";
                    drive-strength;
            };
    };
```

The pinctrl-x properties link to a pin configuration for a given state of the device. The pinctrl-names property associates a name to each state. In our driver, we will use only one state, and the name default is used for the pinctrl-names property. The name default is selected by our device driver without having to make a pinctrl function call.

In our DT device node, the pinctrl-0 property list several phandles, each of which points to a pin configuration node. These referenced pin configuration nodes must be child nodes of the pin controller that they configure. The first pin configuration node applies the pull-up configuration to the gpi0 and gpio1 pins (GPort 0, pins 0 and 1 of the CY8C9520A device). The second pin configuration node applies the pull-down configuration to the gpio2 pin (GPort 0, pin 2) and finally, the last pin configuration node applies the strong drive configuration to the gpio3 pin (GPort 0, pin 3). These pin configurations will be written to the CY8C9520A registers through the cygpio_pinconf_set callback function, which was previously described.

LAB 7.5 PWM controller driver description

The Linux PWM (Pulse Width Modulation) framework offers an interface that can be used from user space (sysfs) and kernel space (API) and allows to:

- control PWM output(s) such as period, duty cycle and polarity.
- capture a PWM signal and report its period and duty cycle.

This section will explain how to implement a PWM controller driver for our CY8C9520A device. As in other frameworks previously explained, there is a main structure that we have to configure and that will have to be registered to the PWM core. The name of this structure is pwm_chip and will be filled with a description of the PWM controller, the number of PWM devices provided by the controller, and the chip-specific callback functions which will support the PWM operations. You can see below the code that configures the pwm_chip structure inside our driver´s probe() function:

```
/* Setup of the pwm_chip controller */
cygpio->pwm_chip.dev = &client->dev;
cygpio->pwm_chip.ops = &cy8c9520a_pwm_ops;
cygpio->pwm_chip.base = PWM_BASE_ID;
cygpio->pwm_chip.npwm = NPWM;
```

The npwm variable sets the number of PWM channels. The CY8C9520A device has four PWM channels. The ops variable points to the cy8c9520a_pwm_ops structure, which includes pointers to the PWM chip-specific callback functions that will configure, enable and disable the PWM channels of the CY8C9520A device.

```
/* Declare the PWM callback functions */
static const struct pwm_ops cy8c9520a_pwm_ops = {
   .request = cy8c9520a_pwm_request,
   .config = cy8c9520a_pwm_config,
   .enable = cy8c9520a_pwm_enable,
   .disable = cy8c9520a_pwm_disable,
};
```

The cy8c9520a_pwm_config callback function will set up the period and the duty cycle for each PWM channel of the device. The cy8c9520a_pwm_enable and cy8c9520a_pwm_disable functions will enable/disable each PWM channel of the device. In the listing code of the driver, you can see the full code for these callback functions. These functions can be called from the user space using the sysfs method or from the kernel space (API) using a PWM user kernel driver. You will use the syfs method during the driver´s demonstration section.

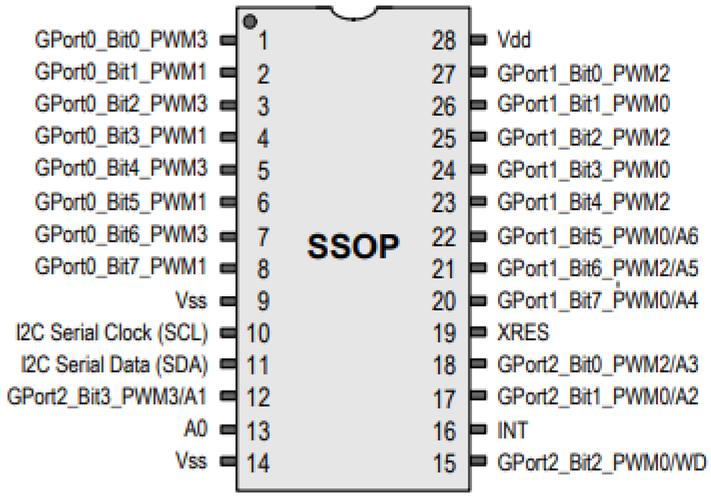
Finally, you will add the following lines in bold to the device-tree configuration of our cy8c9520a device:

```
cy8c9520a: cy8c9520a@20 {
          compatible = "cy8c9520a";
          reg = <0x20>;
          interrupt-controller;
          #interrupt-cells = <2>;
          gpio-controller;
          #gpio-cells = <2>;
```

```
            interrupts = <23 1>;
            interrupt-parent = <&gpio>;

            #pwm-cells = <2>;
            pwm0 = <20>; // pwm not supported
            pwm1 = <3>;
            pwm2 = <20>; // pwm not supported
            pwm3 = <2>;

            pinctrl-names = "default";
            pinctrl-0 = <&accel_int_pin &cy8c9520apullups &cy8c9520apulldowns
&cy8c9520adrivestrength>;

            cy8c9520apullups: pinmux1 {
                   pins = "gpio0", "gpio1";
                   bias-pull-up;
            };

            cy8c9520apulldowns: pinmux2 {
                   pins = "gpio2";
                   bias-pull-down;
            };

            /* pwm channel */
            cy8c9520adrivestrength: pinmux3 {
                   pins = "gpio3";
                   drive-strength;
            };
    };
```

The `pwmX` property will select the pin of the CY8C9520A device that will be configured as a PWM channel. You will select a pin for every PWM channel (PWM0 to PWM3) of the device. In the following image extracted from the data-sheet of the CY8C9520A device, you can see which PWM channel is associated with each port pin of the device. In our device tree, we will set the pwm1 channel to the Bit 3 (gpio3) of the GPort0 and the pwm3 channel to the bit 2 (gpio2) of the GPort0. If a PWM channel is not used, you will set its `pwmX` property to 20. This configuration is just an example, you can of course add your own configuration.

Figure 2. CY8C9520A 28-Pin Device



You will recover the values of the pwmX properties using the device_property_read_u32() function inside the probe() function.

```
/* parse the DT to get the pwm-pin mapping */
   for (i = 0; i < NPWM; i++) {
          ret = device_property_read_u32(&client->dev, name[i], &tmp);
          if (!ret)
                 cygpio->pwm_number[i] = tmp;
          else
                 goto err;
   };
```

See in the next **Listing 7-7** the complete "GPIO-PWM-PINCTRL expander device" driver source code for the Raspberry Pi 4 Model B processor. You can see in bold the lines that have been added to the "GPIO child device" driver.

Note: The "GPIO-PWM-PINCTRL expander device" driver source code developed for the Raspberry Pi 4 Model B board is included in the linux_5.4_rpi4_drivers.zip file under the linux_5.4_CY8C9520A_pwm_pinctrl folder, and can be downloaded from the GitHub repository at https://github.com/ALIBERA/linux_book_2nd_edition

Listing 7-7: CY8C9520A_pwm_pinctrl.c

```
#include <linux/i2c.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/gpio/driver.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/pwm.h>
#include <linux/slab.h>
#include <linux/pinctrl/pinctrl.h>
#include <linux/pinctrl/pinconf.h>
#include <linux/pinctrl/pinconf-generic.h>

#define DRV_NAME                "cy8c9520a"

/* cy8c9520a settings */
#define NGPIO                           20
#define DEVID_CY8C9520A                 0x20
#define NPORTS                          3
#define NPWM                            4
#define PWM_MAX_PERIOD                  0xff
#define PWM_BASE_ID                     0
#define PWM_CLK                         0x00
#define PWM_TCLK_NS                     31250  /* 32kHz */
#define PWM_UNUSED                      20

/* Register offset  */
#define REG_INPUT_PORT0                 0x00
#define REG_OUTPUT_PORT0                0x08
#define REG_INTR_STAT_PORT0             0x10
#define REG_PORT_SELECT                 0x18
#define REG_INTR_MASK                   0x19
#define REG_PIN_DIR                     0x1c
#define REG_DRIVE_PULLUP                0x1d
#define REG_DRIVE_PULLDOWN              0x1e
#define REG_DEVID_STAT                  0x2e

/* Register PWM */
#define REG_SELECT_PWM                  0x1a
#define REG_PWM_SELECT                  0x28
#define REG_PWM_CLK                     0x29
#define REG_PWM_PERIOD                  0x2a
#define REG_PWM_PULSE_W                 0x2b

/* definition of the global structure for the driver */
struct cy8c9520a {
   struct i2c_client     *client;
```

```
    struct gpio_chip      gpio_chip;
    struct pwm_chip       pwm_chip;
    struct gpio_desc      *gpio;
    int                   irq;
    struct mutex          lock;
    /* protect serialized access to the interrupt controller bus */
    struct mutex          irq_lock;
    /* cached output registers */
    u8                    outreg_cache[NPORTS];
    /* cached IRQ mask */
    u8                    irq_mask_cache[NPORTS];
    /* IRQ mask to be applied */
    u8                    irq_mask[NPORTS];
    int                   pwm_number[NPWM];

    struct pinctrl_dev    *pctldev;
    struct pinctrl_desc   pinctrl_desc;
};

/* Per-port GPIO offset */
static const u8 cy8c9520a_port_offs[] = {
    0,
    8,
    16,
};

static const struct pinctrl_pin_desc cy8c9520a_pins[] = {
    PINCTRL_PIN(0, "gpio0"),
    PINCTRL_PIN(1, "gpio1"),
    PINCTRL_PIN(2, "gpio2"),
    PINCTRL_PIN(3, "gpio3"),
    PINCTRL_PIN(4, "gpio4"),
    PINCTRL_PIN(5, "gpio5"),
    PINCTRL_PIN(6, "gpio6"),
    PINCTRL_PIN(7, "gpio7"),
    PINCTRL_PIN(8, "gpio8"),
    PINCTRL_PIN(9, "gpio9"),
    PINCTRL_PIN(10, "gpio10"),
    PINCTRL_PIN(11, "gpio11"),
    PINCTRL_PIN(12, "gpio12"),
    PINCTRL_PIN(13, "gpio13"),
    PINCTRL_PIN(14, "gpio14"),
    PINCTRL_PIN(15, "gpio15"),
    PINCTRL_PIN(16, "gpio16"),
    PINCTRL_PIN(17, "gpio17"),
    PINCTRL_PIN(18, "gpio18"),
    PINCTRL_PIN(19, "gpio19"),
};
```

```
/* return the port of the gpio */
static inline u8 cypress_get_port(unsigned int gpio)
{
   u8 i = 0;
   for (i = 0; i < sizeof(cy8c9520a_port_offs) - 1; i ++) {
          if (! (gpio / cy8c9520a_port_offs[i + 1]))
                 break;
   }
   return i;
}

/* get the gpio offset inside its respective port */
static inline u8 cypress_get_offs(unsigned gpio, u8 port)
{
   return gpio - cy8c9520a_port_offs[port];
}

static int cygpio_pinctrl_get_groups_count(struct pinctrl_dev *pctldev)
{
   return 0;
}

static const char *cygpio_pinctrl_get_group_name(struct pinctrl_dev *pctldev,
                                                 unsigned int group)
{
   return NULL;
}

static int cygpio_pinctrl_get_group_pins(struct pinctrl_dev *pctldev,
                                         unsigned int group,
                                         const unsigned int **pins,
                                         unsigned int *num_pins)
{
   return -ENOTSUPP;
}

/*
 * global pin control operations, to be implemented by
 * pin controller drivers
 * pinconf_generic_dt_node_to_map_pin function
 * will parse a device tree "pin configuration node", and create
 * mapping table entries for it
 */
static const struct pinctrl_ops cygpio_pinctrl_ops = {
   .get_groups_count = cygpio_pinctrl_get_groups_count,
   .get_group_name = cygpio_pinctrl_get_group_name,
   .get_group_pins = cygpio_pinctrl_get_group_pins,
```

```
#ifdef CONFIG_OF
	.dt_node_to_map = pinconf_generic_dt_node_to_map_pin,
	.dt_free_map = pinconf_generic_dt_free_map,
#endif
};

/* Configure the Drive Mode Register Settings */
static int cygpio_pinconf_set(struct pinctrl_dev *pctldev, unsigned int pin,
			      unsigned long *configs, unsigned int num_configs)
{
	struct cy8c9520a *cygpio = pinctrl_dev_get_drvdata(pctldev);
	struct i2c_client *client = cygpio->client;
	enum pin_config_param param;
	u32 arg;
	int ret = 0;
	int i;
	u8 offs = 0;
	u8 val = 0;
	u8 port = cypress_get_port(pin);
	u8 pin_offset = cypress_get_offs(pin, port);

	dev_err(&client->dev, "cygpio_pinconf_set function is called\n");

	mutex_lock(&cygpio->lock);

	for (i = 0; i < num_configs; i++) {
		param = pinconf_to_config_param(configs[i]);
		arg = pinconf_to_config_argument(configs[i]);

		switch (param) {
		case PIN_CONFIG_BIAS_PULL_UP:
			offs = 0x0;
			dev_info(&client->dev,
				 "The pin %d drive mode is PIN_CONFIG_BIAS_PULL_UP\n",
				  pin);
			break;
		case PIN_CONFIG_BIAS_PULL_DOWN:
			offs = 0x01;
			dev_info(&client->dev,
				 "The pin %d drive mode is PIN_CONFIG_BIAS_PULL_DOWN\n",
				 pin);
			break;
		case PIN_CONFIG_DRIVE_STRENGTH:
			offs = 0x04;
			dev_info(&client->dev,
				 "The pin %d drive mode is PIN_CONFIG_DRIVE_STRENGTH\n",
				 pin);
			break;
```

```
	case PIN_CONFIG_BIAS_HIGH_IMPEDANCE:
		offs = 0x06;
		dev_info(&client->dev,
			"The pin %d drive mode is
PIN_CONFIG_BIAS_HIGH_IMPEDANCE\n", pin);
		break;
	default:
		dev_err(&client->dev, "Invalid config param %04x\n", param);
		return -ENOTSUPP;
	}

	ret = i2c_smbus_write_byte_data(client, REG_PORT_SELECT, port);
	if (ret < 0) {
		dev_err(&client->dev, "can't select port %u\n", port);
		goto end;
	}

	ret = i2c_smbus_read_byte_data(client, REG_DRIVE_PULLUP + offs);
	if (ret < 0) {
		dev_err(&client->dev, "can't read pin direction\n");
		goto end;
	}

	val = (u8)(ret | BIT(pin_offset));

	ret = i2c_smbus_write_byte_data(client, REG_DRIVE_PULLUP + offs, val);
	if (ret < 0) {
		dev_err(&client->dev, "can't set drive mode port %u\n", port);
		goto end;
	}

    }

end:
    mutex_unlock(&cygpio->lock);
    return ret;
}

/*
 * pin config operations, to be implemented by
 * pin configuration capable drivers
 * pin_config_set: configure an individual pin
 */
static const struct pinconf_ops cygpio_pinconf_ops = {
    .pin_config_set = cygpio_pinconf_set,
    .is_generic = true,
};
```

```
/*
 * struct gpio_chip get callback function.
 * It gets the input value of the GPIO line (0=low, 1=high)
 * accessing to the REG_INPUT_PORT register
 */
static int cy8c9520a_gpio_get(struct gpio_chip *chip,
                              unsigned int gpio)
{
   int ret;
   u8 port, in_reg;

   struct cy8c9520a *cygpio = gpiochip_get_data(chip);

   dev_info(chip->parent, "cy8c9520a_gpio_get function is called\n");

   /* get the input port address address (in_reg) for the GPIO */
   port = cypress_get_port(gpio);
   in_reg = REG_INPUT_PORT0 + port;

   dev_info(chip->parent, "the in_reg address is %u\n", in_reg);

   mutex_lock(&cygpio->lock);

   ret = i2c_smbus_read_byte_data(cygpio->client, in_reg);
   if (ret < 0) {
          dev_err(chip->parent, "can't read input port %u\n", in_reg);
   }

   dev_info(chip->parent,
           "cy8c9520a_gpio_get function with %d value is returned\n",
           ret);

   mutex_unlock(&cygpio->lock);

   /*
    * check the status of the GPIO in its input port register
    * and return it. If expression is not 0 returns 1
    */
   return !!(ret & BIT(cypress_get_offs(gpio, port)));
}

/*
 * struct gpio_chip set callback function.
 * It sets the output value of the GPIO line in
 * GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the REG_OUTPUT_PORT register
 */
```

```
static void cy8c9520a_gpio_set(struct gpio_chip *chip,
                               unsigned int gpio, int val)
{
   int ret;
   u8 port, out_reg;
   struct cy8c9520a *cygpio = gpiochip_get_data(chip);

   dev_info(chip->parent,
           "cy8c9520a_gpio_set_value func with %d value is called\n",
           val);

   /* get the output port address address (out_reg) for the GPIO */
   port = cypress_get_port(gpio);
   out_reg = REG_OUTPUT_PORT0 + port;

   mutex_lock(&cygpio->lock);

   /*
    * if val is 1, gpio output level is high
    * if val is 0, gpio output level is low
    * the output registers were previously cached in cy8c9520a_setup()
    */
   if (val) {
          cygpio->outreg_cache[port] |= BIT(cypress_get_offs(gpio, port));
   } else {
          cygpio->outreg_cache[port] &= ~BIT(cypress_get_offs(gpio, port));
   }

   ret = i2c_smbus_write_byte_data(cygpio->client, out_reg,
                                   cygpio->outreg_cache[port]);
   if (ret < 0) {
          dev_err(chip->parent, "can't write output port %u\n", port);
   }

   mutex_unlock(&cygpio->lock);
}

/*
 * struct gpio_chip direction_output callback function.
 * It configures the GPIO as an output writing to
 * the REG_PIN_DIR register of the selected port
 */
static int cy8c9520a_gpio_direction_output(struct gpio_chip *chip,
                                           unsigned int gpio, int val)
{
   int ret;
   u8 pins, port;
```

```
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    /* gets the port number of the gpio */
    port = cypress_get_port(gpio);

    dev_info(chip->parent, "cy8c9520a_gpio_direction output is called\n");

    mutex_lock(&cygpio->lock);

    /* select the port where we want to config the GPIO as output */
    ret = i2c_smbus_write_byte_data(cygpio->client, REG_PORT_SELECT, port);
    if (ret < 0) {
           dev_err(chip->parent, "can't select port %u\n", port);
           goto err;
    }

    ret = i2c_smbus_read_byte_data(cygpio->client, REG_PIN_DIR);
    if (ret < 0) {
           dev_err(chip->parent, "can't read pin direction\n");
           goto err;
    }

    /* simply transform int to u8 */
    pins = (u8)ret & 0xff;

    /* add the direction of the new pin. Set 1 if input and set 0 is output */
    pins &= ~BIT(cypress_get_offs(gpio, port));

    ret = i2c_smbus_write_byte_data(cygpio->client, REG_PIN_DIR, pins);
    if (ret < 0) {
           dev_err(chip->parent, "can't write pin direction\n");
    }

err:
    mutex_unlock(&cygpio->lock);
    cy8c9520a_gpio_set(chip, gpio, val);
    return ret;
}

/*
 * struct gpio_chip direction_input callback function.
 * It configures the GPIO as an input writing to
 * the REG_PIN_DIR register of the selected port
 */
static int cy8c9520a_gpio_direction_input(struct gpio_chip *chip,
                                          unsigned int gpio)
{
    int ret;
```

```
	u8 pins, port;

	struct cy8c9520a *cygpio = gpiochip_get_data(chip);

	/* gets the port number of the gpio */
	port = cypress_get_port(gpio);

	dev_info(chip->parent, "cy8c9520a_gpio_direction input is called\n");

	mutex_lock(&cygpio->lock);

	/* select the port where we want to config the GPIO as input */
	ret = i2c_smbus_write_byte_data(cygpio->client, REG_PORT_SELECT, port);
	if (ret < 0) {
		dev_err(chip->parent, "can't select port %u\n", port);
		goto err;
	}

	ret = i2c_smbus_read_byte_data(cygpio->client, REG_PIN_DIR);
	if (ret < 0) {
		dev_err(chip->parent, "can't read pin direction\n");
		goto err;
	}

	/* simply transform int to u8 */
	pins = (u8)ret & 0xff;

	/*
	 * add the direction of the new pin.
	 * Set 1 if input (out == 0) and set 0 is ouput (out == 1)
	 */
	pins |= BIT(cypress_get_offs(gpio, port));

	ret = i2c_smbus_write_byte_data(cygpio->client, REG_PIN_DIR, pins);
	if (ret < 0) {
		dev_err(chip->parent, "can't write pin direction\n");
		goto err;
	}

err:
	mutex_unlock(&cygpio->lock);
	return ret;
}

/* function to lock access to slow bus (i2c) chips */
static void cy8c9520a_irq_bus_lock(struct irq_data *d)
{
	struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
```

```
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    dev_info(chip->parent, "cy8c9520a_irq_bus_lock is called\n");
    mutex_lock(&cygpio->irq_lock);
}

/*
 * function to sync and unlock slow bus (i2c) chips
 * REG_INTR_MASK register is accessed via I2C
 * write 0 to the interrupt mask register line to
 * activate the interrupt on the GPIO
 */
static void cy8c9520a_irq_bus_sync_unlock(struct irq_data *d)
{
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    int ret, i;
    unsigned int gpio;
    u8 port;
    dev_info(chip->parent, "cy8c9520a_irq_bus_sync_unlock is called\n");
    gpio = d->hwirq;
    port = cypress_get_port(gpio);

    /* irq_mask_cache stores the last value of irq_mask for each port */
    for (i = 0; i < NPORTS; i++) {
            /*
             * check if some of the bits have changed from the last cached value
             * irq_mask registers were initialized in cy8c9520a_irq_setup()
             */
            if (cygpio->irq_mask_cache[i] ^ cygpio->irq_mask[i]) {
                    dev_info(chip->parent, "gpio %u is unmasked\n", gpio);
                    cygpio->irq_mask_cache[i] = cygpio->irq_mask[i];
                    ret = i2c_smbus_write_byte_data(cygpio->client,
                                                    REG_PORT_SELECT, i);
                    if (ret < 0) {
                            dev_err(chip->parent, "can't select port %u\n", port);
                            goto err;
                    }

                    /* enable the interrupt for the GPIO unmasked */
                    ret = i2c_smbus_write_byte_data(cygpio->client, REG_INTR_MASK,
                                                    cygpio->irq_mask[i]);
                    if (ret < 0) {
                            dev_err(chip->parent,
                                    "can't write int mask on port %u\n", port);
                            goto err;
                    }

                    ret = i2c_smbus_read_byte_data(cygpio->client, REG_INTR_MASK);
```

```
                  dev_info(chip->parent, "the REG_INTR_MASK value is %d\n", ret);

           }
    }

err:
    mutex_unlock(&cygpio->irq_lock);
}

/*
 * mask (disable) the GPIO interrupt.
 * In the initial setup all the int lines are masked
 */
static void cy8c9520a_irq_mask(struct irq_data *d)
{
    u8 port;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    unsigned gpio = d->hwirq;
    port = cypress_get_port(gpio);
    dev_info(chip->parent, "cy8c9520a_irq_mask is called\n");

    cygpio->irq_mask[port] |= BIT(cypress_get_offs(gpio, port));
}

/*
 * unmask (enable) the GPIO interrupt.
 * In the initial setup all the int lines are masked
 */
static void cy8c9520a_irq_unmask(struct irq_data *d)
{
    u8 port;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    unsigned gpio = d->hwirq;
    port = cypress_get_port(gpio);
    dev_info(chip->parent, "cy8c9520a_irq_unmask is called\n");

    cygpio->irq_mask[port] &= ~BIT(cypress_get_offs(gpio, port));
}

/* set the flow type (IRQ_TYPE_LEVEL/etc.) of the IRQ */
static int cy8c9520a_irq_set_type(struct irq_data *d, unsigned int type)
{
    int ret = 0;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
```

```
	dev_info(chip->parent, "cy8c9520a_irq_set_type is called\n");

	if ((type != IRQ_TYPE_EDGE_BOTH) && (type != IRQ_TYPE_EDGE_FALLING)) {
		dev_err(&cygpio->client->dev,
			"irq %d: unsupported type %d\n",
			 d->irq, type);
		ret = -EINVAL;
		goto err;
	}

err:
	return ret;
}

/* Iinitialization of the irq_chip structure with callback functions */
static struct irq_chip cy8c9520a_irq_chip = {
	.name                 = "cy8c9520a-irq",
	.irq_mask             = cy8c9520a_irq_mask,
	.irq_unmask           = cy8c9520a_irq_unmask,
	.irq_bus_lock         = cy8c9520a_irq_bus_lock,
	.irq_bus_sync_unlock  = cy8c9520a_irq_bus_sync_unlock,
	.irq_set_type         = cy8c9520a_irq_set_type,
};

/*
 * interrupt handler for the cy8c9520a. It is called when
 * there is a rising or falling edge in the unmasked GPIO
 */
static irqreturn_t cy8c9520a_irq_handler(int irq, void *devid)
{
	struct cy8c9520a *cygpio = devid;
	u8 stat[NPORTS], pending;
	unsigned port, gpio, gpio_irq;
	int ret;

	pr_info ("the interrupt ISR has been entered\n");

	/*
	 * store in stat and clear (to enable ints)
	 * the three interrupt status registers by reading them
	 */
	ret = i2c_smbus_read_i2c_block_data(cygpio->client,
					    REG_INTR_STAT_PORT0,
					    NPORTS, stat);
	if (ret < 0) {
		memset(stat, 0, sizeof(stat));
	}
```

```
	ret = IRQ_NONE;

	for (port = 0; port < NPORTS; port ++) {
		mutex_lock(&cygpio->irq_lock);

		/*
		 * In every port check the GPIOs that have their int unmasked
		 * and whose bits have been enabled in their REG_INTR_STAT_PORT
		 * register due to an interrupt in the GPIO, and store the new
		 * value in the pending register
		 */
		pending = stat[port] & (~cygpio->irq_mask[port]);
		mutex_unlock(&cygpio->irq_lock);

		/* Launch the ISRs of all the gpios that requested an interrupt */
		while (pending) {
			ret = IRQ_HANDLED;
			/* get the first gpio that has got an int */
			gpio = __ffs(pending);

			/* clears the gpio in the pending register */
			pending &= ~BIT(gpio);

			/* gets the int number associated to this gpio */
			gpio_irq = cy8c9520a_port_offs[port] + gpio;

			/* launch the ISR of the GPIO child driver */
			handle_nested_irq(irq_find_mapping(cygpio->gpio_chip.irq.domain,
							   gpio_irq));

		}
	}

	return ret;
}

/*
 * select the period and the duty cycle of the PWM signal (in nanoseconds)
 * echo 100000 > pwm1/period
 * echo 50000 > pwm1/duty_cycle
 */
static int cy8c9520a_pwm_config(struct pwm_chip *chip, struct pwm_device *pwm,
				int duty_ns, int period_ns)
{
	int ret;
	int period = 0, duty = 0;

	struct cy8c9520a *cygpio =
```

```
		container_of(chip, struct cy8c9520a, pwm_chip);
	struct i2c_client *client = cygpio->client;

	dev_info(&client->dev, "cy8c9520a_pwm_config is called\n");

	if (pwm->pwm > NPWM) {
		return -EINVAL;
	}

	period = period_ns / PWM_TCLK_NS;
	duty = duty_ns / PWM_TCLK_NS;

	/*
	 * Check period's upper bound.  Note the duty cycle is already sanity
	 * checked by the PWM framework.
	 */
	if (period > PWM_MAX_PERIOD) {
		dev_err(&client->dev, "period must be within [0-%d]ns\n",
			PWM_MAX_PERIOD * PWM_TCLK_NS);
		return -EINVAL;
	}

	mutex_lock(&cygpio->lock);

	/*
	 * select the pwm number (from 0 to 3)
	 * to set the period and the duty for the enabled pwm pins
	 */
	ret = i2c_smbus_write_byte_data(client, REG_PWM_SELECT, (u8)pwm->pwm);
	if (ret < 0) {
		dev_err(&client->dev, "can't write to REG_PWM_SELECT\n");
		goto end;
	}

	ret = i2c_smbus_write_byte_data(client, REG_PWM_PERIOD, (u8)period);
	if (ret < 0) {
		dev_err(&client->dev, "can't write to REG_PWM_PERIOD\n");
		goto end;
	}

	ret = i2c_smbus_write_byte_data(client, REG_PWM_PULSE_W, (u8)duty);
	if (ret < 0) {
		dev_err(&client->dev, "can't write to REG_PWM_PULSE_W\n");
		goto end;
	}

end:
	mutex_unlock(&cygpio->lock);
```

```
    return ret;
}

/*
 * Enable the PWM signal
 * echo 1 > pwm1/enable
 */
static int cy8c9520a_pwm_enable(struct pwm_chip *chip, struct pwm_device *pwm)
{
    int ret, gpio, port, pin;
    u8 out_reg, val;

    struct cy8c9520a *cygpio =
        container_of(chip, struct cy8c9520a, pwm_chip);
    struct i2c_client *client = cygpio->client;

    dev_info(&client->dev, "cy8c9520a_pwm_enable is called\n");

    if (pwm->pwm > NPWM) {
           return -EINVAL;
    }

    /*
     * get the pin configured as pwm in the device tree
     * for this pwm port (pwm_device)
     */
    gpio = cygpio->pwm_number[pwm->pwm];
    port = cypress_get_port(gpio);
    pin = cypress_get_offs(gpio, port);
    out_reg = REG_OUTPUT_PORT0 + port;

    /*
     * Set pin as output driving high and select the port
     * where the pwm will be set
     */
    ret = cy8c9520a_gpio_direction_output(&cygpio->gpio_chip, gpio, 1);
    if (val < 0) {
           dev_err(&client->dev, "can't set pwm%u as output\n", pwm->pwm);
           return ret;
    }

    mutex_lock(&cygpio->lock);

    /* Enable PWM pin in the selected port */
    val = i2c_smbus_read_byte_data(client, REG_SELECT_PWM);
    if (val < 0) {
           dev_err(&client->dev, "can't read REG_SELECT_PWM\n");
```

```
        ret = val;
        goto end;
    }
    val |= BIT((u8)pin);
    ret = i2c_smbus_write_byte_data(client, REG_SELECT_PWM, val);
    if (ret < 0) {
        dev_err(&client->dev, "can't write to SELECT_PWM\n");
        goto end;
    }

end:
    mutex_unlock(&cygpio->lock);

    return ret;
}

/*
 * Disable the PWM signal
 * echo 0 > pwm1/enable
 */
static void cy8c9520a_pwm_disable(struct pwm_chip *chip, struct pwm_device *pwm)
{
    int ret, gpio, port, pin;
    u8 val;

    struct cy8c9520a *cygpio =
        container_of(chip, struct cy8c9520a, pwm_chip);
    struct i2c_client *client = cygpio->client;

    dev_info(&client->dev, "cy8c9520a_pwm_disable is called\n");

    if (pwm->pwm > NPWM) {
        return;
    }

    gpio = cygpio->pwm_number[pwm->pwm];
    if (PWM_UNUSED == gpio) {
        dev_err(&client->dev, "pwm%d is unused\n", pwm->pwm);
        return;
    }

    port = cypress_get_port(gpio);
    pin = cypress_get_offs(gpio, port);

    mutex_lock(&cygpio->lock);

    /* Disable PWM */
    val = i2c_smbus_read_byte_data(client, REG_SELECT_PWM);
```

```
    if (val < 0) {
            dev_err(&client->dev, "can't read REG_SELECT_PWM\n");
            goto end;
    }
    val &= ~BIT((u8)pin);
    ret = i2c_smbus_write_byte_data(client, REG_SELECT_PWM, val);
    if (ret < 0) {
            dev_err(&client->dev, "can't write to SELECT_PWM\n");
    }

end:
    mutex_unlock(&cygpio->lock);

    return;
}

/*
 * Request the PWM device
 * echo 0 > export
 */
static int cy8c9520a_pwm_request(struct pwm_chip *chip, struct pwm_device *pwm)
{
    int gpio = 0;
    struct cy8c9520a *cygpio =
        container_of(chip, struct cy8c9520a, pwm_chip);
    struct i2c_client *client = cygpio->client;

    dev_info(&client->dev, "cy8c9520a_pwm_request is called\n");

    if (pwm->pwm > NPWM) {
            return -EINVAL;
    }

    gpio = cygpio->pwm_number[pwm->pwm];
    if (PWM_UNUSED == gpio) {
            dev_err(&client->dev, "pwm%d unavailable\n", pwm->pwm);
            return -EINVAL;
    }

    return 0;
}

/* Declare the PWM callback functions */
static const struct pwm_ops cy8c9520a_pwm_ops = {
    .request = cy8c9520a_pwm_request,
    .config = cy8c9520a_pwm_config,
    .enable = cy8c9520a_pwm_enable,
    .disable = cy8c9520a_pwm_disable,
```

```
};

/* Initial setup for the cy8c9520a */
static int cy8c9520a_setup(struct cy8c9520a *cygpio)
{
	int ret, i;
	struct i2c_client *client = cygpio->client;

	/* Disable PWM, set all GPIOs as input.  */
	for (i = 0; i < NPORTS; i ++) {
		ret = i2c_smbus_write_byte_data(client, REG_PORT_SELECT, i);
		if (ret < 0) {
			dev_err(&client->dev, "can't select port %u\n", i);
			goto end;
		}

		ret = i2c_smbus_write_byte_data(client, REG_SELECT_PWM, 0x00);
		if (ret < 0) {
			dev_err(&client->dev, "can't write to SELECT_PWM\n");
			goto end;
		}

		ret = i2c_smbus_write_byte_data(client, REG_PIN_DIR, 0xff);
		if (ret < 0) {
			dev_err(&client->dev, "can't write to PIN_DIR\n");
			goto end;
		}
	}

	/* Cache the output registers (Output Port 0, Output Port 1, Output Port 2) */
	ret = i2c_smbus_read_i2c_block_data(client, REG_OUTPUT_PORT0,
					    sizeof(cygpio->outreg_cache),
					    cygpio->outreg_cache);
	if (ret < 0) {
		dev_err(&client->dev, "can't cache output registers\n");
		goto end;
	}

	/* Set default PWM clock source.  */
	for (i = 0; i < NPWM; i ++) {
		ret = i2c_smbus_write_byte_data(client, REG_PWM_SELECT, i);
		if (ret < 0) {
			dev_err(&client->dev, "can't select pwm %u\n", i);
			goto end;
		}

		ret = i2c_smbus_write_byte_data(client, REG_PWM_CLK, PWM_CLK);
		if (ret < 0) {
```

```
                dev_err(&client->dev, "can't write to REG_PWM_CLK\n");
                goto end;
        }
    }

    dev_info(&client->dev, "the cy8c9520a_setup is done\n");

end:
    return ret;
}

/* Interrupt setup for the cy8c9520a */
static int cy8c9520a_irq_setup(struct cy8c9520a *cygpio)
{
    struct i2c_client *client = cygpio->client;
    struct gpio_chip *chip = &cygpio->gpio_chip;
    u8 dummy[NPORTS];
    int ret, i;

    mutex_init(&cygpio->irq_lock);

    dev_info(&client->dev, "the cy8c9520a_irq_setup function is entered\n");

    /*
     * Clear interrupt state registers by reading the three registers
     * Interrupt Status Port0, Interrupt Status Port1, Interrupt Status Port2,
     * and store the values in a dummy array
     */
    ret = i2c_smbus_read_i2c_block_data(client, REG_INTR_STAT_PORT0,
                                        NPORTS, dummy);
    if (ret < 0) {
            dev_err(&client->dev, "couldn't clear int status\n");
            goto err;
    }

    dev_info(&client->dev, "the interrupt state registers are cleared\n");

    /*
     * Initialise Interrupt Mask Port Register (19h) for each port
     * Disable the activation of the INT lines. Each 1 in this
     * register masks (disables) the int from the corresponding GPIO
     */
    memset(cygpio->irq_mask_cache, 0xff, sizeof(cygpio->irq_mask_cache));
    memset(cygpio->irq_mask, 0xff, sizeof(cygpio->irq_mask));

    /* Disable interrupts in all the gpio lines */
    for (i = 0; i < NPORTS; i++) {
            ret = i2c_smbus_write_byte_data(client, REG_PORT_SELECT, i);
```

```
        if (ret < 0) {
                dev_err(&client->dev, "can't select port %u\n", i);
                goto err;
        }

        ret = i2c_smbus_write_byte_data(client, REG_INTR_MASK,
                                        cygpio->irq_mask[i]);
        if (ret < 0) {
                dev_err(&client->dev,
                        "can't write int mask on port %u\n", i);
                goto err;
        }
}

dev_info(&client->dev, "the interrupt mask port registers are set\n");

/* add a nested irqchip to the gpiochip */
ret =  gpiochip_irqchip_add_nested(chip,
                                     &cy8c9520a_irq_chip,
                                     0,
                                     handle_simple_irq,
                                     IRQ_TYPE_NONE);
if (ret) {
        dev_err(&client->dev,
                "could not connect irqchip to gpiochip\n");
        return ret;
}

/*
 * Request interrupt on a GPIO pin of the external processor
 * this processor pin is connected to the INT pin of the cy8c9520a
 */
ret = devm_request_threaded_irq(&client->dev, client->irq, NULL,
                                  cy8c9520a_irq_handler,
                                  IRQF_ONESHOT | IRQF_TRIGGER_HIGH,
                                  dev_name(&client->dev), cygpio);
if (ret) {
        dev_err(&client->dev, "failed to request irq %d\n", cygpio->irq);
                return ret;
}

/*
 * set up a nested irq handler for a gpio_chip from a parent IRQ
 * you can now request interrupts from GPIO child drivers nested
 * to the cy8c9520a driver
 */
gpiochip_set_nested_irqchip(chip,
                             &cy8c9520a_irq_chip,
```

```
                              cygpio->irq);

   dev_info(&client->dev, "the interrupt setup is done\n");

   return 0;
err:
   mutex_destroy(&cygpio->irq_lock);
   return ret;
}

/*
 * Initialize the cy8c9520a gpio controller (struct gpio_chip)
 * and register it to the kernel
 */
static int cy8c9520a_gpio_init(struct cy8c9520a *cygpio)
{
   struct gpio_chip *gpiochip = &cygpio->gpio_chip;
   int err;

   gpiochip->label = cygpio->client->name;
   gpiochip->base = -1;
   gpiochip->ngpio = NGPIO;
   gpiochip->parent = &cygpio->client->dev;
   gpiochip->of_node = gpiochip->parent->of_node;
   gpiochip->can_sleep = true;
   gpiochip->direction_input = cy8c9520a_gpio_direction_input;
   gpiochip->direction_output = cy8c9520a_gpio_direction_output;
   gpiochip->get = cy8c9520a_gpio_get;
   gpiochip->set = cy8c9520a_gpio_set;
   gpiochip->owner = THIS_MODULE;

   /* register a gpio_chip */
   err = devm_gpiochip_add_data(gpiochip->parent, gpiochip, cygpio);
   if (err)
          return err;
   return 0;
}

static int cy8c9520a_probe(struct i2c_client *client,
                           const struct i2c_device_id *id)
{
   struct cy8c9520a *cygpio;
   int ret = 0;
   int i;
   unsigned int dev_id, tmp;
   static const char * const name[] = { "pwm0", "pwm1", "pwm2", "pwm3" };

   dev_info(&client->dev, "cy8c9520a_probe() function is called\n");
```

```
if (!i2c_check_functionality(client->adapter,
                             I2C_FUNC_SMBUS_I2C_BLOCK |
                             I2C_FUNC_SMBUS_BYTE_DATA)) {
       dev_err(&client->dev, "SMBUS Byte/Block unsupported\n");
       return -EIO;
}

/* allocate global private structure for a new device */
cygpio = devm_kzalloc(&client->dev, sizeof(*cygpio), GFP_KERNEL);
if (!cygpio) {
       dev_err(&client->dev, "failed to alloc memory\n");
       return -ENOMEM;
}

cygpio->client = client;

mutex_init(&cygpio->lock);

/* Whoami */
dev_id = i2c_smbus_read_byte_data(client, REG_DEVID_STAT);
if (dev_id < 0) {
       dev_err(&client->dev, "can't read device ID\n");
       ret = dev_id;
       goto err;
}
dev_info(&client->dev, "dev_id=0x%x\n", dev_id & 0xff);

/* parse the DT to get the pwm-pin mapping */
for (i = 0; i < NPWM; i++) {
       ret = device_property_read_u32(&client->dev, name[i], &tmp);
       if (!ret)
              cygpio->pwm_number[i] = tmp;
       else
              goto err;
};

/* Initial setup for the cy8c9520a */
ret = cy8c9520a_setup(cygpio);
if (ret < 0) {
       goto err;
}

dev_info(&client->dev, "the initial setup for the cy8c9520a is done\n");

/* Initialize the cy8c9520a gpio controller */
ret = cy8c9520a_gpio_init(cygpio);
if (ret) {
```

```
        goto err;
}

dev_info(&client->dev, "the setup for the cy8c9520a gpio controller done\n");

/* Interrupt setup for the cy8c9520a */
ret = cy8c9520a_irq_setup(cygpio);
if (ret) {
        goto err;
}

dev_info(&client->dev, "the interrupt setup for the cy8c9520a is done\n");

/* Setup of the pwm_chip controller */
cygpio->pwm_chip.dev = &client->dev;
cygpio->pwm_chip.ops = &cy8c9520a_pwm_ops;
cygpio->pwm_chip.base = PWM_BASE_ID;
cygpio->pwm_chip.npwm = NPWM;

ret = pwmchip_add(&cygpio->pwm_chip);
if (ret) {
        dev_err(&client->dev, "pwmchip_add failed %d\n", ret);
        goto err;
}

dev_info(&client->dev,
         "the setup for the cy8c9520a pwm_chip controller is done\n");

/* Setup of the pinctrl descriptor */
cygpio->pinctrl_desc.name = "cy8c9520a-pinctrl";
cygpio->pinctrl_desc.pctlops = &cygpio_pinctrl_ops;
cygpio->pinctrl_desc.confops = &cygpio_pinconf_ops;
cygpio->pinctrl_desc.npins = cygpio->gpio_chip.ngpio;

cygpio->pinctrl_desc.pins = cy8c9520a_pins;
cygpio->pinctrl_desc.owner = THIS_MODULE;

cygpio->pctldev = devm_pinctrl_register(&client->dev,
                                        &cygpio->pinctrl_desc,
                                        cygpio);
if (IS_ERR(cygpio->pctldev)) {
        ret = PTR_ERR(cygpio->pctldev);
        goto err;
}

dev_info(&client->dev,
         "the setup for the cy8c9520a pinctl descriptor is done\n");
```

```
    /* link the I2C device with the cygpio device */
    i2c_set_clientdata(client, cygpio);

err:
    mutex_destroy(&cygpio->lock);

    return ret;
}

static int cy8c9520a_remove(struct i2c_client *client)
{
    struct cy8c9520a *cygpio = i2c_get_clientdata(client);
    dev_info(&client->dev, "cy8c9520a_remove() function is called\n");
    return pwmchip_remove(&cygpio->pwm_chip);
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "cy8c9520a"},
    {},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static const struct i2c_device_id cy8c9520a_id[] = {
    {DRV_NAME, 0},
    {}
};
MODULE_DEVICE_TABLE(i2c, cy8c9520a_id);

static struct i2c_driver cy8c9520a_driver = {
    .driver = {
              .name = DRV_NAME,
              .of_match_table = my_of_ids,
              .owner = THIS_MODULE,
              },
    .probe = cy8c9520a_probe,
    .remove = cy8c9520a_remove,
    .id_table = cy8c9520a_id,
};
module_i2c_driver(cy8c9520a_driver);

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a driver that controls the \
                   cy8c9520a I2C GPIO expander");
```

LAB 7.5 driver demonstration

Download the linux_5.4_rpi4_drivers.zip file from the github of the book and unzip it in the home folder of your Linux host:

```
~/linux_5.4_rpi4_drivers$ cd linux_5.4_CY8C9520A_pwm_pinctrl
```

Compile and deploy the drivers and the application to the **Raspberry Pi 4 Model B** board:

```
~/linux_5.4_rpi4_drivers/linux_5.4_CY8C9520A_pwm_pinctrl$ make
```

```
~/linux_5.4_rpi4_drivers/linux_5.4_CY8C9520A_pwm_pinctrl$ make deploy
```

```
~/linux_5.4_rpi4_drivers/linux_5.4_CY8C9520A_pwm_pinctrl/linux_5.4_gpio_int_driver
$ make
```

```
~/linux_5.4_rpi4_drivers/linux_5.4_CY8C9520A_pwm_pinctrl/linux_5.4_gpio_int_driver
$ make deploy
```

```
~/linux_5.4_rpi4_drivers/linux_5.4_CY8C9520A_pwm_pinctrl/app$ scp gpio_int.c
root@10.0.0.10:/home
```

```
root@raspberrypi:/home# gcc -o gpio_int gpio_int.c
```

Follow the next instructions to test the drivers:

```
root@raspberrypi:/home# insmod CY8C9520A_pwm_pinctrl.ko
[  601.583868] CY8C9520A_pwm_pinctrl: loading out-of-tree module taints kernel.
[  601.594880] cy8c9520a 1-0020: cy8c9520a_probe() function is called
[  601.603814] cy8c9520a 1-0020: dev_id=0x20
[  601.639410] cy8c9520a 1-0020: the cy8c9520a_setup is done
[  601.644896] cy8c9520a 1-0020: the initial setup for the cy8c9520a is done
[  601.655971] cy8c9520a 1-0020: the setup for the cy8c9520a gpio controller done
[  601.663396] cy8c9520a 1-0020: the cy8c9520a_irq_setup function is entered
[  601.674274] cy8c9520a 1-0020: the interrupt state registers are cleared
[  601.691072] cy8c9520a 1-0020: the interrupt mask port registers are set
[  601.698897] cy8c9520a 1-0020: the interrupt setup is done
[  601.704390] cy8c9520a 1-0020: the interrupt setup for the cy8c9520a is done
[  601.711615] cy8c9520a 1-0020: the setup for the cy8c9520a pwm_chip controller
is done
[  601.720072] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[  601.726451] cy8c9520a 1-0020: The pin 0 drive mode is PIN_CONFIG_BIAS_PULL_UP
[  601.739951] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[  601.746315] cy8c9520a 1-0020: The pin 1 drive mode is PIN_CONFIG_BIAS_PULL_UP
[  601.759975] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[  601.766338] cy8c9520a 1-0020: The pin 2 drive mode is PIN_CONFIG_BIAS_PULL_DOWN
[  601.779998] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[  601.786359] cy8c9520a 1-0020: The pin 3 drive mode is PIN_CONFIG_DRIVE_STRENGTH
[  601.799998] cy8c9520a 1-0020: the setup for the cy8c9520a pinctl descriptor is
done
```

Handle GPIO INT in line 0 of P0 using the gpio interrupt driver

```
/* load the gpio interrupt module */
root@raspberrypi:/home# insmod int_rpi4_gpio.ko
[  650.453164] int_gpio_expand int_gpio: my_probe() function is called.
[  650.459793] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[  650.465731] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[  650.472332] int_gpio_expand int_gpio: IRQ_using_platform_get_irq: 61
[  650.478833] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[  650.484769] cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
[  650.490694] cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
[  650.496482] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[  650.503088] cy8c9520a 1-0020: gpio 0 is unmasked
[  650.513162] cy8c9520a 1-0020: the REG_INTR_MASK value is 254
[  650.519402] int_gpio_expand int_gpio: mydev: got minor 59
[  650.524900] int_gpio_expand int_gpio: my_probe() function is exited.

/* Connect pin 0 of P0 to GND, then disconnect it from GND. Two interrupts are
fired */
root@raspberrypi:/home# [  678.446922] the interrupt ISR has been entered
[  678.454239] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  681.202732] the interrupt ISR has been entered
[  681.210073] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
```

Access the PWM driver via the following sysfs path in user space, /sys/class/pwm

```
root@raspberrypi:/home# cd /sys/class/pwm/

/* Each probed PWM controller will be exported as pwmchipN, where N is the base of
the PWM controller */
root@raspberrypi:/sys/class/pwm# ls
pwmchip0

root@raspberrypi:/sys/class/pwm# cd pwmchip0/

/* npwm is the number of PWM channels this controller supports (read-only) */
root@raspberrypi:/sys/class/pwm/pwmchip0# ls
device  export  npwm  power  subsystem  uevent  unexport

/* Exports a PWM channel (pwm1) with sysfs (write-only). (The PWM channels are
numbered using a per-controller index from 0 to npwm-1.) */
root@raspberrypi:/sys/class/pwm/pwmchip0# echo 1 > export
[  779.937939] cy8c9520a 1-0020: cy8c9520a_pwm_request is called

/* You can see that the pwm1 channel has been created. This channel corresponds to
the pin 3 of our device */
root@raspberrypi:/sys/class/pwm/pwmchip0# ls
device  export  npwm  power  pwm1  subsystem  uevent  unexport

/* Set the total period of the PWM signal (read/write). Value is in nanoseconds */
root@raspberrypi:/sys/class/pwm/pwmchip0# echo 100000 > pwm1/period
```

```
[  854.847874] cy8c9520a 1-0020: cy8c9520a_pwm_config is called

/* Set the active time of the PWM signal (read/write). Value is in nanoseconds */
root@raspberrypi:/sys/class/pwm/pwmchip0# echo 50000 > pwm1/duty_cycle
[  887.217838] cy8c9520a 1-0020: cy8c9520a_pwm_config is called

/* Enable the PWM signal (read/write) where 0 = disabled and 1 = enabled */
root@raspberrypi:/sys/class/pwm/pwmchip0# echo 1 > pwm1/enable
[  909.557877] cy8c9520a 1-0020: cy8c9520a_pwm_enable is called
[  909.563648] cy8c9520a 1-0020: cy8c9520a_gpio_direction output is called
[  909.575907] cy8c9520a 1-0020: cy8c9520a_gpio_set_value func with 1 value is
called

/* Connect pin 0 of P0 to pin 3 of P0. You will see how interrupts are being fired
in each level change of the PWM signal */
[  941.468870] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.475972] the interrupt ISR has been entered
[  941.483726] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.490866] the interrupt ISR has been entered
[  941.498134] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.505233] the interrupt ISR has been entered
[  941.512533] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.519680] the interrupt ISR has been entered
[  941.527394] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.534534] the interrupt ISR has been entered
[  941.542266] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.549405] the interrupt ISR has been entered
[  941.557124] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.564258] the interrupt ISR has been entered
[  941.571956] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.579047] the interrupt ISR has been entered
[  941.586349] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.593436] the interrupt ISR has been entered
[  941.600731] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.607823] the interrupt ISR has been entered
[  941.615119] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.622204] the interrupt ISR has been entered
[  941.629496] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.636566] the interrupt ISR has been entered
[  941.643882] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.650976] the interrupt ISR has been entered
[  941.658264] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.665334] the interrupt ISR has been entered
[  941.672649] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[  941.679739] the interrupt ISR has been entered
.....

/* remove the gpio int module */
root@raspberrypi:/home# rmmod int_rpi4_gpio.ko
[ 2403.281031] int_gpio_expand int_gpio: my_remove() function is called.
```

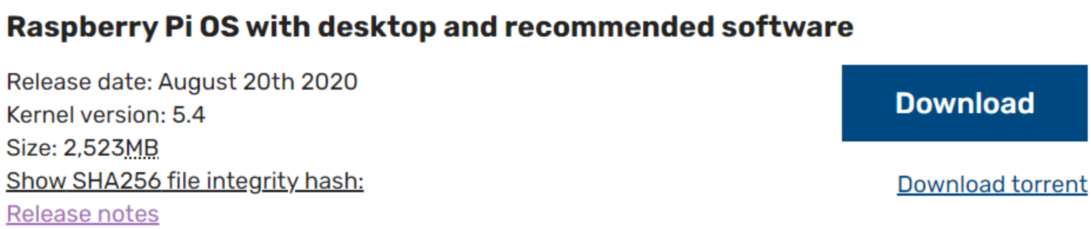
```
[ 2403.287925] int_gpio_expand int_gpio: my_remove() function is exited.
[ 2403.294498] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 2403.300551] cy8c9520a 1-0020: cy8c9520a_irq_mask is called
[ 2403.306133] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[ 2403.312728] cy8c9520a 1-0020: gpio 0 is unmasked
[ 2403.322636] cy8c9520a 1-0020: the REG_INTR_MASK value is 255
[ 2403.328489] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 2403.334432] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called

/* remove the CY8C9520A module */
root@raspberrypi:/home# rmmod CY8C9520A_pwm_pinctrl.ko
[ 2420.271182] cy8c9520a 1-0020: cy8c9520a_remove() function is called
```

LAB 7.6: CY8C9520A device tree overlay

In this lab, you will see how to introduce new hardware support on Raspberry Pi 3 device tree by using the Raspberry Pi specific device tree overlay mechanism. In the previous labs, you described the CY8C9520A device by writing its device tree properties directly in the bcm2710-rpi-3-b.dts file, which describes the hardware on the Raspberry Pi 3 Model B board. The device tree overlay mechanism allows to integrate new hardware, keeping the original device tree source files. Device tree overlays allow you to override specific parts of a device tree and insert dynamically device tree fragments to a live tree and effect change.

When you get to this lab, you will have possibly made many modifications to the original bcm2711-rpi-4-b.dts file and to the kernel source files, so we are going to start from scratch by saving the Raspbian OS image to our uSD and compiling the original kernel and device tree files. I summarize below the main steps to perform these tasks:

You will install on a uSD a **Raspberry Pi OS** image based on **kernel 5.4.y.** Go to https://www.raspberrypi.org/software/operating-systems/ and download Raspberry Pi OS with desktop and recommended software image. Flash the uSD with the downloaded image.



On the host PC, enable UART, SPI and I2C peripherals in the programmed uSD:

```
~$ lsblk
~$ mkdir ~/mnt
```

```
~$ mkdir ~/mnt/fat32
~$ mkdir ~/mnt/ext4
~$ sudo mount /dev/mmcblk0p1 ~/mnt/fat32
~$ 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 nano config.txt
```

```
dtparam=i2c_arm=on
dtparam=spi=on
dtoverlay=spi0-cs
# Enable UART
enable_uart=1
kernel=kernel7l.img
```

On your host PC, create the linux_rpi4 folder, where you are going to download the kernel sources.

```
~$ mkdir linux_rpi4
```

```
~$ cd linux_rpi4/
```

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

```
~/linux_rpi4$ git clone --depth=1 -b rpi-5.4.y
https://github.com/raspberrypi/linux
```

Compile the kernel, modules and device tree files. First, apply the default configuration:

```
~/linux_rpi4/linux$ KERNEL=kernel7l
```

```
~/linux_rpi4/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
bcm2711_defconfig
```

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

```
~/linux_rpi4/linux$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage
modules dtbs
```

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

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

```
~$ sudo mount /dev/mmcblk0p2 ~/mnt/ext4/
~/linux_rpi4/linux$ sudo env PATH=$PATH make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- INSTALL_MOD_PATH=~/mnt/ext4 modules_install
```

Finally, update kernel, device tree files and modules:

```
~/linux_rpi4/linux$ sudo cp ~/mnt/fat32/kernel7l.img ~/mnt/fat32/kernel7l-backup.img
~/linux_rpi4/linux$ sudo cp arch/arm/boot/zImage ~/mnt/fat32/kernel7l.img
~/linux_rpi4/linux$ sudo cp arch/arm/boot/dts/*.dtb ~/mnt/fat32/
~/linux_rpi4/linux$ sudo cp arch/arm/boot/dts/overlays/*.dtb* ~/mnt/fat32/overlays/
~/linux_rpi4/linux$ sudo cp arch/arm/boot/dts/overlays/README ~/mnt/fat32/overlays/
~$ sudo umount ~/mnt/fat32
~$ sudo umount ~/mnt/ext4
```

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

```
~$ sudo minicom – s
```

Set baud rate and other setting as per below:

a. Baud rate 115200
b. Parity none
c. hardware flow control/software flow control none
d. Serial device /dev/ttyUSB0
e. Save setup as dfl

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

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

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

If you modify and compile the kernel or device tree files later, you can copy them to the Raspberry Pi 4 remotely using the secure copy protocol (SCP). You need to connect previously an Ethernet cable between the Raspberry Pi board and your host PC.

```
~/linux_rpi4/linux$ scp arch/arm/boot/zImage root@10.0.0.10:/boot/kernel7l.img
~/linux_rpi4/linux$ scp arch/arm/boot/dts/bcm2711-rpi-4-b.dtb root@10.0.0.10:/boot/
```

Open the bcm2711-rpi-4-b.dts file, which is included in the Linux kernel sources under /arch/arm/boot/dts folder and look for the i2c1 nodes. You will see the following nodes:

```
&i2c1 {
   pinctrl-names = "default";
   pinctrl-0 = <&i2c1_pins>;
   clock-frequency = <100000>;
};

i2c1_pins: i2c1 {
          brcm,pins = <2 3>;
          brcm,function = <BCM2835_FSEL_ALT0>;
          brcm,pull = <BCM2835_PUD_UP>;
};
```

The first node above adds several properties to the i2c1 controller master node. Inside the i2c1 node, you can see the pinctrl-0 property, which points to the i2c1_pins property (second node above), which configures the pins in I2C mode.

The i2c1 controller node is described in the bcm283x.dtsi file, which is included in the Linux kernel sources under /arch/arm/boot/dts folder:

```
i2c1: i2c@7e804000 {
   compatible = "brcm,bcm2835-i2c";
   reg = <0x7e804000 0x1000>;
   interrupts = <2 21>;
   clocks = <&clocks BCM2835_CLOCK_VPU>;
   #address-cells = <1>;
   #size-cells = <0>;
   status = "disabled";
};
```

In the lab 7.5, you added the following sub-nodes and properties (in bold) to the i2c1 and gpio nodes included in the bcm2711-rpi-4-b.dts file

```
&i2c1 {
   pinctrl-names = "default";
   pinctrl-0 = <&i2c1_pins>;
   clock-frequency = <100000>;
   status = "okay";

   cy8c9520a: cy8c9520a@20 {
          compatible = "cy8c9520a";
          reg = <0x20>;
          interrupt-controller;
          #interrupt-cells = <2>;
          gpio-controller;
```

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

            interrupts = <23 1>;
            interrupt-parent = <&gpio>;

            #pwm-cells = <2>;
            pwm0 = <20>; // pwm not supported
            pwm1 = <3>;
            pwm2 = <20>; // pwm not supported
            pwm3 = <2>;

            pinctrl-names = "default";
            pinctrl-0 = <&accel_int_pin &cy8c9520apullups &cy8c9520apulldowns
&cy8c9520adrivestrength>;

            cy8c9520apullups: pinmux1 {
                   pins = "gpio0", "gpio1";
                   bias-pull-up;
            };

            cy8c9520apulldowns: pinmux2 {
                   pins = "gpio2";
                   bias-pull-down;
            };

            /* pwm channel */
            cy8c9520adrivestrength: pinmux3 {
                   pins = "gpio3";
                   drive-strength;
            };
   };

};

&gpio {
   audio_pins: audio_pins {
            brcm,pins = <40 41>;
            brcm,function = <4>;
   };

   key_pin: key_pin {
            brcm,pins = <23>;
            brcm,function = <0>;
            brcm,pull = <1>;
   };

   key_pins: key_pins {
            brcm,pins = <23 24>;
```

```
            brcm,function = <0>;
            brcm,pull = <1 1>;
    };

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

    cs_pins: cs_pins {
            brcm,pins = <23>;
            brcm,function = <1>;  /* Output */
            brcm,pull = <0>;      /* none */
    };

    accel_int_pin: accel_int_pin {
            brcm,pins = <23>;
            brcm,function = <0>;  /* Input */
            brcm,pull = <0>;      /* none */
    };

};
```

As we have commented previously, the purpose of the device tree overlay is to keep our original device tree intact and dynamically add the necessary fragments that describe our new hardware. A DT overlay comprises a number of fragments, each of which targets one node and its subnodes. You will add the code in bold above, by using two fragments. Each fragment will consist of two parts: a target-path property, with the absolute path to the node that the fragment is going to modify, or a target property with the relative path to the node alias (prefixed with an ampersand symbol) that the fragment is going to modify; and the __overlay__ itself, the body of which is added to the target node. In our device tree overlay:

- fragment@0 is adding the accel_int_pin node, with several properties, to the gpio node.
- fragment@1 is adding the cy8c9520a node to the i2c1 node, and is also modifying some properties (like the status property) of the i2c1 node itself.

You will create a cy8c9520a-overlay.dts file, adding the code below and include the file to the arch/arm/boot/dts/overlays folder inside the kernel sources.

```
/ {
    compatible = "brcm,bcm2835";

    fragment@0 {
            target =<&gpio>;
```

```
        __overlay__ {
                accel_int_pin: accel_int_pin {
                        brcm,pins = <23>;
                        brcm,function = <0>;  /* Input */
                        brcm,pull = <0>;      /* none */
                };
        };
};

fragment@1 {
        target = <&i2c1>;
        __overlay__ {
                #address-cells = <1>;
                #size-cells = <0>;
                status = "okay";

                cy8c9520a: cy8c9520a@20 {
                        compatible = "cy8c9520a";
                        reg = <0x20>;
                        interrupt-controller;
                        #interrupt-cells = <2>;
                        gpio-controller;
                        #gpio-cells = <2>;

                        interrupts = <23 1>;
                        interrupt-parent = <&gpio>;

                        #pwm-cells = <2>;
                        pwm0 = <20>; // pwm not supported
                        pwm1 = <3>;
                        pwm2 = <20>; // pwm not supported
                        pwm3 = <2>;

                        pinctrl-names = "default";
                        pinctrl-0 = <&accel_int_pin &cy8c9520apullups
&cy8c9520apulldowns &cy8c9520adrivestrength>;

                        cy8c9520apullups: pinmux1 {
                                pins = "gpio0", "gpio1";
                                bias-pull-up;
                        };

                        cy8c9520apulldowns: pinmux2 {
                                pins = "gpio2";
                                bias-pull-down;
                        };

                        /* pwm channel */
```

```
                        cy8c9520adrivestrength: pinmux3 {
                               pins = "gpio3";
                               drive-strength;
                        };
                 };

          };
   };
};
```

This overlay will get compiled into a .dtbo file. To be compiled, the overlay needs to be referenced in the arch/arm/boot/dts/overlays/Makefile file in the kernel sources .

```
[...]
upstream.dtbo \
upstream-pi4.dtbo \
vc4-fkms-v3d.dtbo \
vc4-kms-kippah-7inch.dtbo \
vc4-kms-v3d.dtbo \
vc4-kms-v3d-pi4.dtbo \
vga666.dtbo \
w1-gpio.dtbo \
w1-gpio-pullup.dtbo \
w5500.dtbo \
wittypi.dtbo \
cy8c9520a.dtbo
```

With this overlay in place, you need to enable it in the config.txt file, as well as the I2C1 overlay with a correct pin-muxing configuration. You can modify the config.txt file directly in the Raspberry Pi board using **Nano** or **Vim** editors.

```
root@raspberrypi:/home# cd /boot
root@raspberrypi:/boot# nano config.txt
```

```
dtparam=i2c_arm=on
#dtparam=i2s=on
dtparam=spi=on
dtoverlay=spi0-cs
enable_uart=1
kernel=kernel7l.img
dtoverlay=cy8c9520a
dtoverlay=i2c1,pins_2_3
```

You can see below the I2C1 overlay (i2c1-overlay.dts), which is included in the arch/arm/boot/dts/overlays folder:

```
/dts-v1/;
/plugin/;

/{
   compatible = "brcm,bcm2835";

   fragment@0 {
          target = <&i2c1>;
          __overlay__ {
                 status = "okay";
                 pinctrl-names = "default";
                 pinctrl-0 = <&i2c1_pins>;
          };
   };

   fragment@1 {
          target = <&i2c1_pins>;
          pins1: __overlay__ {
                 brcm,pins = <2 3>;
                 brcm,function = <4>; /* alt 0 */
          };
   };

   fragment@2 {
          target = <&i2c1_pins>;
          pins2: __dormant__ {
                 brcm,pins = <44 45>;
                 brcm,function = <6>; /* alt 2 */
          };
   };

   fragment@3 {
          target = <&i2c1>;
          __dormant__ {
                 compatible = "brcm,bcm2708-i2c";
          };
   };

   __overrides__ {
          pins_2_3   = <0>,"=1!2";
          pins_44_45 = <0>,"!1=2";
          combine = <0>, "!3";
   };
};
```

You can see in the I2C1 overlay the use of device tree parameters. To avoid the need for lots of device tree overlays, and to reduce the need for users of peripherals to modify DTS files, the

Raspberry Pi loader supports a new feature - device tree parameters. Parameters are defined in the DTS by adding an __overrides__ node to the root. You can read about the different types of parameters in the following link of the Raspberry Pi´s documentation:

https://www.raspberrypi.org/documentation/configuration/device-tree.md

The I2C1 overlay will use Overlay/fragment parameters. The DT parameter mechanism has a number of limitations, including the inability to change the name of a node and to write arbitrary values to arbitrary properties when a parameter is used. One way to overcome some of these limitations is to conditionally include or exclude certain fragments. A fragment can be excluded from the final merge process (disabled) by renaming the __overlay__ node to __dormant__. The parameter declaration syntax has been extended to allow the otherwise illegal zero target phandle to indicate that the following string contains operations at fragment or overlay scope. So far, four operations have been implemented:

```
+<n>    // Enable fragment <n>
-<n>    // Disable fragment <n>
=<n>    // Enable fragment <n> if the assigned parameter value is true, otherwise
disable it
!<n>    // Enable fragment <n> if the assigned parameter value is false, otherwise
disable it
```

In the I2C1 overlay:

```
__overrides__ {
          pins_2_3   = <0>,"=1!2"; // the pins_2_3 parameter enables fragment 1
and disables fragment 2 if value is true
          pins_44_45 = <0>,"!1=2"; // the pins_44_45 parameter enables fragment 2
and disables fragment 1 if value is true
          combine = <0>, "!3"; // the combine parameter enables fragment 3 if the
assigned parameter value is false, otherwise disable it

};
```

Compile the device tree overlay and copy it to the Raspberry Pi device:

```
~/linux_rpi4/linux$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
~/linux_rpi4/linux$ scp arch/arm/boot/dts/overlays/cy8c9520a.dtbo
root@10.0.0.10:/boot/overlays
```

Reboot the Raspberry Pi board.

Compile the “CY8C9520A_pwm_pinctrl” driver and the application and send them to the Raspberry Pi board:

```
~/linux_5.4_rpi4_drivers/device_tree_overlays/linux_5.4_CY8C9520A_pwm_pinctrl$
make
```

```
~/linux_5.4_rpi4_drivers/device_tree_overlays/linux_5.4_CY8C9520A_pwm_pinctrl$
make deploy
```

```
~/linux_5.4_rpi4_drivers/device_tree_overlays/linux_5.4_CY8C9520A_pwm_pinctrl/app$
scp gpio_int.c root@10.0.0.10:/home
```

```
root@raspberrypi:/home# gcc -o gpio_int gpio_int.c
```

Follow the instructions of the “LAB 7.5 driver demonstration” section to test the driver.

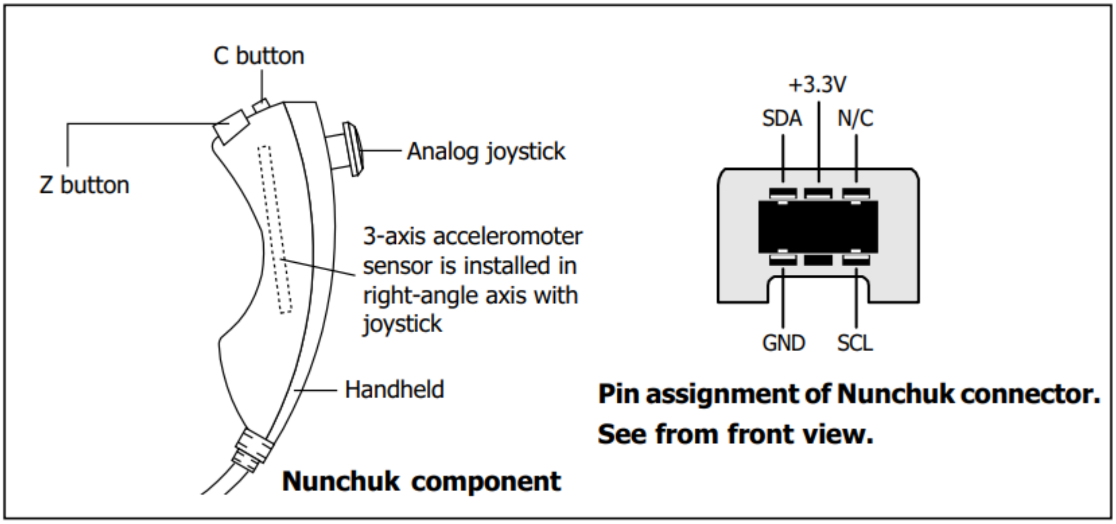
```
root@raspberrypi:/home# insmod CY8C9520A_pwm_pinctrl.ko
[  601.583868] CY8C9520A_pwm_pinctrl: loading out-of-tree module taints kernel.
[  601.594880] cy8c9520a 1-0020: cy8c9520a_probe() function is called
[  601.603814] cy8c9520a 1-0020: dev_id=0x20
[  601.639410] cy8c9520a 1-0020: the cy8c9520a_setup is done
[  601.644896] cy8c9520a 1-0020: the initial setup for the cy8c9520a is done
[  601.655971] cy8c9520a 1-0020: the setup for the cy8c9520a gpio controller done
[  601.663396] cy8c9520a 1-0020: the cy8c9520a_irq_setup function is entered
[  601.674274] cy8c9520a 1-0020: the interrupt state registers are cleared
[  601.691072] cy8c9520a 1-0020: the interrupt mask port registers are set
[  601.698897] cy8c9520a 1-0020: the interrupt setup is done
[  601.704390] cy8c9520a 1-0020: the interrupt setup for the cy8c9520a is done
[  601.711615] cy8c9520a 1-0020: the setup for the cy8c9520a pwm_chip controller
is done
[  601.720072] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[  601.726451] cy8c9520a 1-0020: The pin 0 drive mode is PIN_CONFIG_BIAS_PULL_UP
[  601.739951] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[  601.746315] cy8c9520a 1-0020: The pin 1 drive mode is PIN_CONFIG_BIAS_PULL_UP
[  601.759975] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[  601.766338] cy8c9520a 1-0020: The pin 2 drive mode is PIN_CONFIG_BIAS_PULL_DOWN
[  601.779998] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[  601.786359] cy8c9520a 1-0020: The pin 3 drive mode is PIN_CONFIG_DRIVE_STRENGTH
[  601.799998] cy8c9520a 1-0020: the setup for the cy8c9520a pinctl descriptor is
done

[...]
```

LAB 10.3: “Nunchuk input subsystem” module

This new LAB 10.3 has been added to the labs of Chapter 10 to reinforce the concepts of creating **Input Subsystem** drivers, which were explained during the chapter ten of this book.

A new low cost evaluation board based on the **WII Nunchuk** controller will be used, thus expanding the number of evaluation boards that can be acquired to practice with the theory explained in Chapter 10.

The WII Nunchuk is a controller for the Wii game console. It includes a 3-axis accelerometer, a X-Y Joystick and two buttons. All these features are available through I2C communication. The next image shows the components of WII Nunchuk and pin assignment of the connector.



For the development of the lab, you will use the MOD-Wii-UEXT-NUNCHUCK from Olimex (https://www.olimex.com/Products/Modules/Sensors/MOD-WII/MOD-Wii-UEXT-NUNCHUCK/open-source-hardware).

The Wii Nunchuk uses a proprietary connector with six pins, which exposes I2C signals, +3V and Gnd. The MOD-Wii-UEXT-NUNCHUCK includes an UEXT adapter, which offers a kind of universal connector supporting three serial communication interfaces: I2C, SPI and RS232. In the MOD-Wii-UEXT-NUNCHUCK only I2C signals are connected to the Nunchuk device.

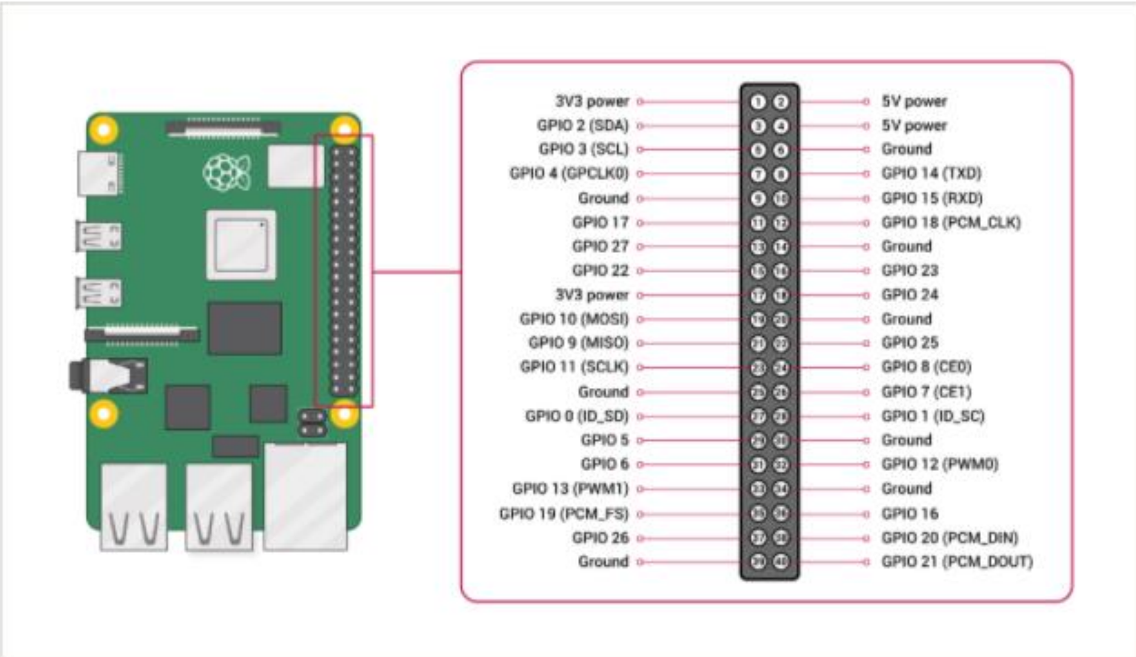
The hardware platforms used in this lab are the Raspberry Pi 4 Model B board and the WII NUNCHUCK controller with UEXT connector, which was previously described.

The same software configuration from the previous LAB 7.6 will be used for this lab.

This LAB 10.3 has been inspired by Bootlin´s Embedded Linux kernel and driver development training materials (https://bootlin.com/training/kernel/).

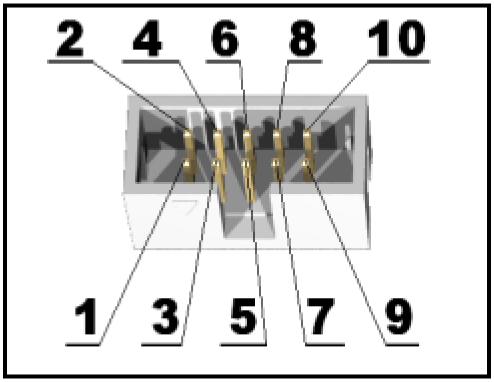
LAB 10.3 hardware description

You will use the I2C pins of the Raspberry Pi 4 Model B 40-pin GPIO header to connect to the Nunchuk´s UEXT connector. In the following imaging, you can see the Raspberry Pi 4 Model B connector:



And this is the Nunchuk´s UEXT connector:

Pin #	Signal Name
1	3.3V
2	GND
3	TXD
4	RXD
5	SCL
6	SDA
7	MISO
8	MOSI
9	SCK
10	SSEL



Connect the Raspberry Pi 4 Model B I2C pins to the Nunchuck I2C ones:

- Connect Raspberry Pi 4 Model B **SCL** to UEXT **SCL** (Pin 5)
- Connect Raspberry Pi 4 Model B **SDA** to UEXT **SDA** (Pin 6)

Also connect the next power pins between the two boards:

- Connect Raspberry Pi 4 Model B **3.3V** to UEXT **3.3V** (Pin 1)
- Connect Raspberry Pi 4 Model B **GND** to UEXT **GND** (Pin 2)

The hardware setup between the two boards is already done!!

LAB 10.3 device tree description

Open the bcm2711-rpi-4-b.dts DT file under the kernel sources and find the i2c1 controller master node. Inside the i2c1 node, you can see the pinctrl-0 property which configures the pins in I2C mode. The i2c1_pins are already defined in the bcm2711-rpi-4-b.dts file inside the gpio node property.

The i2c1 controller is enabled by writing "okay" to the status property. You will set to 100KHz the clock-frequency property. The Nunchuck device communicates with MPU using an I2C bus interface with a maximum frequency of 100kHz.

Now, you will add to the i2c1 controller node the nunchuk node. There must be a DT device node´s compatible property identical to the compatible string stored in one of the driver´s of_device_id structures. The reg property includes the I2C address of the device.

You can see below the device-tree configuration for our nunchuk device:

```
&i2c1 {
   pinctrl-names = "default";
   pinctrl-0 = <&i2c1_pins>;
   clock-frequency = <100000>;
   status = "okay";

   nunchuk: nunchuk@52 {
          compatible = "nunchuk";
          reg = <0x52>;
   };
};
```

LAB 10.3 Nunchuk controller driver description

The main code sections of the driver will be described using two different categories: I2C driver setup, and Input framework driver setup.

I2C driver setup

These are the main code sections:

1. Include the required header files:

   ```
   #include <linux/i2c.h>
   ```

2. Create a i2c_driver structure:

   ```
   static struct i2c_driver nunchuk_driver = {
          .driver = {
                 .name = "nunchuk",
                 .owner = THIS_MODULE,
                 .of_match_table = nunchuk_of_match,
          },
          .probe = nunchuk_probe,
          .remove = nunchuk_remove,
          .id_table = nunchuk_id,
   };
   ```

3. Register to the I2C bus as a driver:

   ```
   module_i2c_driver(nunchuk_driver);
   ```

4. Add "nunchuk" to the list of devices supported by the driver. The compatible variable matches with the compatible property of the nunchuk DT node:

```
static const struct of_device_id nunchuk_of_match[] = {
	{ .compatible = "nunchuk"},
	{}
};
MODULE_DEVICE_TABLE(of, nunchuk_of_match);
```

5. Define an array of i2c_device_id structures:

```
static const struct i2c_device_id nunchuk_id[] = {
	{"nunchuk", 0},
	{}
};
MODULE_DEVICE_TABLE(i2c, nunchuk_id);
```

6. You will use i2c_master_send() and i2c_master_recv() functions to establish a plain I2C communication with the Nunchuck controller. These routines read and write some bytes from/to a client device. The first parameter is the client pointer, which contains the I2C address of the Nunchuk device (the I2C slave address of the Nunchuk is 0x52). The second parameter contains the buffer 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.

   To communicate with the Nunchuk, you must send a handshake signal. In the probe() function, you will send 2 bytes "0xf0, 0x55" to initialize the first register and "0xFB, 0x00" to initialize the second register of the Nunchuk. Then, send one byte "0x00" each time you request data from the Nunchuck. You can see below the nunchuk_read_registers() function, which will read the data from the Nunchuk in 6 byte chunks.

```
static int nunchuk_read_registers(struct i2c_client *client, u8 *buf,
				   int buf_size)
{
	int status;

	mdelay(10);

	buf[0] = 0x00;
	status = i2c_master_send(client, buf, 1);
	if (status >= 0 && status != 1)
		return -EIO;
	if (status < 0)
		return status;

	mdelay(10);

	status = i2c_master_recv(client, buf, buf_size);
	if (status >= 0 && status != buf_size)
```

```
                return -EIO;
        if (status < 0)
                return status;

        return 0;
}
```

The following image shows the six bytes data stream coming from the Nunchuck controller. First 2 bytes are X and Y axis data of the Joystick. Next 3 bytes are X, Y and Z axis data of the accelerometer sensor and the last byte includes the 2 lower bits of the accelerometer axes and the c-button and z-button status.

Data byte receive								Address
Joystick X								0x00
Joystick Y								0x01
Accelerometer X (bit 9 to bit 2 for 10-bit resolution)								0x02
Accelerometer Y (bit 9 to bit 2 for 10-bit resolution)								0x03
Accelerometer Z (bit 9 to bit 2 for 10-bit resolution)								0x04
Accel. Z bit 1	Accel. Z bit 0	Accel. Y bit 1	Accel. Y bit 0	Accel. X bit 1	Accel. X bit 0	C-button	Z-button	0x05

Byte 0x00 : X-axis data of the joystick

Byte 0x01 : Y-axis data of the joystick

Byte 0x02 : X-axis data of the accellerometer sensor

Byte 0x03 : Y-axis data of the accellerometer sensor

Byte 0x04 : Z-axis data of the accellerometer sensor

Byte 0x05 : bit 0 as Z button status - 0 = pressed and 1 = release

bit 1 as C button status - 0 = pressed and 1 = release

bit 2 and 3 as 2 lower bit of X-axis data of the accellerometer sensor

bit 4 and 5 as 2 lower bit of Y-axis data of the accellerometer sensor

bit 6 and 7 as 2 lower bit of Z-axis data of the accellerometer sensor

Input Framework driver setup

These are the main code sections:

1. Include the required header files:
```
#include <linux/input.h>
#include linux/input-polldev.h /* struct input_polled_dev,
input_allocate_polled_device(), input_register_polled_device() */
```

2. The device model needs to keep pointers between physical devices (devices as handled by the physical bus, I2C in this case) and logical devices (devices handled by subsystems, like the Input subsystem in this case). This need is typically implemented by creating a private data structure to manage the device and implement such pointers between the physical and logical worlds. As you have seen in other labs throughout this book, this private structure allows the driver to manage multiple devices using the same driver. Add the private structure definition to your driver code:
```
struct nunchuk_dev {
	struct input_polled_dev *polled_input;
	struct i2c_client *client;
};
```

3. In the nunchuk_probe() function, declare an instance of the previous structure and allocate it:
```
struct nunchuk_dev *nunchuk;

nunchuk = devm_kzalloc(&client->dev,
                       sizeof(*nunchuk),
                       GFP_KERNEL);
```

4. To be able to access your private data structure in other functions of the driver, you need to attach it to the i2c_client structure using the i2c_set_clientdata() function. This function stores nunchuk in client->dev->driver_data. You can retrieve the nunchuk pointer to the private structure using the function i2c_get_clientdata(client).
```
i2c_set_clientdata(client, nunchuk); /* write it in the probe() function */
nunchuk = i2c_get_clientdata(client); /* write it in the remove() function */
```

5. Allocate the input_polled_dev structure in probe():
```
polled_device = devm_input_allocate_polled_device(&client->dev);
```

6. Initialize the polled input device. Keep pointers between physical devices (devices as handled by the physical bus, I2C in this case) and logical devices:
```
nunchuk->client = client; /* store pointer to the I2C device in the global
structure, needed for exchanging data with the nunchuk device */
```

```
polled_device->private = nunchuk; /* struct polled_device can store the
driver-specific data in void *private. Place the pointer to the private
structure here; in this way you will be able to recover the nunchuk pointer
later (for example in the nunchuk_poll() function) */
polled_device->poll_interval = 50; /* Callback interval */
polled_device->poll = nunchuk_poll; /* Callback, that will be called every 50
ms interval */
polled_device->input->dev.parent = &client->dev; /* keep pointers between
physical devices and logical devices */
polled_device->input->name = "WII Nunchuk"; /* input sub-device parameters
that will appear in log on registering the device */
polled_device->input->id.bustype = BUS_I2C; /* input sub-device parameters */
```

7. Set event types and event codes for the Nunchuk device:

```
/* Set EV_KEY type events and from those BTN_C and BTN_Z event codes */
set_bit(EV_KEY, input->evbit);
set_bit(BTN_C, input->keybit); /* buttons */
set_bit(BTN_Z, input->keybit);

/*
 * Set EV_ABS type events and from those
 * ABS_X, ABS_Y, ABS_RX, ABS_RY and ABS_RZ event codes
 */
set_bit(EV_ABS, input->evbit);
set_bit(ABS_X, input->absbit); /* joystick */
set_bit(ABS_Y, input->absbit);
set_bit(ABS_RX, input->absbit); /* accelerometer */
set_bit(ABS_RY, input->absbit);
set_bit(ABS_RZ, input->absbit);

/*
 * fill additional fields in the input_dev struct for
 * each absolute axis nunchuk has
 */
input_set_abs_params(input, ABS_X, 0x00, 0xff, 0, 0);
input_set_abs_params(input, ABS_Y, 0x00, 0xff, 0, 0);

input_set_abs_params(input, ABS_RX, 0x00, 0x3ff, 0, 0);
input_set_abs_params(input, ABS_RY, 0x00, 0x3ff, 0, 0);
input_set_abs_params(input, ABS_RZ, 0x00, 0x3ff, 0, 0);
```

8. Register in probe() and unregister in remove() the polled_input device to the input core. Once registered the device is global for the rest of the driver functions until it is unregistered. After this call, the device is ready to accept requests from the user space applications.

```
input_register_polled_device(nunchuk->polled_input);
input_unregister_polled_device(nunchuk->polled_input);
```

9. Write the nunchuk_poll() function. This function will be called every 50 ms. Inside nunchuk_poll(), you will call nunchuk_read_registers(), which read data from the Nunchuk device. The first parameter of the nunchuk_read_registers() function is a pointer to the i2c_client structure. This pointer will allow you to get the Nunckuk I2C address (0x52). The client pointer will be retrieved from client->address using the following lines of code:

```
nunchuk = polled_input->private;
client = nunchuk->client;
```

The first thing you should do is place a 10 ms delay at the beginning of the nunchuk_read_registers() function using the mdelay() function. This delay will separate the following I2C action from any previous I2C action. If you look through the Nunchuk documentation, you will see that each time you want to read from the Nunchuk device, you must first send the byte 0x00. The Nunchuk will then return 6 bytes of data. Therefore, the next thing your nunchuk_read_registers() function should do is send the 0x00 byte using the i2c_master_send() function. This action should be immediately followed by a 10 ms delay using the mdelay() function. Finally, nunchuk_read_registers() should read six bytes of data from the Nunchuk device and store them in buf using the i2c_master_recv() function.

You are going to store the buf[0] and buf[1] joystick values in the joy_x and joy_y variables. You will also get the C button and Z button status from the buf[5] variable and store it in the c_button and z_button variables. The accelerometer data for its three axes will be retrieved from buf[2] and buf[5] and stored in the accel_x, accel_y and accel_z variables.

Finally, you will report the events to the input system. The input_sync() function will tell those who receive the events that a complete report has been sent.

```
static int nunchuk_read_registers(struct i2c_client *client, u8 *buf,
                                  int buf_size)

{

       mdelay(10);

       buf[0] = 0x00;
       i2c_master_send(client, buf, 1);

       mdelay(10);

       i2c_master_recv(client, buf, buf_size);

       return 0;
}
```

```
/*
 * poll handler function read the harware,
 * queue events to be reported (input_report_*)
 * and flush the queued events (input_sync)
 */
static void nunchuk_poll(struct input_polled_dev *polled_input)
{
	u8 buf[6];
	int joy_x, joy_y, z_button, c_button, accel_x, accel_y, accel_z;
	struct i2c_client *client;
	struct nunchuk_dev *nunchuk;

	/*
	 * Recover the global nunchuk structure and from it the client address
	 * to stablish an I2C transaction with the nunchuck device
	 */
	nunchuk = polled_input->private;
	client = nunchuk->client;

	/* Read the registers of the nunchuk device */
	nunchuk_read_registers(client, buf, ARRAY_SIZE(buf))

	joy_x = buf[0];
	joy_y = buf[1];

	/* Bit 0 indicates if Z button is pressed */
	z_button = (buf[5] & BIT(0)? 0 : 1);
	/* Bit 1 indicates if C button is pressed */
	c_button = (buf[5] & BIT(1)? 0 : 1);

	accel_x = (buf[2] << 2) | ((buf[5] >> 2) & 0x3);
	accel_y = (buf[3] << 2) | ((buf[5] >> 4) & 0x3);
	accel_z = (buf[4] << 2) | ((buf[5] >> 6) & 0x3);

	/* Report events to the input system */
	input_report_abs(polled_input->input, ABS_X, joy_x);
	input_report_abs(polled_input->input, ABS_Y, joy_y);

	input_event(polled_input->input, EV_KEY, BTN_Z, z_button);
	input_event(polled_input->input, EV_KEY, BTN_C, c_button);

	input_report_abs(polled_input->input, ABS_RX, accel_x);
	input_report_abs(polled_input ->input, ABS_RY, accel_y);
	input_report_abs(polled_input->input, ABS_RZ, accel_z);

	/*
	 * Tell those who receive the events
```

```
		 * that a complete report has been sent
		 */
		input_sync(polled_input->input);
	}
```

Listing 10-3: nunchuk.c

```
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/input.h>
#include <linux/input-polldev.h>

/* create private structure */
struct nunchuk_dev {
	struct input_polled_dev *polled_input;
	struct i2c_client *client;
};

static int nunchuk_read_registers(struct i2c_client *client, u8 *buf,
				  int buf_size)
{
	int status;

	mdelay(10);

	buf[0] = 0x00;
	status = i2c_master_send(client, buf, 1);
	if (status >= 0 && status != 1)
		return -EIO;
	if (status < 0)
		return status;

	mdelay(10);

	status = i2c_master_recv(client, buf, buf_size);
	if (status >= 0 && status != buf_size)
		return -EIO;
	if (status < 0)
		return status;

	return 0;
}

/*
 * poll handler function read the harware,
 * queue events to be reported (input_report_*)
 * and flush the queued events (input_sync)
```

```
 */
static void nunchuk_poll(struct input_polled_dev *polled_input)
{
   u8 buf[6];
   int joy_x, joy_y, z_button, c_button, accel_x, accel_y, accel_z;
   struct i2c_client *client;
   struct nunchuk_dev *nunchuk;

   /*
    * Recover the global nunchuk structure and from it the client address
    * to stablish an I2C transaction with the nunchuck device
    */
   nunchuk = polled_input->private;
   client = nunchuk->client;

   /* Read the registers of the nunchuk device */
   if (nunchuk_read_registers(client, buf, ARRAY_SIZE(buf)) < 0)
   {
          dev_info(&client->dev, "Error reading the nunchuk registers.\n");
          return;
   }

   joy_x = buf[0];
   joy_y = buf[1];

   /* Bit 0 indicates if Z button is pressed */
    z_button = (buf[5] & BIT(0)? 0 : 1);
    /* Bit 1 indicates if C button is pressed */
    c_button = (buf[5] & BIT(1)? 0 : 1);

   accel_x = (buf[2] << 2) | ((buf[5] >> 2) & 0x3);
   accel_y = (buf[3] << 2) | ((buf[5] >> 4) & 0x3);
   accel_z = (buf[4] << 2) | ((buf[5] >> 6) & 0x3);

   /* Report events to the input system */
   input_report_abs(polled_input->input, ABS_X, joy_x);
   input_report_abs(polled_input->input, ABS_Y, joy_y);

   input_event(polled_input->input, EV_KEY, BTN_Z, z_button);
   input_event(polled_input->input, EV_KEY, BTN_C, c_button);

   input_report_abs(polled_input->input, ABS_RX, accel_x);
   input_report_abs(polled_input ->input, ABS_RY, accel_y);
   input_report_abs(polled_input->input, ABS_RZ, accel_z);

   /*
    * Tell those who receive the events
    * that a complete report has been sent
```

```
     */
    input_sync(polled_input->input);
}

static int nunchuk_probe(struct i2c_client *client,
                         const struct i2c_device_id *id)
{
    int ret;
    u8 buf[2];
    struct device *dev = &client->dev;

    /* declare a pointer to the private structure */
    struct nunchuk_dev *nunchuk;

    /* declare pointers to input_dev and input_polled_dev structures */
    struct input_dev *input;
    /* For devices that can to be polled on a timer basis */
    struct input_polled_dev *polled_device;

    dev_info(&client->dev, "nunchuck_probe() function is called.\n");

    /* allocate private structure for new device */
    nunchuk = devm_kzalloc(&client->dev, sizeof(*nunchuk), GFP_KERNEL);
    if (nunchuk == NULL)
           return -ENOMEM;

    /* Associate client->dev with nunchuk private structure */
    i2c_set_clientdata(client, nunchuk);

    /* Allocate the struct input_polled_dev */
    polled_device = devm_input_allocate_polled_device(&client->dev);
    if (!polled_device) {
           dev_err(dev, "unable to allocate input device\n");
           return -ENOMEM;
    }

    /* Store the client device in the global structure */
    nunchuk->client = client;

    /* Initialize the polled input device */
    /* To recover nunchuk in the poll() function */
    polled_device->private = nunchuk;

    /* Fill in the poll interval */
    polled_device->poll_interval = 50;

    /* Fill in the poll handler */
    polled_device->poll = nunchuk_poll;
```

```
polled_device->input->dev.parent = &client->dev;

polled_device->input->name = "WII Nunchuk";
polled_device->input->id.bustype = BUS_I2C;

/*
 * Store the polled device in the global structure
 * to recover it in the remove() function
 */
nunchuk->polled_input = polled_device;

input = polled_device->input;

/* Set EV_KEY type events and from those BTN_C and BTN_Z event codes */
set_bit(EV_KEY, input->evbit);
set_bit(BTN_C, input->keybit); /* buttons */
set_bit(BTN_Z, input->keybit);

/*
 * Set EV_ABS type events and from those
 * ABS_X, ABS_Y, ABS_RX, ABS_RY and ABS_RZ event codes
 */
set_bit(EV_ABS, input->evbit);
set_bit(ABS_X, input->absbit); /* joystick */
set_bit(ABS_Y, input->absbit);
set_bit(ABS_RX, input->absbit); /* accelerometer */
set_bit(ABS_RY, input->absbit);
set_bit(ABS_RZ, input->absbit);

/*
 * fill additional fields in the input_dev struct for
 * each absolute axis nunchuk has
 */
input_set_abs_params(input, ABS_X, 0x00, 0xff, 0, 0);
input_set_abs_params(input, ABS_Y, 0x00, 0xff, 0, 0);

input_set_abs_params(input, ABS_RX, 0x00, 0x3ff, 0, 0);
input_set_abs_params(input, ABS_RY, 0x00, 0x3ff, 0, 0);
input_set_abs_params(input, ABS_RZ, 0x00, 0x3ff, 0, 0);

/* Nunchuk handshake */
buf[0] = 0xf0;
buf[1] = 0x55;
ret = i2c_master_send(client, buf, 2);
if (ret >= 0 && ret != 2)
        return -EIO;
if (ret < 0)
```

```
		return ret;

	udelay(1);

	buf[0] = 0xfb;
	buf[1] = 0x00;
	ret = i2c_master_send(client, buf, 1);
	if (ret >= 0 && ret != 1)
		return -EIO;
	if (ret < 0)
		return ret;

	/* Finally, register the input device */
	ret = input_register_polled_device(nunchuk->polled_input);
	if (ret < 0)
		return ret;

	return 0;
}

static int nunchuk_remove(struct i2c_client *client)
{
	struct nunchuk_dev *nunchuk;
	nunchuk = i2c_get_clientdata(client);
	input_unregister_polled_device(nunchuk->polled_input);
	dev_info(&client->dev, "nunchuk_remove()\n");

	return 0;
}

/* Add entries to device tree */
static const struct of_device_id nunchuk_of_match[] = {
	{ .compatible = "nunchuk"},
	{}
};
MODULE_DEVICE_TABLE(of, nunchuk_of_match);

static const struct i2c_device_id nunchuk_id[] = {
	{ "nunchuk", 0 },
	{}
};
MODULE_DEVICE_TABLE(i2c, nunchuk_id);

/* create struct i2c_driver */
static struct i2c_driver nunchuk_driver = {
	.driver = {
		.name = "nunchuk",
		.owner = THIS_MODULE,
```

```
            .of_match_table = nunchuk_of_match,
    },
    .probe = nunchuk_probe,
    .remove = nunchuk_remove,
    .id_table = nunchuk_id,
};

/* Register to i2c bus as a driver */
module_i2c_driver(nunchuk_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a Nunchuk Wii I2C driver");
```

LAB 10.3 driver demonstration

Download the linux_5.4_rpi4_drivers.zip file from the github of the book and unzip it in the home folder of your Linux host:

```
~/linux_5.4_rpi4_drivers$ cd nunchuk_drivers
```

Compile and deploy the drivers to the **Raspberry Pi 4 Model B** board:

```
~/linux_5.4_rpi4_drivers/nunchuk_drivers$ make
```

```
~/linux_5.4_rpi4_drivers/nunchuk_drivers$ make deploy
```

Compile now your device tree and copy it to the Raspberry Pi device:

```
~/linux_rpi4/linux$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage
dtbs
```

```
~/linux_rpi4/linux$ scp arch/arm/boot/zImage root@10.0.0.10:/boot/kernel7l.img
```

```
~/linux_rpi4/linux$ scp arch/arm/boot/dts/bcm2711-rpi-4-b.dtb
root@10.0.0.10:/boot/
```

Reboot the Raspberry Pi Board.

Follow the next instructions to test the driver:

```
/* load the nunchuk module */
root@raspberrypi:/home/pi# insmod nunchuk.ko
[  107.644441] nunchuk: loading out-of-tree module taints kernel.
[  107.651479] nunchuk 1-0052: nunchuck_probe() function is called.
[  107.660810] input: WII Nunchuk as /devices/platform/soc/3f804000.i2c/i2c-1/1-
0052/input/input0
root@raspberrypi:/home/pi#

/* launch the evtest application and play with the nunchuk device */
root@raspberrypi:/home/pi# evtest
No device specified, trying to scan all of /dev/input/event*
```

```
Available devices:
/dev/input/event0:      WII Nunchuk
Select the device event number [0-0]: 0
Input driver version is 1.0.1
Input device ID: bus 0x18 vendor 0x0 product 0x0 version 0x0
Input device name: "WII Nunchuk"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 306 (BTN_C)
    Event code 309 (BTN_Z)
  Event type 3 (EV_ABS)
    Event code 0 (ABS_X)
      Value    126
      Min        0
      Max      255
    Event code 1 (ABS_Y)
      Value    130
      Min        0
      Max      255
    Event code 3 (ABS_RX)
      Value    669
      Min        0
      Max     1023
    Event code 4 (ABS_RY)
      Value    513
      Min        0
      Max     1023
    Event code 5 (ABS_RZ)
      Value    634
      Min        0
      Max     1023
Properties:
Testing ... (interrupt to exit)
Event: time 1608594499.723581, type 3 (EV_ABS), code 3 (ABS_RX), value 669
Event: time 1608594499.723581, type 3 (EV_ABS), code 4 (ABS_RY), value 513
Event: time 1608594499.723581, type 3 (EV_ABS), code 5 (ABS_RZ), value 634
Event: time 1608594499.723581, -------------- SYN_REPORT ------------
Event: time 1608594499.803433, type 3 (EV_ABS), code 3 (ABS_RX), value 580
Event: time 1608594499.803433, type 3 (EV_ABS), code 4 (ABS_RY), value 482
Event: time 1608594499.803433, type 3 (EV_ABS), code 5 (ABS_RZ), value 665
Event: time 1608594499.803433, -------------- SYN_REPORT ------------
Event: time 1608594499.883281, type 3 (EV_ABS), code 3 (ABS_RX), value 490
Event: time 1608594499.883281, type 3 (EV_ABS), code 4 (ABS_RY), value 451
Event: time 1608594499.883281, type 3 (EV_ABS), code 5 (ABS_RZ), value 698
Event: time 1608594499.883281, -------------- SYN_REPORT ------------
Event: time 1608594499.963330, type 3 (EV_ABS), code 3 (ABS_RX), value 401
Event: time 1608594499.963330, type 3 (EV_ABS), code 4 (ABS_RY), value 421
```

```
Event: time 1608594499.963330, type 3 (EV_ABS), code 5 (ABS_RZ), value 730
Event: time 1608594499.963330, -------------- SYN_REPORT ------------
Event: time 1608594500.043247, type 3 (EV_ABS), code 3 (ABS_RX), value 387
Event: time 1608594500.043247, type 3 (EV_ABS), code 4 (ABS_RY), value 426
Event: time 1608594500.043247, type 3 (EV_ABS), code 5 (ABS_RZ), value 726
Event: time 1608594500.043247, -------------- SYN_REPORT ------------
Event: time 1608594500.123308, type 1 (EV_KEY), code 309 (BTN_Z), value 1
Event: time 1608594500.123308, type 3 (EV_ABS), code 3 (ABS_RX), value 388
Event: time 1608594500.123308, type 3 (EV_ABS), code 4 (ABS_RY), value 430
Event: time 1608594500.123308, type 3 (EV_ABS), code 5 (ABS_RZ), value 728
Event: time 1608594500.123308, -------------- SYN_REPORT ------------
Event: time 1608594500.203264, type 3 (EV_ABS), code 3 (ABS_RX), value 387
Event: time 1608594500.203264, type 3 (EV_ABS), code 4 (ABS_RY), value 429
Event: time 1608594500.203264, type 3 (EV_ABS), code 5 (ABS_RZ), value 730
Event: time 1608594500.203264, -------------- SYN_REPORT ------------
Event: time 1608594500.283249, type 3 (EV_ABS), code 3 (ABS_RX), value 389
Event: time 1608594500.283249, type 3 (EV_ABS), code 4 (ABS_RY), value 434
Event: time 1608594500.283249, type 3 (EV_ABS), code 5 (ABS_RZ), value 733
Event: time 1608594500.283249, -------------- SYN_REPORT ------------
Event: time 1608594500.363260, type 1 (EV_KEY), code 309 (BTN_Z), value 0
Event: time 1608594500.363260, type 3 (EV_ABS), code 3 (ABS_RX), value 396
Event: time 1608594500.363260, type 3 (EV_ABS), code 4 (ABS_RY), value 438
Event: time 1608594500.363260, type 3 (EV_ABS), code 5 (ABS_RZ), value 734
Event: time 1608594500.363260, -------------- SYN_REPORT ------------
Event: time 1608594500.444774, type 1 (EV_KEY), code 309 (BTN_Z), value 1
Event: time 1608594500.444774, type 3 (EV_ABS), code 3 (ABS_RX), value 394
Event: time 1608594500.444774, type 3 (EV_ABS), code 4 (ABS_RY), value 435
Event: time 1608594500.444774, type 3 (EV_ABS), code 5 (ABS_RZ), value 735
Event: time 1608594500.444774, -------------- SYN_REPORT ------------
Event: time 1608594500.523272, type 3 (EV_ABS), code 3 (ABS_RX), value 398
Event: time 1608594500.523272, type 3 (EV_ABS), code 4 (ABS_RY), value 437
Event: time 1608594500.523272, type 3 (EV_ABS), code 5 (ABS_RZ), value 733
Event: time 1608594500.523272, -------------- SYN_REPORT ------------
Event: time 1608594500.603247, type 1 (EV_KEY), code 309 (BTN_Z), value 0
Event: time 1608594500.603247, type 3 (EV_ABS), code 3 (ABS_RX), value 402
Event: time 1608594500.603247, -------------- SYN_REPORT ------------

/* exit with ^C */
root@raspberrypi:/home/pi#

/* remove the nunchuk device */
root@raspberrypi:/home/pi# rmmod nunchuk.ko
[  202.746933] nunchuk 1-0052: nunchuk_remove()
root@raspberrypi:/home/pi#
```

LAB 11.6: “Nunchuk provider and consumer” modules

The IIO drivers can export their channels to other consumer drivers that can use them. These devices are typically ADCs that will have other consumers within the kernel; for example battery chargers, fuel gauges, and even thermal sensors can use the IIO ADC driver´s channels. In this lab, you are going to develop two drivers: the first one will be an IIO provider driver which will read the 3-axis data from the Nunchuk´s accelerometer sensor; the second one will be an Input Subsystem consumer driver which will read the IIO channel values from the IIO provider driver and report them to the input subsystem.

Accelerometers are usually used in consumer equipments and their availability over time is very limited. Using this provider-consumer approach, it will be easy to replace the accelerometer of the Nunchuk controller with a new one that has an IIO driver available in the kernel mainline (or you can develop your own), and reuse the consumer driver without doing any modification.

You will use the same HW and SW setup from the previous LAB 10.3.

Configure the following kernel settings that will be needed during the development of the lab.

```
~/linux_rpi4/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig

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
```

Nunchuck provider module

Let's start with the development of the Nunchuk provider driver. The operation of the driver will be quite simple. You will use the IIO framework to develop a driver that will read the acceleration of the Nunchuk´s accelerometer axes.

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

1. Include the function headers:

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

```
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>
```

2. Create a private nunchuk_accel structure:

```
struct nunchuk_accel {
       struct i2c_client *client;
};
```

3. In the nunchuk_accel_probe() function, declare an instance of the private structure and allocate the iio_dev structure.

```
struct iio_dev *indio_dev;
struct nunchuk_accel *nunchuk_accel;
indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*nunchuk_accel));
```

4. Initialize the iio_device and the nunchuk_accel private structure within the nunchuk_accel_probe() function. The nunchuk_accel private structure will be previously allocated using the iio_priv() function. Keep pointers between physical devices (devices as handled by the physical bus, I2C in this case) and logical devices:

```
nunchuk_accel = iio_priv(indio_dev); /* To be able to access the private
structure in other functions of the driver you need to attach it to the
iio_dev structure using the iio_priv() function.You will retrieve the pointer
"nunchuk_accel" to the private structure using the same function iio_priv() */

nunchuck_accel->client = client; /* Keep pointer to the I2C device, needed for
exchanging data with the Nunchuk device */

indio_dev->name = "Nunchuk Accel"; /* Store the iio_dev name */

indio_dev->dev.parent = &client->dev; /* keep pointers between physical
devices (devices as handled by the physical bus, I2C in this case) and logical
devices */

indio_dev->info = &nunchuk_info; /* store the address of the iio_info
structure which contains a pointer variable to the IIO raw reading callback */

indio_dev->channels = nunchuk_channels; /* store address of the iio_chan_spec
array of structures */

indio_dev->num_channels = 3; /* set number of channels of the device */

indio_dev->modes = INDIO_DIRECT_MODE; /* device operating mode. DIRECT_MODE
indicates that the collected data is not cached, and the single data can be
read directly under sysfs */
```

5. Register the device to the IIO core. Now, the device is global to the rest of the driver functions until it is unregistered. After this call the device is ready to accept requests from user space applications.

```
devm_iio_device_register(&client->dev, indio_dev);
```

6. Create the iio_chan_spec structures to expose to the user space the sysfs attributes of each channel. The type variable specifies what type of measurement is the channel making, the acceleration in the case of our driver. The modified field of iio_chan_spec is set to 1. Modifiers are specified by using the channel2 field of the same struct iio_chan_spec and are used to indicate a physically unique characteristic of the channel such as the acceleration axis in the case of our driver. The info_mask_separate variable indicates which information should be unique to the channel.

```
#define NUNCHUK_IIO_CHAN(axis) {                        \
        .type = IIO_ACCEL,                              \
        .modified = 1,                                  \
        .channel2 = IIO_MOD_##axis,                     \
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),   \
}

static const struct iio_chan_spec nunchuk_channels[] = {
        NUNCHUK_IIO_CHAN(X),
        NUNCHUK_IIO_CHAN(Y),
        NUNCHUK_IIO_CHAN(Z),
};
```

The IIO channel definitions above will generate the following data channel access attributes below:

```
/sys/bus/iio/devices/iio:deviceX/in_accel_x_raw
/sys/bus/iio/devices/iio:deviceX/in_accel_y_raw
/sys/bus/iio/devices/iio:deviceX/in_accel_z_raw
```

7. Create the iio_info structure to declare the hooks the IIO core will use for this device. There is only one kernel hook available for interactions from the user space with the sysfs attributes of each channel.

```
static const struct iio_info nunchuk_info = {
        .read_raw     = nunchuk_accel_read_raw,
};
```

The nunchuk_accel_read_raw function returns the value of each axis when the user space reads any of the in_accel_X_raw entries. The axis of the accelerometer will be filtered using the channel2 modifier.

```
static int nunchuk_accel_read_raw(struct iio_dev *indio_dev,
                             struct iio_chan_spec const *chan,
                             int *val, int *val2, long mask)
{
        char buf[6];
```

```
	struct nunchuk_accel *nunchuk_accel = iio_priv(indio_dev);
	struct i2c_client *client = nunchuk_accel->client;

	/* read data from nunchuk */
	nunchuk_read_registers(client, buf, ARRAY_SIZE(buf));

	/* data needs to be written to 'val', 'val2' is ignored */
	switch (chan->channel2) {
	case IIO_MOD_X:
		*val = (buf[2] << 2) | ((buf[5] >> 2) & 0x3);
		break;
	case IIO_MOD_Y:
		*val = (buf[3] << 2) | ((buf[5] >> 4) & 0x3);
		break;
	case IIO_MOD_Z:
		*val = (buf[4] << 2) | ((buf[5] >> 6) & 0x3);
		break;
	default:
		return -EINVAL;
	}

	return IIO_VAL_INT;
}
```

8. Declare a list of devices supported by the driver.

```
static const struct of_device_id nunchuk_accel_of_match[] = {
	{ .compatible = "nunchuk_accel"},
	{}
};
MODULE_DEVICE_TABLE(of, nunchuk_accel_of_match);
```

9. Define an array of i2c_device_id structures:

```
static const struct i2c_device_id nunchuk_accel_id[] = {
	{ "nunchuk_accel", 0 },
	{}
};
MODULE_DEVICE_TABLE(i2c, nunchuk_accel_id);
```

10. Add an i2c_driver structure that will be registered to the I2C bus:

```
static struct i2c_driver nunchuk_accel_driver = {
	.driver = {
		.name = "nunchuk_accel",
		.owner = THIS_MODULE,
		.of_match_table = nunchuk_accel_of_match,
	},
	.probe = nunchuk_accel_probe,
	.remove = nunchuk_accel_remove,
```

```
        .id_table = nunchuk_accel_id,
};
```

11. Register your driver with the I2C bus:

```
module_i2c_driver(nunchuk_accel_driver);
```

Listing 11-12: nunchuk_accel.c

```
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>

struct nunchuk_accel {
   struct i2c_client *client;
};

#define NUNCHUK_IIO_CHAN(axis) {                          \
   .type = IIO_ACCEL,                                     \
   .modified = 1,                                         \
   .channel2 = IIO_MOD_##axis,                            \
   .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),          \
}

static const struct iio_chan_spec nunchuk_channels[] = {
   NUNCHUK_IIO_CHAN(X),
   NUNCHUK_IIO_CHAN(Y),
   NUNCHUK_IIO_CHAN(Z),
};

static int nunchuk_read_registers(struct i2c_client *client,
                                 char *buf, int buf_size)
{
   int ret;

   mdelay(10);

   buf[0] = 0x00;
   ret = i2c_master_send(client, buf, 1);
   if (ret >= 0 && ret != 1)
          return -EIO;
   if (ret < 0)
          return ret;

   mdelay(10);

   ret = i2c_master_recv(client, buf, buf_size);
```

```
	if (ret >= 0 && ret != buf_size)
		return -EIO;
	if (ret < 0)
		return ret;

	return 0;
}

static int nunchuk_accel_read_raw(struct iio_dev *indio_dev,
				 struct iio_chan_spec const *chan,
				 int *val, int *val2, long mask)
{
	char buf[6];
	struct nunchuk_accel *nunchuk_accel = iio_priv(indio_dev);
	struct i2c_client *client = nunchuk_accel->client;

	/* read data from nunchuk */
	if (nunchuk_read_registers(client, buf, ARRAY_SIZE(buf)) < 0)
	{
		dev_info(&client->dev, "Error reading the nunchuk registers.\n");
		return -EIO;
	}

	/* data needs to be written to 'val', 'val2' is ignored */
	switch (chan->channel2) {
	case IIO_MOD_X:
		*val = (buf[2] << 2) | ((buf[5] >> 2) & 0x3);
		break;
	case IIO_MOD_Y:
		*val = (buf[3] << 2) | ((buf[5] >> 4) & 0x3);
		break;
	case IIO_MOD_Z:
		*val = (buf[4] << 2) | ((buf[5] >> 6) & 0x3);
		break;
	default:
		return -EINVAL;
	}

	return IIO_VAL_INT;
}

static const struct iio_info nunchuk_info = {
	.read_raw	= nunchuk_accel_read_raw,
};

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

```
int ret;
u8 buf[2];
struct iio_dev *indio_dev;

/* declare an instance of the private structure */
struct nunchuk_accel *nunchuk_accel;

dev_info(&client->dev, "nunchuk_accel_probe() function is called.\n");

/* allocate the iio_dev structure */
indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*nunchuk_accel));
if (indio_dev == NULL)
        return -ENOMEM;

nunchuk_accel = iio_priv(indio_dev);
/* Associate client->dev with nunchuk private structure */
i2c_set_clientdata(client, nunchuk_accel);
nunchuk_accel->client = client;

indio_dev->name = "Nunchuk Accel";
indio_dev->dev.parent = &client->dev;
indio_dev->info = &nunchuk_info;
indio_dev->channels = nunchuk_channels;
indio_dev->num_channels = 3;
indio_dev->modes = INDIO_DIRECT_MODE;

/* Nunchuk handshake */
buf[0] = 0xf0;
buf[1] = 0x55;
ret = i2c_master_send(client, buf, 2);
if (ret >= 0 && ret != 2)
        return -EIO;
if (ret < 0)
        return ret;

udelay(1);

buf[0] = 0xfb;
buf[1] = 0x00;
ret = i2c_master_send(client, buf, 1);
if (ret >= 0 && ret != 1)
        return -EIO;
if (ret < 0)
        return ret;

ret = devm_iio_device_register(&client->dev, indio_dev);
if (ret)
        return ret;
```

```
	dev_info(&client->dev, "nunchuk registered\n");

	return 0;
}

static int nunchuk_accel_remove(struct i2c_client *client)
{
	dev_info(&client->dev, "nunchuk_remove()\n");
	return 0;
}

/* Add entries to device tree */
static const struct of_device_id nunchuk_accel_of_match[] = {
	{ .compatible = "nunchuk_accel"},
	{}
};
MODULE_DEVICE_TABLE(of, nunchuk_accel_of_match);


static const struct i2c_device_id nunchuk_accel_id[] = {
	{ "nunchuk_accel", 0 },
	{}
};
MODULE_DEVICE_TABLE(i2c, nunchuk_accel_id);

/* create struct i2c_driver */
static struct i2c_driver nunchuk_accel_driver = {
	.driver = {
		.name = "nunchuk_accel",
		.owner = THIS_MODULE,
		.of_match_table = nunchuk_accel_of_match,
	},
	.probe = nunchuk_accel_probe,
	.remove = nunchuk_accel_remove,
	.id_table = nunchuk_accel_id,
};

/* Register to i2c bus as a driver */
module_i2c_driver(nunchuk_accel_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a nunchuk Accelerometer IIO framework I2C driver");
```

Nunchuck consumer module

Let's continue with the development of the Nunchuk consumer driver. You will use the Input Subsystem framework to develop a platform driver that will consume the IIO channels of the Nunchuk provider driver.

The Nunchuk consumer driver will be very similar to the Nunchuk driver which was described in the LAB 10.3, so we will only describe the code used for the consumer operations:

1. You will declare a private structure which includes pointers variables to iio_channels structures.

```
/* create private structure */
struct nunchuk_dev {
	struct input_polled_dev *polled_input;
	/* declare pointers to the IIO channels of the provider device */
	struct iio_channel *accel_x, *accel_y, *accel_z;
};
```

2. In the nunchuk_probe() function, you will get the pointers to the iio_channels of the provider device and store them in your private structure.

```
/* Get pointer to channel "accel_x" of the provider device */
nunchuk->accel_x = devm_iio_channel_get(dev, "accel_x");

/* Get pointer to channel "accel_y" of the provider device */
nunchuk->accel_y = devm_iio_channel_get(dev, "accel_y");

/* Get pointer to channel "accel_z" of the provider device */
nunchuk->accel_z = devm_iio_channel_get(dev, "accel_z");
```

3. In the nunchuk_poll() function, you will get the IIO channel raw values from the provider device and report the ABS_RX, ABS_RY, and ABS_RZ events to the input system.

```
static void nunchuk_poll(struct input_polled_dev *polled_input)
{
	int accel_x, accel_y, accel_z;
	struct nunchuk_dev *nunchuk;

	nunchuk = polled_input->private;

	/* Read IIO "accel_x" channel raw value from the provider device */
	iio_read_channel_raw(nunchuk->accel_x, &accel_x);

	/* Report ABS_RX event to the input system */
	input_report_abs(polled_input->input, ABS_RX, accel_x);

	/* Read IIO "accel_y" channel raw value from the provider device */
	iio_read_channel_raw(nunchuk->accel_y, &accel_y);
```

```
        /* Report ABS_RY event to the input system */
        input_report_abs(polled_input ->input, ABS_RY, accel_y);

        /* Read IIO "accel_z" channel raw value from the provider device */
        iio_read_channel_raw(nunchuk->accel_x, &accel_z);

        /* Report ABS_RZ event to the input system */
        input_report_abs(polled_input->input, ABS_RZ, accel_z);

        input_sync(polled_input->input);
}
```

Listing 11-13: nunchuk_consumer.c

```
#include <linux/module.h>
#include <linux/delay.h>
#include <linux/input.h>
#include <linux/iio/consumer.h>
#include <linux/iio/types.h>
#include <linux/input-polldev.h>
#include <linux/platform_device.h>

/* create private structure */
struct nunchuk_dev {
   struct input_polled_dev *polled_input;
   /* declare pointers to the IIO channels of the provider device */
   struct iio_channel *accel_x, *accel_y, *accel_z;
};

/*
 * poll handler function read the harware,
 * queue events to be reported (input_report_*)
 * and flush the queued events (input_sync)
 */
static void nunchuk_poll(struct input_polled_dev *polled_input)
{
   int accel_x, accel_y, accel_z;
   struct nunchuk_dev *nunchuk;
   int ret;

   nunchuk = polled_input->private;

   /* Read IIO "accel_x" channel raw value from the provider device */
   ret = iio_read_channel_raw(nunchuk->accel_x, &accel_x);
   if (unlikely(ret < 0))
          return;
```

```
    /* Report ABS_RX event to the input system */
    input_report_abs(polled_input->input, ABS_RX, accel_x);

    /* Read IIO "accel_y" channel raw value from the provider device */
    ret = iio_read_channel_raw(nunchuk->accel_y, &accel_y);
    if (unlikely(ret < 0))
            return;

    /* Report ABS_RY event to the input system */
    input_report_abs(polled_input ->input, ABS_RY, accel_y);

    /* Read IIO "accel_z" channel raw value from the provider device */
    ret = iio_read_channel_raw(nunchuk->accel_x, &accel_z);
    if (unlikely(ret < 0))
            return;

    /* Report ABS_RZ event to the input system */
    input_report_abs(polled_input->input, ABS_RZ, accel_z);

    /*
     * Tell those who receive the events
     * that a complete report has been sent
     */
    input_sync(polled_input->input);
}

static int nunchuk_probe(struct platform_device *pdev)
{
    int ret;
    struct device *dev = &pdev->dev;
    enum iio_chan_type type;

    /* declare a pointer to the private structure */
    struct nunchuk_dev *nunchuk;

    /* declare pointers to input_dev and input_polled_dev structures */
    struct input_polled_dev *polled_device;
    struct input_dev *input;

    dev_info(dev, "nunchuck_probe() function is called.\n");

    /* allocate private structure for nunchuk device */
    nunchuk = devm_kzalloc(dev, sizeof(*nunchuk), GFP_KERNEL);
    if (nunchuk == NULL)
            return -ENOMEM;

    /* Get pointer to channel "accel_x" of the provider device */
    nunchuk->accel_x = devm_iio_channel_get(dev, "accel_x");
```

```
if (IS_ERR(nunchuk->accel_x))
        return PTR_ERR(nunchuk->accel_x);

if (!nunchuk->accel_x->indio_dev)
        return -ENXIO;

/* Get type of "accel_x" channel */
ret = iio_get_channel_type(nunchuk->accel_x, &type);
if (ret < 0)
        return ret;

if (type != IIO_ACCEL) {
        dev_err(dev, "not accelerometer channel %d\n", type);
        return -EINVAL;
}

/* Get pointer to channel "accel_y" of the provider device */
nunchuk->accel_y = devm_iio_channel_get(dev, "accel_y");
if (IS_ERR(nunchuk->accel_y))
        return PTR_ERR(nunchuk->accel_y);

if (!nunchuk->accel_y->indio_dev)
        return -ENXIO;

/* Get type of "accel_y" channel */
ret = iio_get_channel_type(nunchuk->accel_y, &type);
if (ret < 0)
        return ret;

if (type != IIO_ACCEL) {
        dev_err(dev, "not accel channel %d\n", type);
        return -EINVAL;
}

/* Get pointer to channel "accel_z" of the provider device */
nunchuk->accel_z = devm_iio_channel_get(dev, "accel_z");
if (IS_ERR(nunchuk->accel_z))
        return PTR_ERR(nunchuk->accel_z);

if (!nunchuk->accel_z->indio_dev)
        return -ENXIO;

/* Get type of "accel_z" channel */
ret = iio_get_channel_type(nunchuk->accel_z, &type);
if (ret < 0)
        return ret;

if (type != IIO_ACCEL) {
```

```
        dev_err(dev, "not accel channel %d\n", type);
        return -EINVAL;
}

/* Allocate the struct input_polled_dev */
polled_device = devm_input_allocate_polled_device(dev);
if (!polled_device) {
        dev_err(dev, "unable to allocate input device\n");
        return -ENOMEM;
}

/* Initialize the polled input device */
/* To recover nunchuk in the poll() function */
polled_device->private = nunchuk;

/* Fill in the poll interval */
polled_device->poll_interval = 50;

/* Fill in the poll handler */
polled_device->poll = nunchuk_poll;

polled_device->input->name = "WII accel consumer";
polled_device->input->id.bustype = BUS_HOST;

/*
 * Store the polled device in the global structure
 * to recover it in the remove() function
 */
nunchuk->polled_input = polled_device;

input = polled_device->input;

/* To recover nunchuck structure from remove() function */
platform_set_drvdata(pdev, nunchuk);

/*
 * Set EV_ABS type events and from those
 * ABS_X, ABS_Y, ABS_RX, ABS_RY and ABS_RZ event codes
 */
set_bit(EV_ABS, input->evbit);
set_bit(ABS_RX, input->absbit); /* accelerometer */
set_bit(ABS_RY, input->absbit);
set_bit(ABS_RZ, input->absbit);

/*
 * fill additional fields in the input_dev struct for
 * each absolute axis nunchuk has
 */
```

```
	input_set_abs_params(input, ABS_RX, 0x00, 0x3ff, 0, 0);
	input_set_abs_params(input, ABS_RY, 0x00, 0x3ff, 0, 0);
	input_set_abs_params(input, ABS_RZ, 0x00, 0x3ff, 0, 0);

	/* Finally, register the input device */
	ret = input_register_polled_device(nunchuk->polled_input);
	if (ret < 0)
		return ret;

	return 0;
}

static int nunchuk_remove(struct platform_device *pdev)
{
	struct nunchuk_dev *nunchuk = platform_get_drvdata(pdev);
	input_unregister_polled_device(nunchuk->polled_input);
	dev_info(&pdev->dev, "nunchuk_remove()\n");

	return 0;
}

/* Add entries to device tree */
/* Declare a list of devices supported by the driver */
static const struct of_device_id nunchuk_of_ids[] = {
	{ .compatible = "nunchuk_consumer"},
	{},
};
MODULE_DEVICE_TABLE(of, nunchuk_of_ids);

/* Define platform driver structure */
static struct platform_driver nunchuk_platform_driver = {
	.probe = nunchuk_probe,
	.remove = nunchuk_remove,
	.driver = {
		.name = "nunchuk_consumer",
		.of_match_table = nunchuk_of_ids,
		.owner = THIS_MODULE,
	}
};

/* Register our platform driver */
module_platform_driver(nunchuk_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a Nunchuk consumer platform driver");
```

LAB 11.6 device tree description

Open the bcm2711-rpi-4-b.dts DT file under the kernel sources and find the i2c1 controller master node. Inside the i2c1 node, you can see the pinctrl-0 property which configures the pins in I2C mode. The i2c1_pins are already defined in the bcm2711-rpi-4-b.dts file inside the gpio node property.

The i2c1 controller is enabled by writing "okay" to the status property. You will set to 100KHz the clock-frequency property. The Nunchuck device communicates with MPU using an I2C bus interface with a maximum frequency of 100kHz.

Now, you will remove the nunchuk node from the LAB 10.3 and add the nunchuk_accel node of our provider driver to the i2c1 controller node. There must be a DT device node´s compatible property identical to the compatible string stored in one of the driver´s of_device_id structures. The reg property includes the I2C address of the device.

You can see below the device-tree configuration for our nunchuk_accel device. The io-channel-cells property provides the number of cells in an IIO specifier; Typically 0 for nodes with a single IIO output and 1 for nodes with multiple IIO outputs (as is the case of our Nunchuk provider driver).

```
&i2c1 {
	pinctrl-names = "default";
	pinctrl-0 = <&i2c1_pins>;
	clock-frequency = <100000>;
	status = "okay";

	nunchuk_accel: nunchuk_accel@52 {
		#io-channel-cells = <1>;
		compatible = "nunchuk_accel";
		reg = <0x52>;
	};

	/* nunchuk: nunchuk@52 {
		compatible = "nunchuk";
		reg = <0x52>;
	}; */

};
```

Now, add the nunchuk_consumer node of our consumer driver to the / node. The io-channels property provides a list of phandle and IIO specifier pairs, one pair for each IIO input to the device. The io-channel-names property provides a list of IIO input name strings sorted in the same order as the io-channels property. Consumer drivers will use io-channel-names to match IIO input names with IIO specifiers.

```
// ============================================
// Board specific stuff here

/ {
	sd_vcc_reg: sd_vcc_reg {
		compatible = "regulator-fixed";
		regulator-name = "vcc-sd";
		regulator-min-microvolt = <3300000>;
		regulator-max-microvolt = <3300000>;
		regulator-boot-on;
		enable-active-high;
		gpio = <&expgpio 6 GPIO_ACTIVE_HIGH>;
	};

	nunchuk_consumer {
		compatible = "nunchuk_consumer";
		io-channels = <&nunchuk_accel 0>, <&nunchuk_accel 1>,
<&nunchuk_accel 2>;
		io-channel-names = "accel_x", "accel_y", "accel_z";
	};
};
```

LAB 11.6 driver demonstration

Download the linux_5.4_rpi4_drivers.zip file from the github of the book and unzip it in the home folder of your Linux host:

```
~/linux_5.4_rpi4_drivers$ cd nunchuk_drivers
```

Compile and deploy the drivers to the **Raspberry Pi 4 Model B** board:

```
~/linux_5.4_rpi4_drivers/nunchuk_drivers$ make
```

```
~/linux_5.4_rpi4_drivers/nunchuk_drivers$ make deploy
```

Compile now your kernel and device tree and copy them to the Raspberry Pi device:

```
~/linux_rpi4/linux$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
~/linux_rpi4/linux$ scp arch/arm/boot/dts/bcm2711-rpi-4-b.dtb
root@10.0.0.10:/boot/
```

Reboot the Raspberry Pi Board.

Follow the next instructions to test the driver:

```
/* check if the Nunchuk has been detected */
root@raspberrypi:/home/pi# i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- 52 -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
root@raspberrypi:/home/pi#

/* load the provider module */
root@raspberrypi:/home/pi# insmod nunchuk_accel.ko
[  105.571075] nunchuk_accel: loading out-of-tree module taints kernel.
[  105.578457] nunchuk_accel 1-0052: nunchuk_accel_probe() function is called.
[  105.586513] nunchuk_accel 1-0052: nunchuk registered

/* check ths sysfs entries under the iio device */
root@raspberrypi:/home/pi# cd /sys/bus/iio/devices/iio\:device0
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat name
Nunchuk Accel
root@raspberrypi:/sys/bus/iio/devices/iio:device0# ls
dev             in_accel_y_raw  name     power      uevent
in_accel_x_raw  in_accel_z_raw  of_node  subsystem

/* play with Nunchuk and read the value of the axes */
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_accel_x_raw
1007
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_accel_x_raw
856
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_accel_x_raw
703

/* load the consumer driver */
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cd /home/pi/
root@raspberrypi:/home/pi# insmod nunchuk_consumer.ko
[  169.861789] nunchuk_consumer soc:nunchuk_consumer: nunchuck_probe() function
is called.
[  169.870407] input: WII accel consumer as /devices/platform/soc/soc:nunchuk_co
nsumer/input/input0

/* launch the evtest application and play with the Nunchuck controller */
root@raspberrypi:/home/pi# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0:      WII accel consumer
Select the device event number [0-0]: 0
Input driver version is 1.0.1
Input device ID: bus 0x19 vendor 0x0 product 0x0 version 0x0
Input device name: "WII accel consumer"
Supported events:
  Event type 0 (EV_SYN)
  Event type 3 (EV_ABS)
    Event code 3 (ABS_RX)
      Value    650
```

```
      Min        0
      Max     1023
    Event code 4 (ABS_RY)
      Value    500
      Min        0
      Max     1023
    Event code 5 (ABS_RZ)
      Value    582
      Min        0
      Max     1023
Properties:
Testing ... (interrupt to exit)
Event: time 1608593843.374146, type 3 (EV_ABS), code 3 (ABS_RX), value 650
Event: time 1608593843.374146, type 3 (EV_ABS), code 4 (ABS_RY), value 500
Event: time 1608593843.374146, type 3 (EV_ABS), code 5 (ABS_RZ), value 582
Event: time 1608593843.374146, -------------- SYN_REPORT ------------
Event: time 1608593843.493549, type 3 (EV_ABS), code 3 (ABS_RX), value 499
Event: time 1608593843.493549, type 3 (EV_ABS), code 4 (ABS_RY), value 374
Event: time 1608593843.493549, type 3 (EV_ABS), code 5 (ABS_RZ), value 421
Event: time 1608593843.493549, -------------- SYN_REPORT ------------
Event: time 1608593843.613407, type 3 (EV_ABS), code 3 (ABS_RX), value 425
Event: time 1608593843.613407, type 3 (EV_ABS), code 4 (ABS_RY), value 379
Event: time 1608593843.613407, type 3 (EV_ABS), code 5 (ABS_RZ), value 438
Event: time 1608593843.613407, -------------- SYN_REPORT ------------
Event: time 1608593843.733405, type 3 (EV_ABS), code 3 (ABS_RX), value 452
Event: time 1608593843.733405, type 3 (EV_ABS), code 4 (ABS_RY), value 340
Event: time 1608593843.733405, type 3 (EV_ABS), code 5 (ABS_RZ), value 233
Event: time 1608593843.733405, -------------- SYN_REPORT ------------
Event: time 1608593843.853410, type 3 (EV_ABS), code 3 (ABS_RX), value 158
Event: time 1608593843.853410, type 3 (EV_ABS), code 4 (ABS_RY), value 469
Event: time 1608593843.853410, type 3 (EV_ABS), code 5 (ABS_RZ), value 517
Event: time 1608593843.853410, -------------- SYN_REPORT ------------
Event: time 1608593843.973406, type 3 (EV_ABS), code 3 (ABS_RX), value 520
Event: time 1608593843.973406, type 3 (EV_ABS), code 4 (ABS_RY), value 922
Event: time 1608593843.973406, type 3 (EV_ABS), code 5 (ABS_RZ), value 546

/* Press ^C to exit */
root@raspberrypi:/home/pi#

/* remove the consumer module */
root@raspberrypi:/home/pi# rmmod nunchuk_consumer.ko
[  761.187031] nunchuk_consumer soc:nunchuk_consumer: nunchuk_remove()

/* remove the provider module */
root@raspberrypi:/home/pi# rmmod nunchuk_accel.ko
[  767.490807] nunchuk_accel 1-0052: nunchuk_remove()
root@raspberrypi:/home/pi#
```

13

Linux USB Device Drivers

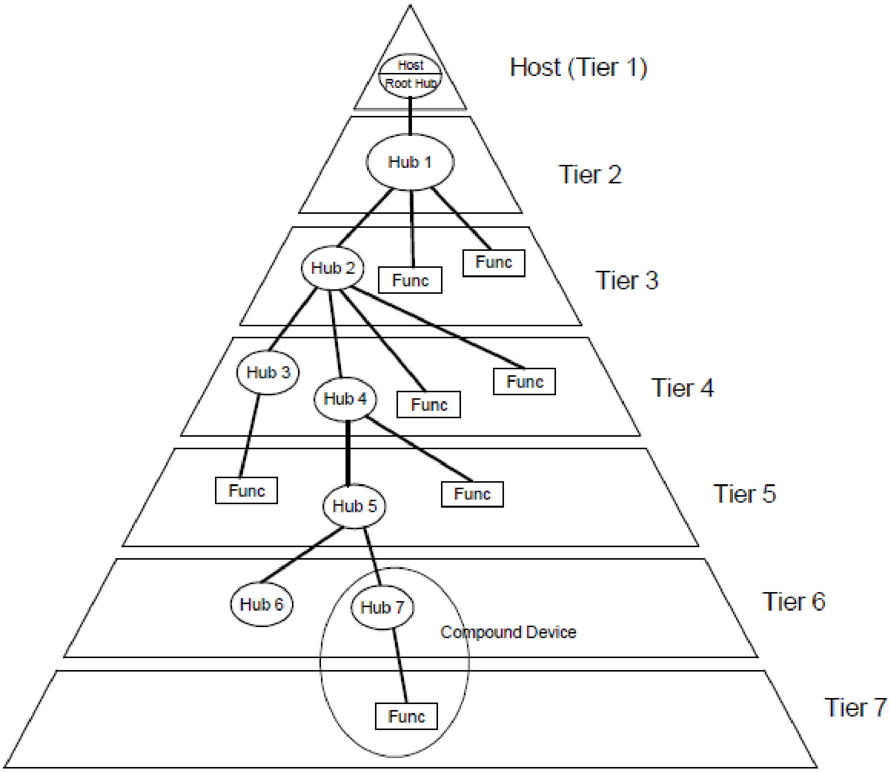
USB (abbreviation of **Universal Serial Bus**) was designed as a low cost, serial interface solution with bus power provided from the USB host to support a wide range of peripheral devices. The original bus speeds for USB were low speed at 1.5 Mbps, followed by full speed at 12 Mbps, and then high speed at 480 Mbps. With the advent of the USB 3.0 specification, the super speed was defined at 4.8 Gbps. Maximum data throughput, i.e. the line rate minus overhead is approximately 384 Kbps, 9.728 Mbps, and 425.984 Mbps for low, full and high speed respectively. Note that this is the maximum data throughput and it can be adversely affected by a variety of factors, including software processing, other USB bandwidth utilization on the same bus, etc.

One of the biggest advantages of USB is that it supports dynamic attachment and removal, which is a type of interface referred to as "plug and play". Following attachment of a USB peripheral device, the host and the device communicate to automatically advance the externally visible device state from the attached state through powered, default, addressed and finally to the configured states. Additionally, all devices must conform to the suspend state in which a very low bus power consumption specification must be met. Power conservation in the suspended state is another USB benefit.

Throughout this chapter, we will focus on the USB 2.0 specification, which includes low, full and high speed device specifications. Compliance to USB 2.0 specification for peripheral devices does not necessarily indicate that the device is a high speed device, however a hub advertised as USB 2.0 compliant, must be high speed capable. A USB 2.0 device can be High Speed, Full Speed, or Low Speed.

USB 2.0 Bus Topology

USB devices fall into the category of hubs - which provide additional downstream attachment points, or functions - which provide a capability to the system. The USB physical interconnection is a tiered star topology (see next Figure). Starting with the host and "root hub" at tier 1, up to seven tiers with a maximum of 127 devices can be supported. Tier 2 through 6 may have one or more hub devices in order to support communication to the next tier. A compound device (one which has both a hub and peripheral device functions) may not occupy tier 7.



The physical USB interconnection is accomplished for all USB 2.0 (up to high speed) devices via a simple 4-wire interface with bi-directional differential data (D+ and D-), power (VBUS) and ground. The VBUS power is nominally +5V. An "A-type" connector and mating plug are used for all host ports as well as downstream facing hub ports. A "B-type" connector and mating plug are used for all peripheral devices as well as the upstream facing port of a hub. Cable connections between host, hubs and devices can each be a maximum of 5 meters or ~16 feet. With the maximum of 7 tiers, cabling connections can be up to 30 meters or ~ 98 feet total.

USB Bus Enumeration and Device Layout

USB is a Host-controlled polled bus where all transactions are initiated by the USB host. Nothing on the bus happens without the host first initiating it; the USB devices cannot initiate a transaction, it is the host which polls each device, requesting data or sending data. All attached and removed USB devices are identified by a process termed "bus enumeration".

An attached device is recognized by the host and its speed (low, full or high) is identified via a signaling mechanism using the D+/D- USB data pair. When a new USB device is connected to the bus through a hub the device enumeration process starts. Each hub provides an IN endpoint, which is used to inform the host about newly attached devices. The host continually polls on this endpoint to receive device attachment and removal events from the hub. Once a new device was attached and the hub notified the host about this event, the USB bus driver of the host enables the attached device and starts requesting information from the device. This is done with standard USB requests which are sent through the default control pipe to endpoint zero of the device.

Information is requested in terms of **descriptors**. USB descriptors are data structures that are provided by devices to describe all of their attributes. This includes e.g. the product/vendor ID, any device class affiliation, and strings describing the product and vendor. Additionally, information about all available endpoints is provided. After the host reads all the necessary information from the device, it tries to find a matching device driver. The details of this process are dependant on the used operating system. After the first descriptors were read from the attached USB device, the host uses the vendor and product ID from the device descriptor to find a matching device driver.

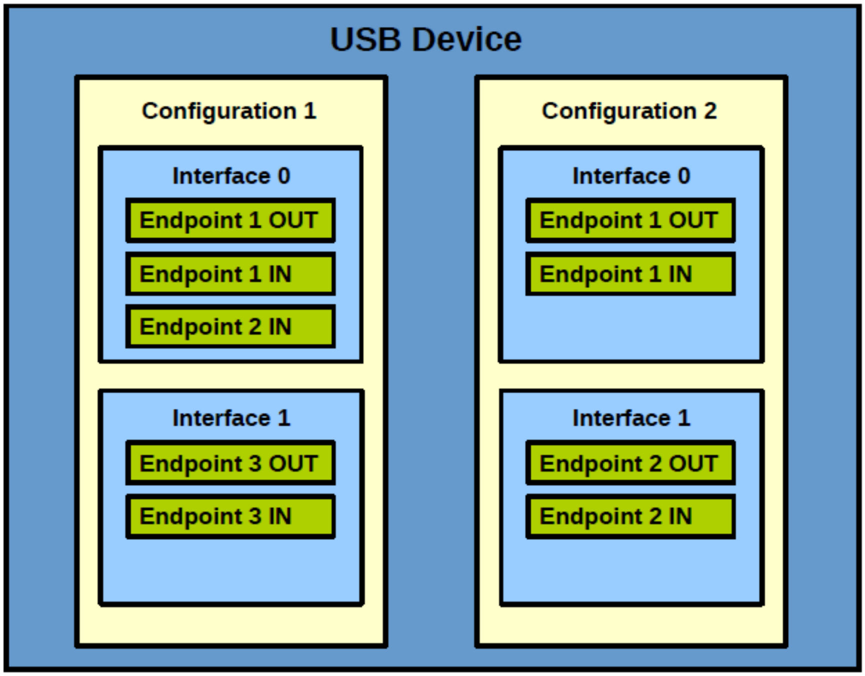
The attached device will initially utilize the default USB address of 0. Additionally, all USB devices are comprised of a number of independent endpoints, which provide a terminus for communication flow between the host and device.

Endpoints can be categorized into **control** and **data** endpoints. Every USB device must provide at least one control endpoint at address 0 called the default endpoint or Endpoint0. This endpoint is bidirectional, that is, the host can send data to the endpoint and receive data from it within one transfer. The purpose of a control transfer is to enable the host to obtain device information, configure the device, or perform control operations that are unique to the device.

The endpoint is a buffer that typically consists of a block of memory or registers which stores received data or contain data which is ready to transmit. Each endpoint is assigned a unique endpoint number determined at design time, however, all devices must support the default control endpoint (ep0) which is assigned number 0 and may transfer data in both directions. All other endpoints may transfer data in one direction (always from the host perspective), labeled "Out", i.e. data from the host, or "In", i.e. data to the host. The endpoint number is a 4-bit integer associated with an endpoint (0-15); the same endpoint number is used to describe two endpoints, for instance EP 1 IN and EP 1 OUT. An endpoint address is the combination of an endpoint number and an endpoint direction, for example: EP 1 IN, EP 1 OUT, EP 3 IN. The endpoint addresses are encoded with the direction and number in a single byte, where the direction is the MSB (1=IN, 0=OUT) and the number is the lower four bits. For example:

- EP 1 IN = 0x81
- EP 1 OUT = 0x01
- EP 3 IN = 0x83
- EP 3 OUT = 0x03

An USB configuration defines the capabilities and features of a device, mainly its power capabilities and interfaces. The device can have multiple configurations, but only one is active at a time. A configuration can have one or more USB interfaces that define the functionality of the device. Typically, there is a one-to-one correlation between a function and an interface. However, certain devices expose multiple interfaces related to one function. For example, a USB device that comprises a keyboard with a built-in speaker will offer an interface for playing audio and an interface for key presses. In addition, the interface contains alternate settings that define the bandwidth requirements of the function associated with the interface. Each interface contains one or more endpoints, which are used to transfer data to and from the device. To sum up, a group of endpoints form an interface, and a set of interfaces constitutes a configuration in the device. The following image shows a multiple-interfaces USB device:



After a matching device driver was found and loaded, it's the task of the device driver to select one of the provided device configurations, one or more interfaces within that configuration, and an alternate setting for each interface. Most USB devices don't provide multiple interfaces or multiple alternate settings. The device driver selects one of the configurations based on its own capabilities and the available bandwidth on the bus and activates this configuration on the attached device. At this point, all interfaces and their endpoints of the selected configuration are set up and the device is ready for use.

Communication from the host to each device endpoint uses a communication "pipe" which is established during enumeration. The pipe is a logical association between the host and the device. The pipe is purely a software term. A pipe talks to an endpoint on a device, and that endpoint has an address. The other end of a pipe is always the host controller. A pipe for an endpoint is opened when the device is configured either by selecting a configuration and an interface's alternate setting. Therefore, they become targets for I/O operations. A pipe has all the properties of an endpoint, but it is active and be used to communicate with the host. The combination of the device address, endpoint number and direction allows the host to uniquely reference each endpoint.

USB Data Transfers

Once the enumeration is complete, the host and device are free to carry out communications via data transfers from the host to the device or vice versa. Both directions of transfers are initiated by the host. Four different types of transfers are defined. These types are:

- **Control Transfers**: Used to configure a device at attach time and can be used for other device-specific purposes, for example device specific register read/write access as well as control of other pipes on the device. Control transfers consist of up to three distinct stages, a setup stage containing a request, a data stage if necessary to or from the host, and a status stage indicating the success of the transfer. USB has a number of standardized transactions that are implemented by using control transfers. For example, the "Set Address" and "Get Descriptor" transactions are always utilized in the device enumeration procedure described above. The "Set Configuration" request is another standard transaction which is also used during device enumeration.
- **Bulk Data Transfers**: Capable of transferring relatively large quantities of data or bursty data. Bulk transfers do not have guaranteed timing, but can provide the fastest data transfer rates if the USB bus is not occupied by other activity.
- **Interrupt Data Transfers**: Used for timely, but reliable delivery of data, for example, characters or coordinates with human-perceptible echo or feedback response characteristics. Interrupt transfers have a guaranteed maximum latency, i.e. time between transaction attempts. USB mice and keyboards typically use interrupt data transfers.
- **Isochronous Data Transfers**: Occupy a pre-negotiated amount of USB bandwidth with a pre-negotiated delivery latency. Isochronous transfers have guaranteed timing, but do not have error correction capability. Isochronous data must be delivered at the rate received to maintain its timing and additionally may be sensitive to delivery delays. A typical use for isochronous transfers would be audio or video streaming.

USB Device Classes

The USB specification and supplemental documents define a number of device classes that categorize USB devices, according to capability and interface requirements. When a host retrieves device information, class classification helps the host determine how to communicate with the USB device. The hub is a specially designated class of devices that has additional requirements in the USB specification. Other examples of classes of peripheral devices are human interface, also known as HID, printer, imaging, mass storage and communications. The USB UART devices usually fall into the communications device class (CDC) of USB devices.

Human Interface Device Class

The HID class devices usually interface with humans in some capacity. HID-class devices include mice, keyboards, printers, etc. However, the HID specification merely defines basic requirements for devices and the protocol for data transfer, and devices do not necessarily depend on any direct human interaction. HID devices must meet a few general requirements that are imposed to keep the HID interface standardized and efficient.

- All HID devices must have a control endpoint (Endpoint 0) and an interrupt IN endpoint. Many devices also use an interrupt OUT endpoint. In most cases, HID devices are not allowed to have more than one OUT and one IN endpoint.
- All data transferred must be formatted as reports whose structure is defined in the report descriptor.
- HID devices must respond to standard HID requests in addition to all standard USB requests.

Before the HID device can enter its normal operating mode and transfer data to the host, the device must properly enumerate. The enumeration process consists of a number of calls made by the host for descriptors stored in the device that describe the device's capabilities. The device must respond with descriptors that follow a standard format. Descriptors contain all basic information about a device. The USB specification defines some of the descriptors retrieved, and the HID specification defines other required descriptors. The following section discusses the descriptor structures a host expects to receive.

USB Descriptors

The host software obtains descriptors from an attached device by sending various standard control requests to the default endpoint during enumeration, immediately upon a device being attached. Those requests specify the type of descriptor to retrieve. In response to such requests, the device sends descriptors that include information about the device, its configurations, interfaces and the related endpoints. Device descriptors contain information about the whole device.

Every USB device exposes a **device descriptor** that indicates the device's class information, vendor and product identifiers, and number of configurations. Each configuration exposes it's **configuration descriptor** that indicates the number of interfaces and power characteristics. Each interface exposes an **interface descriptor** for each of its alternate settings that contains information about the class and the number of endpoints. Each endpoint within each interface exposes **endpoint descriptors** that indicate the endpoint type and the maximum packet size.

Descriptors begin with a byte describing the descriptor length in bytes. This length equals the total number of bytes in the descriptor, including the byte storing the length. The next byte indicates the descriptor type, which allows the host to correctly interpret the rest of the bytes contained in the descriptor. The content and values of the rest of the bytes are specific to the type of descriptor being transmitted. The descriptor structure must follow specifications exactly; the host will ignore received descriptors containing errors in size or value, potentially causing enumeration to fail and prohibiting further communication between the device and the host.

USB Device Descriptors

Every Universal Serial Bus (USB) device must be able to provide a single device descriptor that contains relevant information about the device. For example, the **idVendor** and **idProduct** fields specify the vendor and product identifiers, respectively. The **bcdUSB** field indicates the version of the USB specification to which the device conforms. For example, 0x0200 indicates that the device is designed as per the USB 2.0 specification. The **bcdDevice** value indicates the device defined revision number. The device descriptor also indicates the total number of configurations that the device supports. You can see below an example of a structure that contains all the device descriptor fields:

```
typedef struct __attribute__ ((packed))
{
    uint8_t bLength;                // Length of this descriptor.
    uint8_t bDescriptorType;        // DEVICE descriptor type
(USB_DESCRIPTOR_DEVICE).
    uint16_t bcdUSB;                // USB Spec Release Number (BCD).
    uint8_t bDeviceClass;           // Class code (assigned by the USB-IF). 0xFF-
Vendor specific.
    uint8_t bDeviceSubClass;        // Subclass code (assigned by the USB-IF).
    uint8_t bDeviceProtocol;        // Protocol code (assigned by the USB-IF).
0xFF-Vendor specific.
    uint8_t bMaxPacketSize0;        // Maximum packet size for endpoint 0.
    uint16_t idVendor;              // Vendor ID (assigned by the USB-IF).
    uint16_t idProduct;             // Product ID (assigned by the manufacturer).
    uint16_t bcdDevice;             // Device release number (BCD).
    uint8_t iManufacturer;          // Index of String Descriptor describing the
manufacturer.
    uint8_t iProduct;               // Index of String Descriptor describing the
product.
    uint8_t iSerialNumber;          // Index of String Descriptor with the
device's serial number.
    uint8_t bNumConfigurations;     // Number of possible configurations.

} USB_DEVICE_DESCRIPTOR
```

The first item **blength** describes the descriptor length and should be common to all USB device descriptors.

The item **bDescriptorType** is the constant one-byte designator for device descriptors and should be common to all device descriptors.

The BCD-encoded two-byte **bcdUSB** item tells the system which USB specification release guidelines the device follows. This number might need to be altered in devices that take advantage of additions or changes included in future revisions of the USB specification, as the host will use this item to help determine what driver to load for the device.

If the USB device class is to be defined in the device descriptor, this item **bDeviceClass** would contain a constant defined in the USB specification. Device classes defined in other descriptors should set the device class item in the device descriptor to 0x00.

If the device class item discussed above is set to 0x00, then the device **bDeviceSubClass** item should also be set to 0x00. This item can tell the host information about the device's subclass setting.

The item **bDeviceProtocol** can tell the host whether the device supports high-speed transfers. If the above two items are set to 0x00, this one should also be set to 0x00.

The item **bMaxPacketSize0** tells the host the maximum number of bytes that can be contained inside a single control endpoint transfer. For low-speed devices, this byte must be set to 8, while full-speed devices can have maximum endpoint 0 packet sizes of 8, 16, 32, or 64.

The two-byte item **idVendor** identifies the vendor ID for the device. Vendor IDs can be acquired through the USB.org website. Host applications will search attached USB devices' vendor IDs to find a particular device needed for an application.

Like the vendor ID, the two-byte item **idProduct** uniquely identifies the attached USB device. Product IDs can be acquired through the USB.org web site.

The item **bcdDevice** is used along with the vendor ID and the Product ID to uniquely identify each USB device.

The next three one-byte items tell the host which string array index to use when retrieving UNICODE strings describing attached devices that are displayed by the system on-screen. This string describes the manufacturer of the attached device. An **iManufacturer** string index value of 0x00 indicates to the host that the device does not have a value for this string stored in memory.

The index **iProduct** will be used when the host wants to retrieve the string that describes the attached product. For example the string could read "USB Keyboard".

The string pointed to by the index **iSerialNumber** can contain the UNICODE text for the product serial number.

This item **bNumConfigurations** tells the host how many configurations the device supports. A configuration is the definition of the device's functional capabilities, including endpoint configuration. All devices must contain at least one configuration, but more than one can be supported.

USB Configuration Descriptor

The USB device can have several different configurations, although most of the devices only have one. The configuration descriptor specifies how the device is powered, its maximum power consumption, and the number of its interfaces. There are two possible configurations, one for when the device is bus powered and another when it is mains powered. You can see below an example of a structure that contains all the configuration descriptor fields:

```
typedef struct __attribute__ ((packed))
{
    uint8_t bLength;             // Size of Descriptor in Bytes
    uint8_t bDescriptorType;     // Configuration Descriptor (0x02)
    uint16_t wTotalLength;       // Total length in bytes of data returned
    uint8_t bNumInterfaces;      // Number of Interfaces
    uint8_t bConfigurationValue; // Value to use as an argument to select this
configuration
    uint8_t iConfiguration;      // Index of String Descriptor describing this
configuration
    uint8_t bmAttributes;        // power parameters for the configuration
    uint8_t bMaxPower;           // Maximum Power Consumption in 2mA units

} USB_CONFIGURATION_DESCRIPTOR;
```

The item **blenght** defines the length of the configuration descriptor. This is a standard length.

The item **bDescriptorTye** is the constant one-byte 0x02 designator for configuration descriptors.

The two-byte **wTotalLength** item defines the length of this descriptor and all of the other descriptors associated with this configuration. For example, the length could be calculated by adding the length of the configuration descriptor, the interface descriptor, the HID class descriptor, and two endpoint descriptors associated with this interface. This two-byte item follows a "little endian" data format. The item defines the length of this descriptor and all of the other descriptors associated with this configuration.

The **bNumInterfaces** item defines the number of interface settings contained in this configuration.

The **bConfigurationValue** item is used by the SetConfiguration request to select this configuration.

The **iConfiguration** item is an index to a string descriptor describing the configuration in human readable form.

The **bmAttributes** item tells the host whether the device supports USB features such as remote wake-up. Item bits are set or cleared to describe these conditions. Check the USB specification for a detailed discussion on this item.

The **bMaxPower** item tells the host how much current the device will require to function properly at this configuration.

USB Interface Descriptor

The USB interface descriptor may contain information about alternate settings of a USB interface. The interface descriptor has a **bInterfaceNumber** field which specifies the interface number and a **bAlternateSetting** field which allows alternative settings for that interface. For example, you could have a device with two interfaces. The first interface could have a **bInterfaceNumber** set to zero, indicating it is the first interface descriptor and a **bAlternativeSetting** set to zero. The second interface could have a **bInterfaceNumber** set to one and a **bAlternativeSetting** set to zero (default). This second interface could also have a **bAlternativeSetting** set to one, being an alternative setting for the second interface.

The **bNumEndpoints** item provides the number of endpoints used by the interface.

The **bInterfaceClass**, **bInterfaceSubClass** and **bInterfaceProtocol** items specify the supported classes (e.g. HID, mass storage etc.). This allows many devices to use class drivers preventing the need to write specific drivers for your device. The **iInterface** item allows for a string description of the interface.

You can see below an example of a structure containing the interface descriptor fields:

```
typedef struct __attribute__ ((packed))
{
    uint8_t bLength;            // Size of Descriptor in Bytes (9 Bytes)
    uint8_t bDescriptorType;    // Interface Descriptor (0x04)
    uint8_t bInterfaceNumber;   // Number of Interface
    uint8_t bAlternateSetting;  // Value used to select alternative setting
    uint8_t bNumEndPoints;      // Number of Endpoints used for this interface
    uint8_t bInterfaceClass;    // Class Code (Assigned by USB Org)
    uint8_t bInterfaceSubClass; // Subclass Code (Assigned by USB Org)
    uint8_t bInterfaceProtocol; // Protocol Code (Assigned by USB Org)
    uint8_t iInterface;         // Index of String Descriptor Describing this
interface
```

```
} USB_INTERFACE_DESCRIPTOR;
```

USB Endpoint Descriptor

The USB endpoint descriptors describe endpoints which are different to endpoint zero. Endpoint zero is a control endpoint, which is configured before any other descriptors. The host will use the information returned from these USB endpoint descriptors to specify the transfer type, direction, polling interval, and maximum packet size for each endpoint. You can see below an example of a structure that contains all the endpoint descriptor fields:

```
typedef struct __attribute__ ((packed))
{
    uint8_t bLength;            // Size of Descriptor in Bytes (7 bytes)
    uint8_t bDescriptorType;    // Endpoint Descriptor (0x05)
    uint8_t bEndpointAddress;   // Endpoint Address.Bits 0..3b Endpoint Number.
Bits 4..6b Reserved.Set to Zero.Bits 7 Direction 0 = Out, 1 = In
    uint8_t bmAttributes        // Transfer type
    uint16_t wMaxPacketSize;    // Maximum Packet Size this endpoint can send or
receive
    uint8_t  bInterval;         // Interval for polling endpoint data transfers

} USB_ENDPOINT_DESCRIPTOR;
```

The **bEndpointAddress** indicates what endpoint this descriptor is describing.

The **bmAttributes** specifies the transfer type. This can either be Control, Interrupt, Isochronous or Bulk Transfers. If an Isochronous endpoint is specified, additional attributes can be selected such as the synchronisation and usage types. **Bits 0..1** are the transfer type: 00 = Control, 01 = Isochronous, 10 = Bulk, 11 = Interrupt. **Bits 2..7** are reserved. If the endpoint is Isochronous **Bits 3..2** = Synchronisation Type (ISO Mode): 00 = No Synchonisation, 01 = Asynchronous, 10 = Adaptive, 11 = Synchronous. **Bits 5..4** = Usage Type (ISO Mode): 00 = Data Endpoint, 01 = Feedback Endpoint, 10 = Explicit Feedback Data Endpoint, 11 = Reserved.

The **wMaxPacketSize** item indicates the maximum payload size for this endpoint.

The **bInterval** item is used to specify the polling interval of endpoint data transfers. Ignored for Bulk and Control Endpoints. The units are expressed in frames, thus this equates to either 1ms for low/full speed devices and 125us for high speed devices.

USB String Descriptors

The USB string descriptors (USB_STRING_DESCRIPTOR) provide human readable information to the other descriptors. They are optional. If a device does not support string descriptors, all references to string descriptors within device, configuration, and interface descriptors must be set to zero.

String descriptors are UNICODE encoded characters, so that multiple languages can be supported with a single product. When requesting a string descriptor, the requester specifies the desired language using a 16-bit language ID (LANGID) defined by the USB-IF. String index zero is used for all languages and returns a string descriptor that contains an array of two-byte LANGID codes supported by the device.

Offset	Field	Type	Size	Value	Description
0	bLength	uint8_t	N + 2	Number	Size of this descriptor in bytes.
1	bDescriptorType	uint8_t	1	Constant	String Descriptor Type
2	wLANGID[0]	uint8_t	2	Number	LANGID code zero (for example 0x0407 German (Standard)).
...	...	...	...	...	...
N	wLANGID[x]	uint8_t	2	Number	LANGID code zero x (for example 0x0409 English (United States)).

The UNICODE string descriptor is not NULL-terminated. The string length is computed by subtracting two from the value of the first byte of the descriptor.

Offset	Field	Type	Size	Value	Description
0	bLength	uint8_t	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	uint8_t	1	Constant	String Descriptor Type
2	bString	uint8_t	N	Number	UNICODE encoded string.

USB HID Descriptor

The USB HID Device Class supports devices that are used by humans to control the operation of computer systems. The HID class of devices include a wide variety of human interface, data indicator, and data feedback devices with various types of output directed to the end user. Some common examples of HID class devices include:

- Keyboards
- Pointing devices such as a standard mouse, joysticks, and trackballs
- Front-panel controls like knobs, switches, buttons, and sliders
- Controls found on telephony, gaming or simulation devices such as steering wheels, rudder pedals, and dial pads
- Data devices such as bar-code scanners, thermometers, analyzers

The following descriptors are required in an USB HID Device:

- Standard Device Descriptor
- Standard Configuration Descriptor
- Standard Interface Descriptor for the HID Class
- Class-Specific HID Descriptor
- Standard Endpoint Descriptor for Interrupt IN endpoint
- Class-Specific Report Descriptor

The Class-Specific HID descriptor looks like this:

```
typedef struct __attribute__((packed))
{
    uint8_t     bLength;
    uint8_t     bDescriptorType;
    uint16_t    bcdHID;
    uint8_t     bCountryCode;
    uint8_t     bNumDescriptors;
    uint8_t     bReportDescriptorType;
    uint16_t    wItemLength;

} USB_HID_DESCRIPTOR;
```

The **bLength** item describes the size of the HID descriptor. It can vary depending on the number of subordinate descriptors, such as report descriptors, that are included in this HID configuration definition.

The **bDescriptorType** 0x21 value is the constant one-byte designator for device descriptors and should be common to all HID descriptors.

The two-byte **bcdHID** item tells the host which version of the HID class specification the device follows. USB specification requires that this value be formatted as a binary coded decimal digit, meaning that the upper and lower nibbles of each byte represent the number '0'...9'. For example, 0x0101 represents the number 0101, which equals a revision number of 1.01 with an implied decimal point.

If the device was designed to be localized to a specific country, the **bCountryCode** item tells the host which country. Setting the item to 0x00 tells the host that the device was not designed to be localized to any country.

The **bNumDescriptors** item tells the host how many report descriptors are contained in this HID configuration. The following two-byte pairs of items describe each contained report descriptor.

The **bReportDescriptorType** item describes the first descriptor which will follow the transfer of this HID descriptor. For example, if the value is "0x22" indicates that the descriptor to follow is a report descriptor.

The **wItemLength** item tells the host the size of the descriptor that is described in the preceding item.

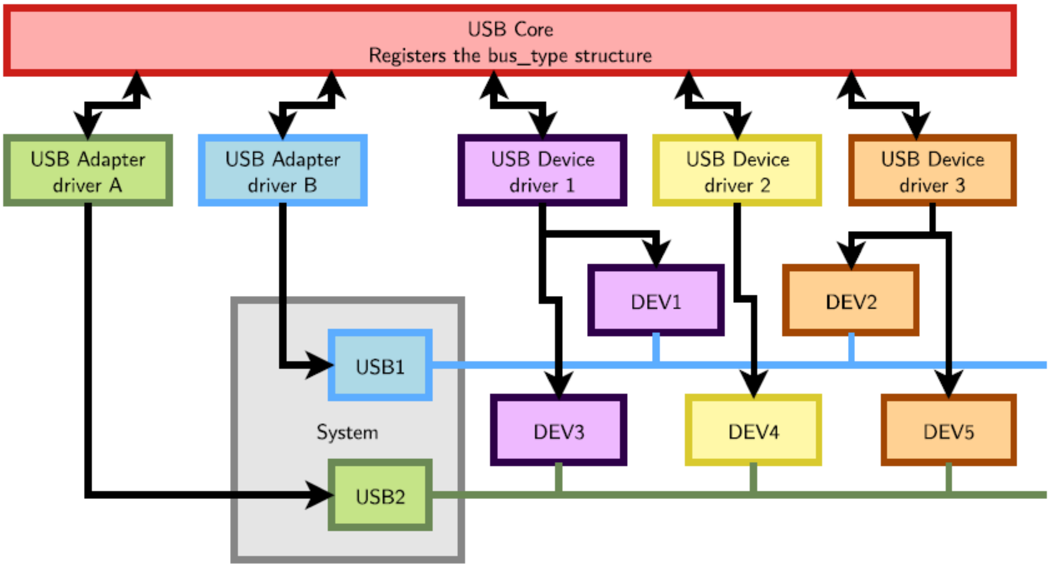
The **HID report descriptor** is a hard coded array of bytes that describe the device’s data packets. This includes: how many packets the device supports, how large are the packets, and the purpose of each byte and bit in the packet. For example, a keyboard with a calculator program button can tell the host that the button pressed/released state is stored as the 2nd bit in the 6th byte in data packet number 4.

The Linux USB Subsystem

USB support was added to Linux early in the 2.2 kernel series and has continued to develop since then. Besides support for each new generation of USB, various host controllers gained support, new drivers for peripherals have been added and advanced features for latency measurement and improved power management introduced.

The USB Linux framework supports USB devices as well as on the hosts that control the devices. The USB Linux framework also supports gadget drivers and classes to be used within these peripheral devices. We will focus on this chapter in the development of Linux USB device drivers running on hosts.

In Linux the "USB Core" is a specific API implemented to support USB peripheral devices and host controllers. This API abstracts all hardware by defining a set of data structures, macros and functions. Host-side drivers for USB devices talk to these "usbcore" APIs. There are two sets of APIs, one is intended for general-purpose USB device drivers (the ones that will be developed through this chapter), and the other is for drivers that are part of the core. Such core drivers include the hub driver (which manages trees of USB devices) and several different kinds of USB host adapter drivers, which control individual busses. The following image shows an example of a Linux USB Subsystem:



The Linux USB API supports synchronous calls for control and bulk messages. It also supports asynchronous calls for all kinds of data transfer by using request structures called "URBs" (USB Request Blocks).

The only host-side drivers that actually touch hardware (reading/writing registers, handling IRQs, and so on) are the Host Controller Devices (HCDs) drivers. In theory, all HCDs provide the same functionality through the same API. In practice, that's becoming more true, but there are still differences that crop up, especially with fault handling on the less common controllers. Different controllers don't necessarily report the same aspects of failures, and recovery from faults (including software-induced ones like unlinking an URB) isn't yet fully consistent.

The main focus of this chapter is the development of Linux Host USB device drivers. All the sections that follow are related to the development of this type of drivers.

Writing Linux USB Device Drivers

In the following labs, you will develop several USB device drivers through which you will understand the basic framework of a Linux USB device driver. But before you proceed with the labs, you need to familiarize yourselves with the main USB data structures and functions. The following sections will explain these structures and functions in detail.

USB Device Driver Registration

The first thing a Linux USB device driver needs to do is register itself with the Linux USB core, giving it some information about which devices the driver supports and which functions to call when a device supported by the driver is inserted or removed from the system. All of this information is passed to the USB core in the usb_driver structure. See below the usb_driver definition for a USB seven segment driver located at /linux/drivers/usb/misc/usbsevseg.c:

```
static struct usb_driver sevseg_driver = {
   .name =               "usbsevseg",
   .probe =              sevseg_probe,
   .disconnect =         sevseg_disconnect,
   .suspend =            sevseg_suspend,
   .resume =             sevseg_resume,
   .reset_resume =       sevseg_reset_resume,
   .id_table =           id_table,
};
```

The variable name is a string that describes the driver. It is used in informational messages printed to the system log. The probe() and disconnect() hotplugging callbacks are called when a device that matches the information provided in the id_table variable is either seen or removed.

The probe() function is called by the USB core into the driver to see if the driver is willing to manage a particular interface on a device. If it is, the probe() function returns zero and uses usb_set_intfdata() to associate driver specific data with the interface. It may also use usb_set_interface() to specify the appropriate altsetting. If unwilling to manage the interface, return -ENODEV, if genuine IO errors occurred, an appropriate negative errno value.

```
int (* probe) (struct usb_interface *intf,const struct usb_device_id *id);
```

The disconnect() callback is called when the interface is no longer accessible, usually because its device has been (or is being) disconnected or the driver module is being unloaded:

```
void disconnect(struct usb device *dev, void *drv context);
```

In the usb_driver structure, some power management (PM) callbacks are defined:

- **suspend**: called when the device is going to be suspended.
- **resume**: called when the device is being resumed.
- **reset_resume**: called when the suspended device has been reset instead of being resumed.

And there are also defined some device level operations:

- **pre_reset**: called when the device is about to be reset.
- **post_reset**: called after the device has been reset.

The USB device drivers use an ID table to support hotplugging. The pointer variable id_table included in the usb_driver structure points to an array of structures of type usb_device_id that announce the devices that the USB device driver supports. Most drivers use the USB_DEVICE() macro to create usb_device_id structures. These structures are registered to the USB core by using the MODULE_DEVICE_TABLE(usb, xxx) macro. The following lines of code included in the /linux/drivers/usb/misc/usbsevseg.c driver creates and register a USB device to the USB core:

```
#define VENDOR_ID        0x0fc5
#define PRODUCT_ID       0x1227

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
   { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
   { },
};
MODULE_DEVICE_TABLE(usb, id_table);
```

The usb_driver structure is registered to the bus core by using the module_usb_driver() function:

```
module_usb_driver(sevseg_driver);
```

Linux Host-Side Data Types

USB device drivers actually bind to interfaces, not devices. Think of them as "interface drivers", though you may not see many devices where the distinction is important. Most USB devices are simple, with only one function, one configuration, one interface, and one alternate setting. The USB interface is represented by the usb_interface structure. This is what the USB core passes to the USB driver´s probe() function when this callback function is being called.

```
struct usb_interface {
  struct usb_host_interface * altsetting;
  struct usb_host_interface * cur_altsetting;
  unsigned num_altsetting;
  struct usb_interface_assoc_descriptor * intf_assoc;
  int minor;
  enum usb_interface_condition condition;
  unsigned sysfs_files_created:1;
  unsigned ep_devs_created:1;
  unsigned unregistering:1;
  unsigned needs_remote_wakeup:1;
  unsigned needs_altsetting0:1;
  unsigned needs_binding:1;
  unsigned resetting_device:1;
  unsigned authorized:1;
  struct device dev;
  struct device * usb_dev;
  atomic_t pm_usage_cnt;
```

```
  struct work_struct reset_ws;
};
```

These are the main members of the usb_interface structure:

- **altsetting**: array of usb_host_interface structures, one for each alternate setting that may be selected. Each one includes a set of endpoint configurations. They will be in no particular order. The usb_host_interface structure for each alternate setting allows to access the usb_endpoint_descriptor structure for each of its endpoints:

  ```
  interface->altsetting[i]->endpoint[j]->desc
  ```

- **cur_altsetting**: the current altsetting.
- **num_altsetting**: number of altsettings defined.

Each interface may have alternate settings. The initial configuration of a device sets altsetting 0, but the device driver can change that setting by using usb_set_interface(). Alternate settings are often used to control the use of periodic endpoints, such as by having different endpoints use different amounts of reserved USB bandwidth. All standards-conformant USB devices that use isochronous endpoints will use them in non-default settings.

The usb_host_interface structure includes an array of usb_host_endpoint structures.

```
/* host-side wrapper for one interface setting's parsed descriptors */
struct usb_host_interface {
   struct usb_interface_descriptor     desc;

   int extralen;
   unsigned char *extra;   /* Extra descriptors */

   /* array of desc.bNumEndpoints endpoints associated with this
    * interface setting.  these will be in no particular order.
    */
   struct usb_host_endpoint *endpoint;

   char *string;         /* iInterface string, if present */
};
```

Each usb_host_endpoint structure includes a usb_endpoint_descriptor structure.

```
struct usb_host_endpoint {
   struct usb_endpoint_descriptor      desc;
   struct usb_ss_ep_comp_descriptor    ss_ep_comp;
   struct usb_ssp_isoc_ep_comp_descriptor     ssp_isoc_ep_comp;
   struct list_head             urb_list;
   void                         *hcpriv;
   struct ep_device             *ep_dev;      /* For sysfs info */
```

```
    unsigned char *extra;   /* Extra descriptors */
    int extralen;
    int enabled;
    int streams;
};
```

The usb_endpoint_descriptor structure contains all the USB-specific data announced by the device itself.

```
struct usb_endpoint_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    __u8  bEndpointAddress;
    __u8  bmAttributes;
    __le16 wMaxPacketSize;
    __u8  bInterval;

    /* NOTE:  these two are _only_ in audio endpoints. */
    /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
    __u8  bRefresh;
    __u8  bSynchAddress;
} __attribute__ ((packed));
```

You can use the following code to obtain the IN and OUT endpoint addresses from the IN and OUT endpoint descriptors, which are included in the current altsetting of the USB interface:

```
struct usb_host_interface *altsetting = intf->cur_altsetting;
int ep_in, ep_out;

/* there are two usb_host_endpoint structures in this interface altsetting.Each
usb_host_endpoint structure contains a usb_endpoint_descriptor */
ep_in = altsetting->endpoint[0].desc.bEndpointAddress;
ep_out = altsetting->endpoint[1].desc.bEndpointAddress;
```

USB Request Block (URB)

Any communication between the host and device is done asynchronously by using USB Request Blocks (urbs).

- An URB consists of all relevant information to execute any USB transaction and deliver the data and status back.
- Execution of an URB is inherently an asynchronous operation, i.e. the usb_submit_urb() call returns immediately after it has successfully queued the requested action.
- Transfers for one URB can be canceled with usb_unlink_urb() at any time.

- Each URB has a completion handler, which is called after the action has been successfully completed or canceled. The URB also contains a context-pointer for passing information to the completion handler.
- Each endpoint for a device logically supports a queue of requests. You can fill that queue, so that the USB hardware can still transfer data to an endpoint while your driver handles the completion of another. This maximizes use of USB bandwidth, and supports seamless streaming of data to (or from) devices when using periodic transfer modes.

These are some fields of the urb structure:

```
struct urb
{
      // (IN) device and pipe specify the endpoint queue
      struct usb_device *dev;         // pointer to associated USB device
      unsigned int pipe;              // endpoint information

      unsigned int transfer_flags;    // URB_ISO_ASAP, URB_SHORT_NOT_OK, etc.

      // (IN) all urbs need completion routines
      void *context;                  // context for completion routine
      usb_complete_t complete;        // pointer to completion routine

      // (OUT) status after each completion
      int status;                     // returned status

      // (IN) buffer used for data transfers
      void *transfer_buffer;          // associated data buffer
      u32 transfer_buffer_length;     // data buffer length
      int number_of_packets;          // size of iso_frame_desc

      // (OUT) sometimes only part of CTRL/BULK/INTR transfer_buffer is used
      u32 actual_length;              // actual data buffer length

     // (IN) setup stage for CTRL (pass a struct usb_ctrlrequest)
      unsigned char *setup_packet;    // setup packet (control only)

     // Only for PERIODIC transfers (ISO, INTERRUPT)
     // (IN/OUT) start_frame is set unless URB_ISO_ASAP isn't set
      int start_frame;                // start frame
      int interval;                   // polling interval

      // ISO only: packets are only "best effort"; each can have errors
      int error_count;                // number of errors
      struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

The USB driver must create a "pipe" using values from the appropriate endpoint descriptor in an interface that it's claimed.

URBs are allocated by calling usb_alloc_urb():

```
struct urb *usb_alloc_urb(int isoframes, int mem_flags)
```

The return value is a pointer to the allocated URB, 0 if the allocation failed. The parameter isoframes specifies the number of isochronous transfer frames you want to schedule. For CTRL/BULK/INT, use 0. The mem_flags parameter holds standard memory allocation flags, letting you control (among other things) whether the underlying code may block or not.

To free an URB, use usb_free_urb():

```
void usb_free_urb(struct urb *urb)
```

Interrupt transfers are periodic, and happen in intervals that are powers of two (1, 2, 4 etc.) units. Units are frames for full and low speed devices, and microframes for high speed ones. You can use the usb_fill_int_urb() macro to fill INT transfer fields. When the write urb is filled up with the proper information by using the usb_fill_int_urb() function, you should point the urb's completion callback to call your own callback function. This function is called when the urb is finished by the USB subsystem. The callback function is called in interrupt context, so caution must be taken not to do very much processing at that time. The usb_submit_urb() call modifies urb->interval to the implemented interval value that is less than or equal to the requested interval value.

An URB is submitted by using the function usb_submit_urb():

```
int usb_submit_urb(struct urb *urb, int mem_flags)
```

The mem_flags parameter, such as GFP_ATOMIC, controls memory allocation, such as whether the lower levels may block when memory is tight. It immediately returns, either with status 0 (request queued) or some error code, usually caused by the following:

- Out of memory (-ENOMEM)
- Unplugged device (-ENODEV)
- Stalled endpoint (-EPIPE)
- Too many queued ISO transfers (-EAGAIN)
- Too many requested ISO frames (-EFBIG)
- Invalid INT interval (-EINVAL)
- More than one packet for INT (-EINVAL)

After submission, urb->status is -EINPROGRESS; however, you should never look at that value except in your completion callback.

There are two ways to cancel an URB you've submitted but which hasn't been returned to your driver yet. For an asynchronous cancel, call usb_unlink_urb():

```
int usb_unlink_urb(struct urb *urb)
```

It removes the urb from the internal list and frees all allocated HW descriptors. The status is changed to reflect unlinking. Note that the URB will not normally have finished when usb_unlink_urb() returns; you must still wait for the completion handler to be called.

To cancel an URB synchronously, call usb_kill_urb():

```
void usb_kill_urb(struct urb *urb)
```

It does everything usb_unlink_urb() does, and in addition it waits until after the URB has been returned and the completion handler has finished.

The completion handler is of the following type:

```
typedef void (*usb_complete_t)(struct urb *)
```

In the completion handler, you should have a look at urb->status to detect any USB errors. Since the context parameter is included in the URB, you can pass information to the completion handler.

LAB 13.1: USB HID Device Application

In this first USB lab, you will learn how to create a fully functional USB HID device and how to send and receive data by using HID reports. For this lab, you are going to use the Curiosity PIC32MX470 Development Board:

https://www.microchip.com/DevelopmentTools/ProductDetails/dm320103#additional-summary

The Curiosity PIC32 MX470 Development Board features PIC32MX Series (PIC32MX470512H) with a 120MHz CPU, 512KB Flash, 128KB RAM , Full Speed USB and multiple expansion options.

The Curiosity Development Board includes an integrated programmer/debugger, excellent user experience options with Multiple LED's, RGB LED and a switch. Each board provides two MikroBus® expansion sockets from MicroElektronika, an I/O expansion header and a Microchip X32 header, which enable customers seeking accelerated application prototype development.

The board is fully integrated with Microchip's MPLAB® X IDE and into PIC32's powerful software framework, MPLAB ® Harmony that a provides flexible and modular interface for application development, a rich set of inter-operable software stack (TCP-IP, USB) and easy to use features.

These are the SW and HW requirements for the lab. The applications of this chapter have been developed using the Windows versions of the tools and are included in the github of this book in the PIC32MX_usb_labs.zip file

- **Development Environment**: MPLAB® X IDE v5.10
- **C Compiler**: MPLAB® XC32 v2.15
- **Software Tools**: MPLAB® Harmony Integrated Software Framework v2.06. GenericHIDSimpleDemo application ("hid_basic" example of Harmony)
- **Hardware Tools**: Curiosity PIC32MX470 Development Board (dm320103)

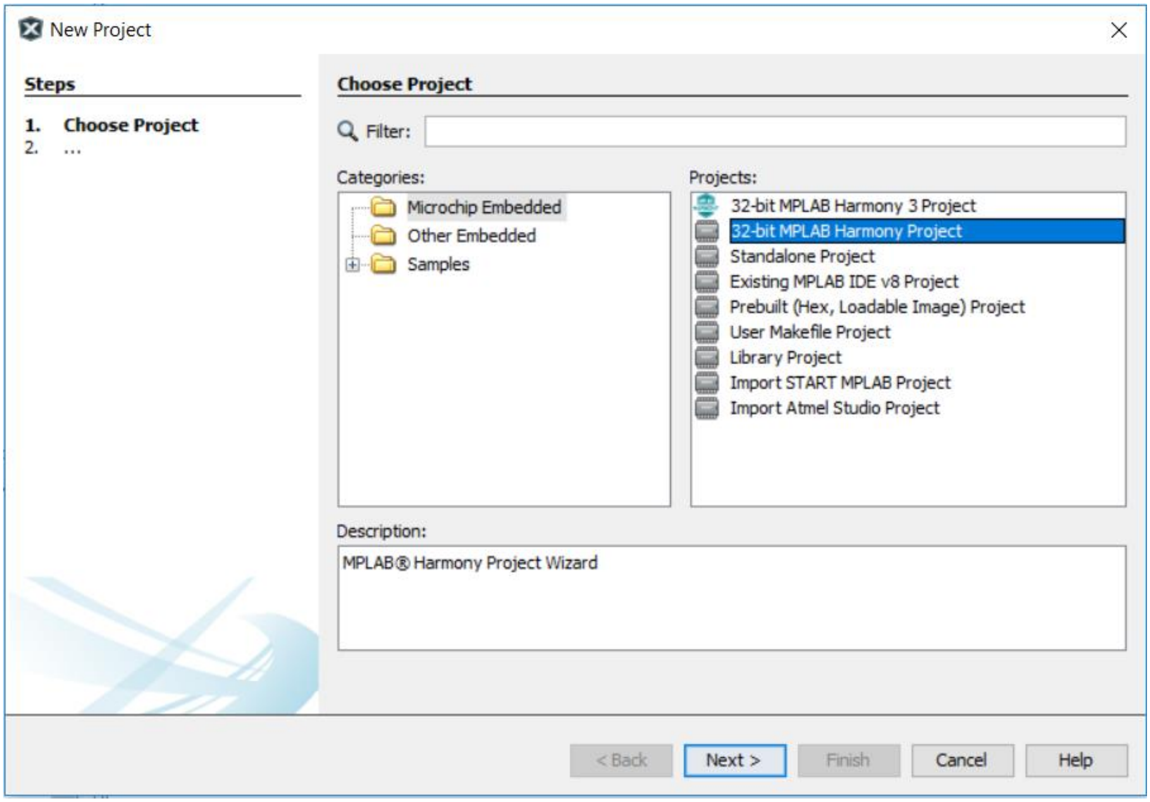
You can download all the previous SW versions from the following links:

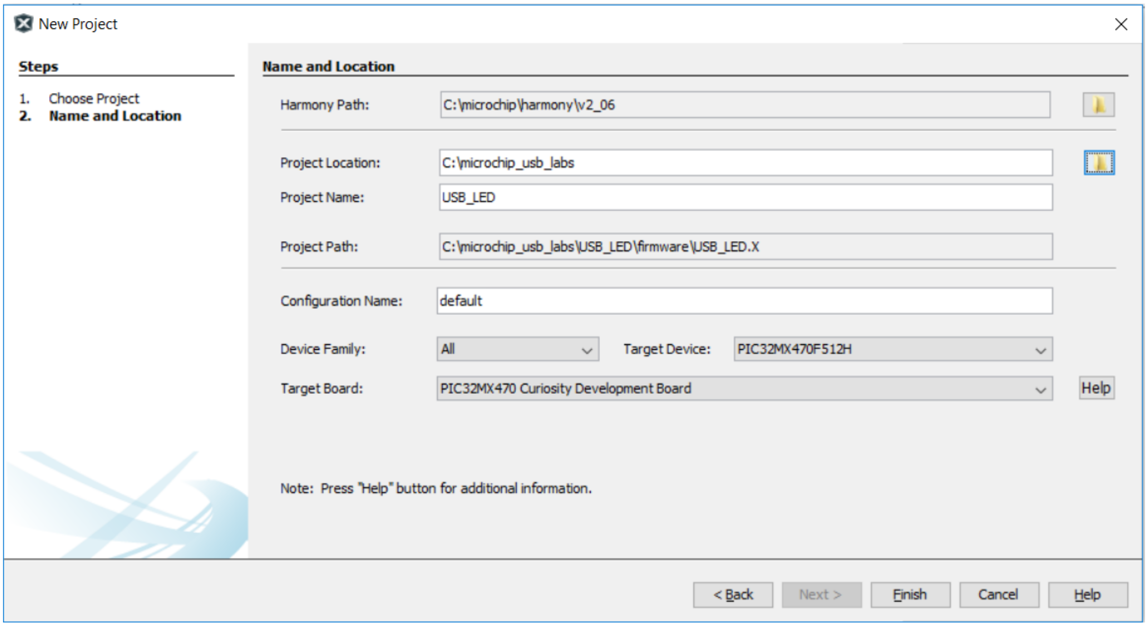
https://www.microchip.com/development-tools/pic-and-dspic-downloads-archive
https://www.microchip.com/mplab/mplab-harmony/mplab-harmony-v2

The objective of this lab is using the MPLAB® Harmony Configurator Tool, create a MPLAB X project and write the code to make a USB Device, so that it can be enumerated as a HID device and communicate with the Linux USB host driver that you will develop in the following lab.

STEP 1: Create a New Project

Create an empty 32-bit MPLAB Harmony Project, named USB_LED, for the Curiosity development board. Save the project in the following folder that was previously created: C:\microchip_usb_labs.

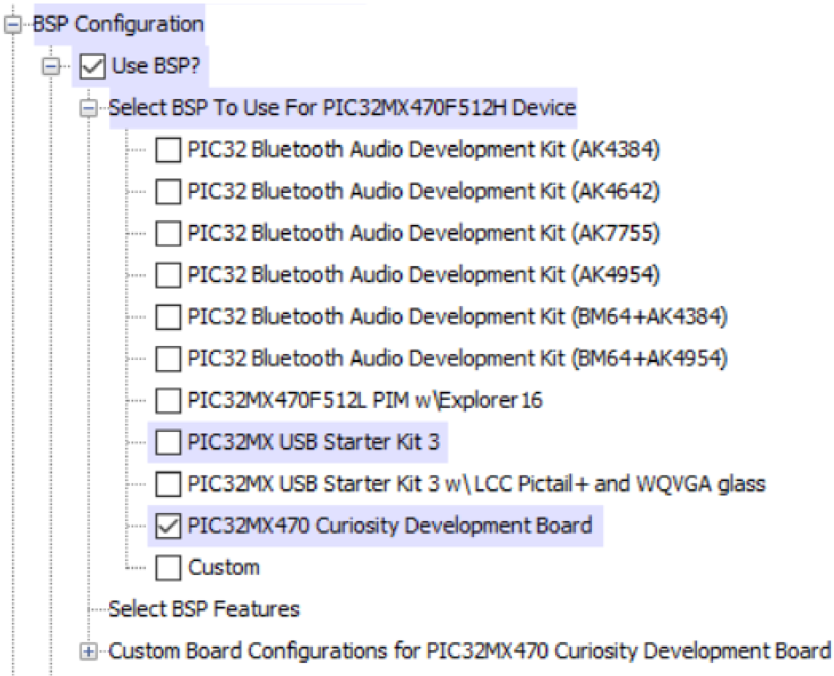

New Project
Steps
1. Choose Project
2. ...
Choose Project
Filter:
Categories:
Microchip Embedded
Other Embedded
Samples
Projects:
32-bit MPLAB Harmony 3 Project
32-bit MPLAB Harmony Project
Standalone Project
Existing MPLAB IDE v8 Project
Prebuilt (Hex, Loadable Image) Project
User Makefile Project
Library Project
Import START MPLAB Project
Import Atmel Studio Project
Description:
MPLAB® Harmony Project Wizard
< Back
Next >
Finish
Cancel
Help



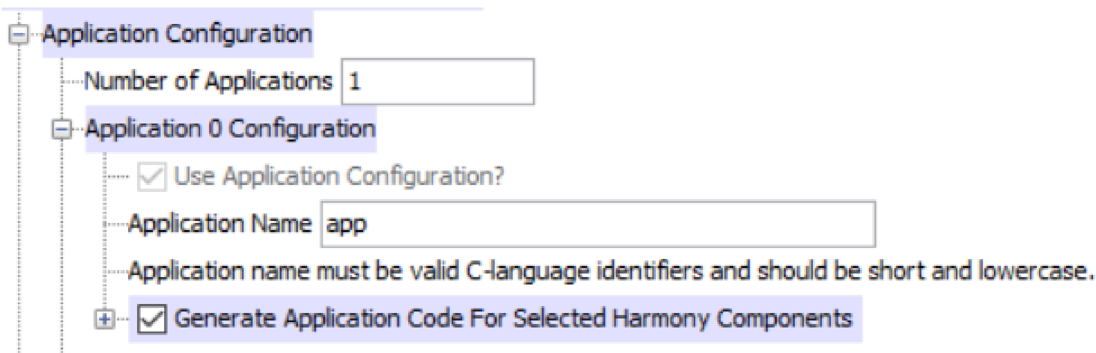
STEP 2: Configure Harmony

Launch the MPLAB Harmony Configurator plugin and click on Tools->Embedded->MPLAB Harmony Configurator.

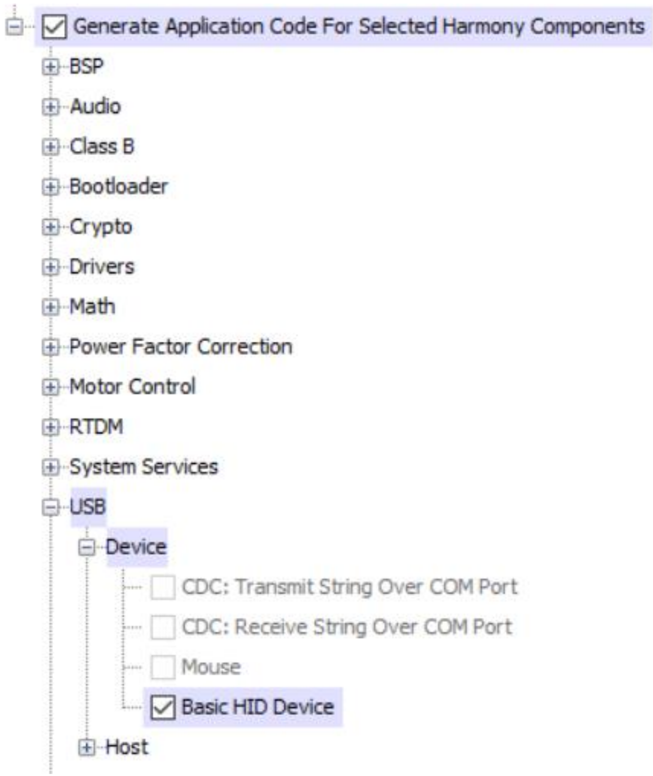
Select your demo board enabling the BSP (Board Support Package):



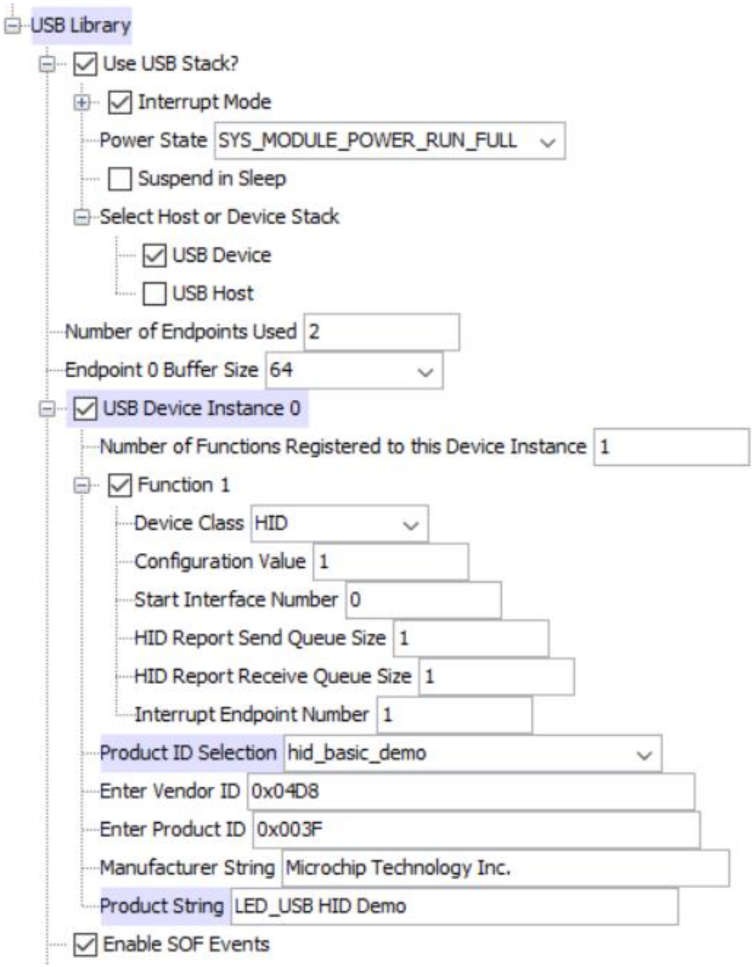
Enable the Generate Application Code For Selected Harmony Components:



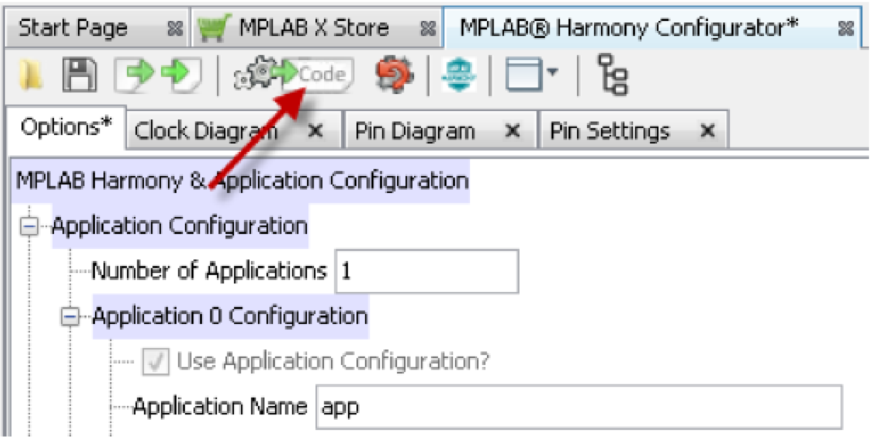
Select the Basic HID Device demo template:



At the USB Library option of Harmony Framework Configuration, select hid_basic_demo as Product ID Selection. Select also the Vendor ID, Product ID, Manufacturer String and Product String as shown in the screen capture below. You have to select a USB Device stack. The USB device will have one control endpoint (ep0) and one interrupt endpoint (composed of IN and OUT endpoints), so you will have to write the value two in the Number of Endpoints Used field. There will be only one configuration and one interface associated with the device.



Generate the code, save the modified configuration, and generate the project:



STEP 3: Modify the Generated Code

Typically, the HID class is used to implement human interface products, such as mice and keyboards. The HID protocol, is however, quite flexible, and can be adapted and used to send/receive general purpose data to/from a USB device.

In the following two labs, you will see how to use the USB protocol for basic general purpose USB data transfer. You will develop Linux USB host drivers that send USB commands to the PIC32MX USB HID device to toggle three LEDs (LED1, LED2, LED3) included in the PIC32MX Curiosity board. The PIC32MX USB HID device will also check the value of the user button (S1) and will reply to the host with a packet that contains the value of that switch.

In this lab, you have to implement the following in the USB device side:

- **Toggle LED(s)**: The Linux USB host driver sends a report to the HID device. The first byte of the report can be 0x01, 0x02, or 0x03. The HID device must toggle LED1 when it receives 0x01, LED2 when it receives 0x02 and LED3 when it receives 0x03 in the report .
- **Get Pushbutton State**: The Linux USB host driver sends a report to the HID Device. The first byte of this report is 0x00. The HID device must reply with another report, where the first byte is the status of the S1 button ("0x00" pressed, "0x01" not pressed)

By examining the `app.c` code generated by the MPLAB Harmony Configurator, the template is expecting you to implement your USB state machine inside the Function `USB_Task()`. The state machine you need to implement will be executed if the HID device is configured; if the HID device is de-configured, the USB state machine needs to return to the INIT state.

After initialization, the HID device needs to wait for a command from the host; scheduling a read request will enable the HID device to receive a report. The state machine needs to wait for the host to send the report; after receiving the report, the application needs to check the first byte of the report. If this byte is 0x01, 0x02 or 0x03, then LED1, LED2 and LED3 must be toggled. If the first byte is 0x00, a response report with the switch status must be sent to the host and then a new read must be scheduled.

STEP 4: Declare the USB State Machine States

To create the USB State Machine, you need to declare an enumeration type (e.g. USB_STATES) containing the labels for the four states, needed to implement the state machine. (e.g. USB_STATE_INIT, USB_STATE_WAITING_FOR_DATA, USB_STATE_SCHEDULE_READ, USB_STATE_SEND_REPORT). Find the section `Type Definitions` in `app.h` file and declare the enumeration type.

```
typedef enum
{
    /* Application's state machine's initial state. */
    APP_STATE_INIT=0,
    APP_STATE_SERVICE_TASKS,

    /* TODO: Define states used by the application state machine. */

} APP_STATES;

/* Declare the USB State Machine states */
typedef enum
{
    /* Application's state machine's initial state. */
    USB_STATE_INIT=0,
     USB_STATE_WAITING_FOR_DATA,
     USB_STATE_SCHEDULE_READ,
     USB_STATE_SEND_REPORT

} USB_STATES;
```

STEP 5: Add New Members to APP_DATA Type

The APP_DATA structure type already contains members needed for the Application State Machine and the enumeration process (state, handleUsbDevice, usbDeviceIsConfigured, etc.); you need to add the members you will use to send and receive HID reports.

Find the APP_DATA structure type in `app.h` file and add the following members:

- a member to store the USB State Machine status
- two pointers to buffer (one for data received and one for data to send)

- two HID transfer handles (one for reception transfer, one for the transmission transfer)
- two flags to indicate the state of the ongoing transfer (one for reception, one for transmission transfer)

```
typedef struct
{
    /* The application's current state */
    APP_STATES state;

    /* TODO: Define any additional data used by the application. */

    /*
     * USB variables used by the HID device application:
     *
     *     handleUsbDevice          : USB Device driver handle
     *     usbDeviceIsConfigured    : If true, USB Device is configured
     *     activeProtocol           : USB HID active Protocol
     *     idleRate                 : USB HID current Idle
     */
    USB_DEVICE_HANDLE                 handleUsbDevice;
    bool                              usbDeviceIsConfigured;
    uint8_t                           activeProtocol;
    uint8_t                           idleRate;

    /* Add new members to APP_DATA type */
    /* USB_Task's current state */
    USB_STATES stateUSB;

    /* Receive data buffer */
    uint8_t * receiveDataBuffer;

    /* Transmit data buffer */
    uint8_t * transmitDataBuffer;

    /* Send report transfer handle*/
    USB_DEVICE_HID_TRANSFER_HANDLE txTransferHandle;

    /* Receive report transfer handle */
    USB_DEVICE_HID_TRANSFER_HANDLE rxTransferHandle;

    /* HID data received flag*/
    bool hidDataReceived;

    /* HID data transmitted flag */
    bool hidDataTransmitted;

} APP_DATA;
```

STEP 6: Declare the Reception and Transmission Buffers

To schedule a report receive or a report send request, you need to provide a pointer to a buffer to store the received data and the data that has to be transmitted. Find the section Global Data Definitions in app.c file and declare two 64 byte buffers.

```
APP_DATA appData;

/* Declare the reception and transmission buffers */
uint8_t receiveDataBuffer[64] __attribute__((aligned(16)));
uint8_t transmitDataBuffer[64] __attribute__((aligned(16)));
```

STEP 7: Initialize the New Members

In Step 5, you added some new members to APP_DATA structure type; those members need to be initialized and some of them need to be initialized just once in the APP_Initialize() function.

Find the APP_Initialize() function in app.c file, and add the code to initialize the USB State Machine state member and the two buffer pointers; the state variable needs to be set to the initial state of the USB State Machine. The two pointers need to point to the corresponding buffers you declared in Step 6.

The other members will be initialized just before their use.

```
void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state = APP_STATE_INIT;

    /* Initialize USB HID Device application data */
    appData.handleUsbDevice       = USB_DEVICE_HANDLE_INVALID;
    appData.usbDeviceIsConfigured = false;
    appData.idleRate              = 0;

    /* Initialize USB Task State Machine appData members */
    appData.receiveDataBuffer = &receiveDataBuffer[0];
    appData.transmitDataBuffer = &transmitDataBuffer[0];
    appData.stateUSB = USB_STATE_INIT;

}
```

STEP 8: Handle the Detach

In the Harmony version we are using, USB_DEVICE_EVENT_DECONFIGURED and USB_DEVICE_EVENT_RESET events are not passed to the Application USB Device Event Handler Function. So the usbDeviceIsConfigured flag of appData structure needs to be set as false inside the USB_DEVICE_EVENT_POWER_REMOVED event.

Find the power removed case (USB_DEVICE_EVENT_POWER_REMOVED), in the APP_USBDeviceEventHandler() function, in app.c file and set the member usbDeviceIsConfigured of appData structure to false.

```
case USB_DEVICE_EVENT_POWER_REMOVED:
   /* VBUS is not available any more. Detach the device. */
    /* STEP 8: Handle the detach */
    USB_DEVICE_Detach(appData.handleUsbDevice);
    appData.usbDeviceIsConfigured = false;
    /* This is reached from Host to Device */
    break;
```

STEP 9: Handle the HID Events

The two flags you declared in Step 5 will be used by the USB State Machine to check the status of the previous report receive or transmit transaction. The status of those two flags need to be updated when the two HID events (report sent and report received) are passed to the Application HID Event Handler Function. You need to be sure that the event is related to the request you made and, for this purpose, you can compare the transfer handle of the request with the transfer handle available in the event: if they match, the event relates to the ongoing request.

Find the APP_USBDeviceHIDEventHandler() function in app.c file, add a local variable to cast the eventData parameter and update the two flags, one in the report received event, one in the report sent event; don't forget to check if the transfer handles are matching before setting the flag to true. To match the transfer handle you need to cast the eventData parameter to the USB Device HID Report Event Data Type; there are two events and two types, one for report received and one for report sent.

```
static void APP_USBDeviceHIDEventHandler
(
    USB_DEVICE_HID_INDEX hidInstance,
    USB_DEVICE_HID_EVENT event,
    void * eventData,
    uintptr_t userData
)
{
    APP_DATA * appData = (APP_DATA *)userData;

    switch(event)
    {
        case USB_DEVICE_HID_EVENT_REPORT_SENT:
        {
            /* This means a Report has been sent.  We are free to send next
             * report. An application flag can be updated here. */
```

```
                /* Handle the HID Report Sent event */
            USB_DEVICE_HID_EVENT_DATA_REPORT_SENT * report =
                    (USB_DEVICE_HID_EVENT_DATA_REPORT_SENT *)eventData;
            if(report->handle == appData->txTransferHandle )
            {
                // Transfer progressed.
                appData->hidDataTransmitted = true;
            }
            break;
        }
        case USB_DEVICE_HID_EVENT_REPORT_RECEIVED:
        {
            /* This means Report has been received from the Host. Report
             * received can be over Interrupt OUT or Control endpoint based on
             * Interrupt OUT endpoint availability. An application flag can be
             * updated here. */

             /* Handle the HID Report Received event */
            USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED * report =
                    (USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED *)eventData;
            if(report->handle == appData->rxTransferHandle )
            {
                // Transfer progressed.
                appData->hidDataReceived = true;
            }
            break;
        }

    [...]

}
```

STEP 10: Create the USB State Machine

The Basic HID Device template that was used to generate the code expects the USB State machine to be placed inside the USB_Task() function; that state machine will be executed until the usbDeviceIsConfigured member of appData structure, is true.

When the USB cable is unplugged, the state machine is no longer executed, but you need to reset it to the initial state to be ready for the next USB connection.

Find the if(appData.usbDeviceIsConfigured) statement of USB_Task() function in app.c file and add the else statement to set the USB State Machine state member of the appData structure to its initial state (e.g. USB_STATE_INIT).

Inside the if statement of the USB_Task() function, you can place the requested state machine; you can create it using a switch statement with four cases, one for each state you declared in the enumeration type you defined in Step 4. Find the if(appData.usbDeviceIsConfigured) statement of the USB_Task() function and add a switch statement for the USB State Machine state member of the appData structure and a case for each entry of the enumeration type of that state member.

In the initialization state of the switch statement add the code to set the transmission flag to true and the two transfer handles to invalid (USB_DEVICE_HID_TRANSFER_HANDLE_INVALID), set the USB State Machine state member of appData structure to the state that schedules a receive request (e.g. USB_STATE_SCHEDULE_READ).

```
static void USB_Task (void)
{
    if(appData.usbDeviceIsConfigured)
    {
        /* Write USB HID Application Logic here. Note that this function is
         * being called periodically the APP_Tasks() function. The application
         * logic should be implemented as state machine. It should not block */

        switch (appData.stateUSB)
        {
            case USB_STATE_INIT:

                appData.hidDataTransmitted = true;
                appData.txTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
                appData.rxTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
                appData.stateUSB = USB_STATE_SCHEDULE_READ;

                break;

            case USB_STATE_SCHEDULE_READ:

                appData.hidDataReceived = false;
                USB_DEVICE_HID_ReportReceive (USB_DEVICE_HID_INDEX_0,
                    &appData.rxTransferHandle, appData.receiveDataBuffer, 64 );
                appData.stateUSB = USB_STATE_WAITING_FOR_DATA;

                break;

            case USB_STATE_WAITING_FOR_DATA:

                if( appData.hidDataReceived )
                {
                    if (appData.receiveDataBuffer[0]==0x01)
                    {
                        BSP_LED_1Toggle();
```

```
                    appData.stateUSB = USB_STATE_SCHEDULE_READ;
                }
                else if (appData.receiveDataBuffer[0]==0x02)
                {
                    BSP_LED_2Toggle();
                    appData.stateUSB = USB_STATE_SCHEDULE_READ;
                }
                else if (appData.receiveDataBuffer[0]==0x03)
                {
                    BSP_LED_3Toggle();
                    appData.stateUSB = USB_STATE_SCHEDULE_READ;
                }
                else if (appData.receiveDataBuffer[0]==0x00)
                {
                    appData.stateUSB = USB_STATE_SEND_REPORT;
                }
                else
                {
                    appData.stateUSB = USB_STATE_SCHEDULE_READ;
                }
            }

            break;

        case USB_STATE_SEND_REPORT:

            if(appData.hidDataTransmitted)
            {
                if( BSP_SwitchStateGet(BSP_SWITCH_1) ==
                      BSP_SWITCH_STATE_PRESSED )
                {
                    appData.transmitDataBuffer[0] = 0x00;
                }
                else
                {
                    appData.transmitDataBuffer[0] = 0x01;
                }

                appData.hidDataTransmitted = false;
                USB_DEVICE_HID_ReportSend (USB_DEVICE_HID_INDEX_0,
                    &appData.txTransferHandle, appData.transmitDataBuffer, 1);
                appData.stateUSB = USB_STATE_SCHEDULE_READ;
            }

            break;
    }
}
else
```

```
        {

            /* Reset the USB Task State Machine */
            appData.stateUSB = USB_STATE_INIT;

        }
    }
```

STEP 11: Schedule a New Report Receive Request

To receive a report from the USB host, you need to schedule a report receive request by using the API provided for the USB HID Function Driver.

Before scheduling the request, the reception flag needs to be set to `false` to check when the request is completed (you set it to `true` in the report received complete event in Step 9).

After scheduling the request, the USB State Machine state needs to be moved to the waiting for data state.

Inside the schedule read state of the switch statement of the `USB_Task()` function, add the code to set the reception flag to `false`, then schedule a new report receive request and finally set the USB State Machine state member of `appData` structure to the state that waits for data from the USB host (e.g. USB_STATE_WAITING_FOR_DATA).

```
    case USB_STATE_SCHEDULE_READ:

        appData.hidDataReceived = false;
         USB_DEVICE_HID_ReportReceive (USB_DEVICE_HID_INDEX_0,
                         &appData.rxTransferHandle, appData.receiveDataBuffer, 64 );
         appData.stateUSB = USB_STATE_WAITING_FOR_DATA;

         break;
```

STEP 12: Receive, Prepare and Send Reports

When the report is received, the reception flag is set to `true`; that means there is valid data in the reception buffer. Inside the switch of the `USB_Task()` function the state is set to USB_STATE_WAITING_FOR_DATA and are checked the next commands that are sent by the Linux USB host driver:

- **0x01**: Toggle the LED1. The state is set to USB_STATE_SCHEDULE_READ.
- **0x02**: Toggle the LED2. The state is set to USB_STATE_SCHEDULE_READ.
- **0x03**: Toggle the LED3. The state is set to USB_STATE_SCHEDULE_READ.
- **0x00**: The USB device gets the Pushbutton state. The state is set to USB_STATE_SEND_REPORT. The HID device replies with a report to the host, where the first byte is the status of the S1 button ("0x00" pressed, "0x01" not pressed).

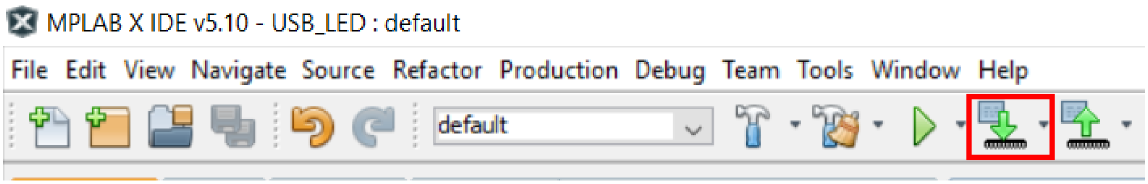
```
case USB_STATE_WAITING_FOR_DATA:

   if( appData.hidDataReceived )
   {
          if (appData.receiveDataBuffer[0]==0x01)
          {
                 BSP_LED_1Toggle();
                 appData.stateUSB = USB_STATE_SCHEDULE_READ;
          }
          else if (appData.receiveDataBuffer[0]==0x02)
          {
                 BSP_LED_2Toggle();
                 appData.stateUSB = USB_STATE_SCHEDULE_READ;
          }
          else if (appData.receiveDataBuffer[0]==0x03)
          {
                 BSP_LED_3Toggle();
                 appData.stateUSB = USB_STATE_SCHEDULE_READ;
          }
          else if (appData.receiveDataBuffer[0]==0x00)
          {
                 appData.stateUSB = USB_STATE_SEND_REPORT;
          }
          else
          {
                 appData.stateUSB = USB_STATE_SCHEDULE_READ;
          }
     }

     break;

 case USB_STATE_SEND_REPORT:

     if(appData.hidDataTransmitted)
     {
          if( BSP_SwitchStateGet(BSP_SWITCH_1) == BSP_SWITCH_STATE_PRESSED )
          {
                appData.transmitDataBuffer[0] = 0x00;
          }
          else
          {
                appData.transmitDataBuffer[0] = 0x01;
          }

             appData.hidDataTransmitted = false;
             USB_DEVICE_HID_ReportSend (USB_DEVICE_HID_INDEX_0,
                       &appData.txTransferHandle, appData.transmitDataBuffer, 1);
             appData.stateUSB = USB_STATE_SCHEDULE_READ;
```

```
}
```

STEP 13: Program The Application

Power the PIC32MX470 Curiosity Development Board from a Host PC through a Type-A male to mini-B USB cable connected to Mini-B port (J3). Ensure that a jumper is placed in J8 header (between 4 & 3) to select supply from debug USB connector.

Build the code and program the device by clicking on the program button as shown below.



LAB 13.2: "USB LED" Module

In the previous lab, you developed the firmware for a fully functional USB HID device that is able to send and receive data by using HID reports. Now, you are going to develop a Linux USB host driver to control that USB device. The driver will send USB commands to toggle LED1, LED2 and LED3 of the PIC32MX470 Curiosity Development Board; it will receive the command from the Linux user space through a sysfs entry and then retransmit it to the PIC32MX HID device. The command values can be 0x01, 0x02, or 0x03. The HID device must toggle LED1 when it receives 0x01, LED2 when it receives 0x02 and LED3 when it receives 0x03 in the report .

You will use the Raspberry Pi 4 Model B board to implement this driver.

You have to stop hid-generic driver from getting control of our custom driver, so you will include our driver´s USB_VENDOR_ID and USB_DEVICE_ID in the list hid_ignore_list[]. Open hid-quirks.c file under /linux/drivers/hid folder in your host PC and add the next line of code (in bold) to the end of the list:

```
static const struct hid_device_id hid_ignore_list[] = {
...
#if IS_ENABLED(CONFIG_MOUSE_SYNAPTICS_USB)
    { HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_TP) },
    { HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_INT_TP) },
    { HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_CPAD) },
    { HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_STICK) },
    { HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_WP) },
    { HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_COMP_TP) },
    { HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_WTP) },
```

```
	{ HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_DPAD) },
#endif
	{ HID_USB_DEVICE(USB_VENDOR_ID_YEALINK, USB_DEVICE_ID_YEALINK_P1K_P4K_B2K) },
	{ HID_USB_DEVICE(0x04d8, 0x003f) },
	{ }
};
```

You have modified the kernel sources, so you have to compile the new kernel and send it to the Raspberry Pi 4:

```
~/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage
~/linux$ scp arch/arm/boot/zImage root@10.0.0.10:/boot/kernel7l.img
```

Reboot the system to launch the new kernel.

Connect the USB Micro-B port (J12) of the PIC32MX470 Curiosity Development Board to one of the four USB Host Type-A connectors on the Raspberry Pi 4 Model B board.

LAB 13.2 Code Description of the "USB LED" Module

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

1. Include the function headers:

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

2. Create the ID table to support hotplugging. The Vendor ID and Product ID values have to match with the ones used in the PIC32MX USB HID device.

   ```
   #define USBLED_VENDOR_ID    0x04D8
   #define USBLED_PRODUCT_ID   0x003F

   /* table of devices that work with this driver */
   static const struct usb_device_id id_table[] = {
   	{ USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
   	{ }
   };
   MODULE_DEVICE_TABLE(usb, id_table);
   ```

3. Create a private structure that will store the driver´s data.

   ```
   struct usb_led {
   	struct usb_device *udev;
   	u8 led_number;
   };
   ```

4. See below an extract of the `probe()` routine with the main lines of code to set up the driver commented.

```
static int led_probe(struct usb_interface *interface,
                  const struct usb_device_id *id)
{
      /* Get the usb_device structure from the usb_interface one */
      struct usb_device *udev = interface_to_usbdev(interface);
      struct usb_led *dev = NULL;
      int retval = -ENOMEM;

      dev_info(&interface->dev, "led_probe() function is called.\n");

      /* Allocate our private data structure */
      dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);

      /* store the usb device in our data structure */
      dev->udev = usb_get_dev(udev);

      /* Attach the USB device data to the USB interface */
      usb_set_intfdata(interface, dev);

      /* create a led sysfs entry to interact with the user space */
      device_create_file(&interface->dev, &dev_attr_led);

      return 0;

}
```

5. Write the led_store() function. Every time your user space application writes to the led sysfs entry (/sys/bus/usb/devices/1-1.3:1.0/led) under the USB device, the driver´s led_store() function is called. The usb_led structure associated with the USB device is recovered by using the usb_get_intfdata() function. The command written in the led sysfs entry is stored in the val variable. Finally, you will send the command value via USB by using the usb_bulk_msg() function.

   The kernel provides two usb_bulk_msg() and usb_control_msg() helper functions that make it possible to transfer simple bulk and control messages without having to create an urb structure, initialize it, submit it and wait for its completion handler. These functions are synchronous and will make your code sleep. You must not call them from interrupt context or with a spinlock held.

   ```
   int usb_bulk_msg(struct usb_device * usb_dev, unsigned int pipe, void * data,
   int len, int * actual_length, int timeout);
   ```

   See below a short description of the usb_bulk_msg() parameters:

   - **usb_dev**: pointer to the usb device to send the message to
   - **pipe**: endpoint "pipe" to send the message to

- **data**: pointer to the data to send
- **len**: length in bytes of the data to send
- **actual_length**: pointer to a location to put the actual length transferred in bytes
- **timeout**: time in msecs to wait for the message to complete before timing out

See below an extract of the led_store() routine:

```
static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                     const char *buf, size_t count)
{
      struct usb_interface *intf = to_usb_interface(dev);
      struct usb_led *led = usb_get_intfdata(intf);
      u8 val;

      /* transform char array to u8 value */
      kstrtou8(buf, 10, &val);

      led->led_number = val;

      /* Toggle led */
      usb_bulk_msg(led->udev, usb_sndctrlpipe(led->udev, 1),
                   &led->led_number,
                   1,
                   NULL,
                   0);

      return count;
}
static DEVICE_ATTR_RW(led);
```

6. Add a usb_driver structure that will be registered to the USB core:

```
static struct usb_driver led_driver = {
      .name =        "usbled",
      .probe =       led_probe,
      .disconnect =  led_disconnect,
      .id_table =    id_table,
};
```

7. Register your driver with the USB bus:

```
module_usb_driver(led_driver);
```

8. Build the module and load it to the target processor. Download the linux_5.4_rpi4_drivers.zip file from the github of the book and unzip it in the home folder of your Linux host:

```
~/linux_5.4_rpi4_drivers$ cd Chapter13_USB_drivers
```

Compile and deploy the drivers to the **Raspberry Pi 4 Model B** board:

~/linux_5.4_rpi4_drivers/Chapter13_USB_drivers$ make

~/linux_5.4_rpi4_drivers/Chapter13_USB_drivers$ make deploy

See in the next **Listing 13-1** the "USB LED" driver source code (usb_led.c).

Listing 13-1: usb_led.c

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

#define USBLED_VENDOR_ID 0x04D8
#define USBLED_PRODUCT_ID0x003F

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
   { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
   { }
};
MODULE_DEVICE_TABLE(usb, id_table);

struct usb_led {
   struct usb_device *udev;
   u8 led_number;
};

static ssize_t led_show(struct device *dev, struct device_attribute *attr,
                    char *buf)
{
   struct usb_interface *intf = to_usb_interface(dev);
   struct usb_led *led = usb_get_intfdata(intf);

   return sprintf(buf, "%d\n", led->led_number);
}

static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                   const char *buf, size_t count)
{
   struct usb_interface *intf = to_usb_interface(dev);
   struct usb_led *led = usb_get_intfdata(intf);
   u8 val;
   int error, retval;
   dev_info(&intf->dev, "led_store() function is called.\n");

   /* transform char array to u8 value */
   error = kstrtou8(buf, 10, &val);
```

```
    if (error)
            return error;

    led->led_number = val;

    if (val == 1 || val == 2 || val == 3)
            dev_info(&led->udev->dev, "led = %d\n", led->led_number);
    else {
            dev_info(&led->udev->dev, "unknown led %d\n", led->led_number);
            retval = -EINVAL;
            return retval;
    }

    /* Toggle led */
    retval = usb_bulk_msg(led->udev, usb_sndctrlpipe(led->udev, 1),
                          &led->led_number,
                          1,
                          NULL,
                          0);
    if (retval) {
            retval = -EFAULT;
            return retval;
    }
    return count;
}
static DEVICE_ATTR_RW(led);

static int led_probe(struct usb_interface *interface,
               const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(interface);
    struct usb_led *dev = NULL;
    int retval = -ENOMEM;

    dev_info(&interface->dev, "led_probe() function is called.\n");

    dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);
    if (!dev) {
            dev_err(&interface->dev, "out of memory\n");
            retval = -ENOMEM;
            goto error;
    }

    dev->udev = usb_get_dev(udev);

    usb_set_intfdata(interface, dev);

    retval = device_create_file(&interface->dev, &dev_attr_led);
```

```
	if (retval)
		goto error_create_file;

	return 0;

error_create_file:
	usb_put_dev(udev);
	usb_set_intfdata(interface, NULL);
error:
	kfree(dev);
	return retval;
}

static void led_disconnect(struct usb_interface *interface)
{
	struct usb_led *dev;

	dev = usb_get_intfdata(interface);

	device_remove_file(&interface->dev, &dev_attr_led);
	usb_set_intfdata(interface, NULL);
	usb_put_dev(dev->udev);
	kfree(dev);

	dev_info(&interface->dev, "USB LED now disconnected\n");
}

static struct usb_driver led_driver = {
	.name =		"usbled",
	.probe =	led_probe,
	.disconnect =	led_disconnect,
	.id_table =	id_table,
};

module_usb_driver(led_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a synchronous led usb controlled module");
```

usb_led.ko Demonstration

```
/*
 * Connect the PIC32MX470 Curiosity Development Board USB Micro-B port (J12) to
 * one of the four USB HostType-A connectors of the Raspberry Pi 4 Model B board.
 * Power the Raspberry Pi 4 Model B board to boot the processor. Keep the
 * PIC32MX470 board powered off
 */
```

```
root@raspberrypi:/home# insmod usb_led.ko /* load the module */
usb_led: loading out-of-tree module taints kernel.
usbcore: registered new interface driver usbled

/* power now the PIC32MX Curiosity board */
root@raspberrypi:/home# usb 1-1.3: new full-speed USB device number 5 using
dwc_otg
usb 1-1.3: New USB device found, idVendor=04d8, idProduct=003f, bcdDevice= 1.00
usb 1-1.3: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-1.3: Product: LED_USB HID Demo
usb 1-1.3: Manufacturer: Microchip Technology Inc.
usbled 1-1.3:1.0: led_probe() function is called.

/* check the new created USB device */
root@raspberrypi:/home# cd /sys/bus/usb/devices/1-1.3:1.0
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# ls
authorized         bInterfaceProtocol  ep_01     power
bAlternateSetting  bInterfaceSubClass  ep_81     subsystem
bInterfaceClass    bNumEndpoints       led       supports_autosuspend
bInterfaceNumber   driver              modalias  uevent

/* Read the configurations of the USB device */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# cat bNumEndpoints
02
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0/ep_01# cat direction
out
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0/ep_81# cat direction
in
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# cat bAlternateSetting
 0
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# cat bInterfaceClass
03
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# cat bNumEndpoints
02

/* Switch on the LED1 of the PIC32MX Curiosity board */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 1 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 1

/* Switch on the LED2 of the PIC32MX Curiosity board */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 2 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 2

/* Switch on the LED3 of the PIC32MX Curiosity board */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 3 > led
usbled 1-1.3:1.0: led_store() function is called.
```

```
usb 1-1.3: led = 3

/* read the led status */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# cat led
3

root@raspberrypi:/home# rmmod usb_led.ko  /* remove the module */
usbcore: deregistering interface driver usbled
usbled 1-1.3:1.0: USB LED now disconnected
```

LAB 13.3: "USB LED and Switch" Module

In this new lab, you will increase the functionality of the previous driver. Besides controlling three LEDs connected to the USB device, the Linux host driver will receive a Pushbutton (S1 switch on the PIC32MX470 Curiosity Development Board) state from the USB HID device. The driver will send a command to the USB device with value 0x00, then the HID device will reply with a report, where the first byte is the status of the S1 button ("0x00" pressed, "0x01" not pressed). In this driver, unlike the previous one, the communication between the host and the device is done asynchronously by using USB Request Blocks (urbs).

LAB 13.3 Code Description of the "USB LED and Switch" Module

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

1. Include the function headers:

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

2. Create the ID table to support hotplugging. The Vendor ID and Product ID values have to match with the ones used in the PIC32MX USB HID device.

   ```
   #define USBLED_VENDOR_ID    0x04D8
   #define USBLED_PRODUCT_ID   0x003F

   /* table of devices that work with this driver */
   static const struct usb_device_id id_table[] = {
         { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
         { }
   };
   MODULE_DEVICE_TABLE(usb, id_table);
   ```

3. Create a private structure that will store the driver´s data.

   ```
   struct usb_led {
         struct usb_device     *udev;
         struct usb_interface  *intf;
   ```

```
	struct urb		*interrupt_out_urb;
	struct urb		*interrupt_in_urb;
	struct usb_endpoint_descriptor *interrupt_out_endpoint;
	struct usb_endpoint_descriptor *interrupt_in_endpoint;
	u8		irq_data;
	u8		led_number;
	u8		ibuffer;
	int		interrupt_out_interval;
	int		ep_in;
	int		ep_out;
};
```

4. See below an extract of the probe() routine with the main lines of code to configure the driver commented.

```
static int led_probe(struct usb_interface *intf,
		     const struct usb_device_id *id)
{
	struct usb_device *udev = interface_to_usbdev(intf);

	/* Get the current altsetting of the USB interface */
	struct usb_host_interface *altsetting = intf->cur_altsetting;
	struct usb_endpoint_descriptor *endpoint;
	struct usb_led *dev = NULL;
	int ep;
	int ep_in, ep_out;
	int size;

	/*
	 * Find the last interrupt out endpoint descriptor
	 * to check its number and its size
	 * Just for teaching purposes
	 */
	usb_find_last_int_out_endpoint(altsetting, &endpoint);

	/* get the endpoint's number */
	ep = usb_endpoint_num(endpoint); /* value from 0 to 15, it is 1 */
	size = usb_endpoint_maxp(endpoint);

	/* Validate endpoint and size */
	if (size <= 0) {
		dev_info(&intf->dev, "invalid size (%d)", size);
		return -ENODEV;
	}

	dev_info(&intf->dev, "endpoint size is (%d)", size);
	dev_info(&intf->dev, "endpoint number is (%d)", ep);

	/* Get the two addresses (IN and OUT) of the Endpoint 1 */
```

```
	ep_in = altsetting->endpoint[0].desc.bEndpointAddress;
	ep_out = altsetting->endpoint[1].desc.bEndpointAddress;

	/* Allocate our private data structure */
	dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);

	/* Store values in the data structure */
	dev->ep_in = ep_in;
	dev->ep_out = ep_out;
	dev->udev = usb_get_dev(udev);
	dev->intf = intf;

	/* allocate the int_out_urb structure */
	dev->interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);

	/* initialize the int_out_urb */
	usb_fill_int_urb(dev->interrupt_out_urb,
			dev->udev,
			usb_sndintpipe(dev->udev, ep_out),
			(void *)&dev->irq_data,
			1,
			led_urb_out_callback, dev, 1);

	/* allocate the int_in_urb structure */
	dev->interrupt_in_urb = usb_alloc_urb(0, GFP_KERNEL);
	if (!dev->interrupt_in_urb)
		goto error_out;

	/* initialize the int_in_urb */
	usb_fill_int_urb(dev->interrupt_in_urb,
			dev->udev,
			usb_rcvintpipe(dev->udev, ep_in),
			(void *)&dev->ibuffer,
			1,
			led_urb_in_callback, dev, 1);

	/* Attach the device data to the interface */
	usb_set_intfdata(intf, dev);

	/* create the led sysfs entry to interact with the user space */
	device_create_file(&intf->dev, &dev_attr_led);

	/* Submit the interrrupt IN URB */
	usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);

	return 0;
}
```

5. Write the led_store() function. Every time your user space application writes to the led sysfs entry (/sys/bus/usb/devices/1-1.3:1.0/led) under the USB device, the driver´s led_store() function is called. The usb_led structure associated with the USB device is recovered by using the usb_get_intfdata() function. The command written in the led sysfs entry is stored in the irq_data variable. Finally, you will send the command value via USB by using the usb_submit_urb() function.

   See below an extract of the led_store() routine:

```
static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                     const char *buf, size_t count)
{
      struct usb_interface *intf = to_usb_interface(dev);
      struct usb_led *led = usb_get_intfdata(intf);
      u8 val;

      /* transform char array to u8 value */
      kstrtou8(buf, 10, &val);

      led->irq_data = val;

      /* send the data out */
      retval = usb_submit_urb(led->interrupt_out_urb, GFP_KERNEL);

      return count;
}
static DEVICE_ATTR_RW(led);
```

6. Create OUT and IN URB's completion callbacks. The interrupt OUT completion callback merely checks the URB status and returns. The interrupt IN completion callback checks the URB status, then reads the ibuffer to know the status received from the PIC32MX board´s S1 switch, and finally re-submits the interrupt IN URB.

```
static void led_urb_out_callback(struct urb *urb)
{
      struct usb_led *dev;

      dev = urb->context;

      /* sync/async unlink faults aren't errors */
      if (urb->status) {
             if (!(urb->status == -ENOENT ||
                 urb->status == -ECONNRESET ||
                 urb->status == -ESHUTDOWN))
                    dev_err(&dev->udev->dev,
                           "%s - nonzero write status received: %d\n",
                           __func__, urb->status);
```

```
        }
}

static void led_urb_in_callback(struct urb *urb)
{
        int retval;
        struct usb_led *dev;

        dev = urb->context;

        if (urb->status) {
               if (!(urb->status == -ENOENT ||
                   urb->status == -ECONNRESET ||
                   urb->status == -ESHUTDOWN))
                      dev_err(&dev->udev->dev,
                             "%s - nonzero write status received: %d\n",
                             __func__, urb->status);
        }

        if (dev->ibuffer == 0x00)
               pr_info ("switch is ON.\n");
        else if (dev->ibuffer == 0x01)
               pr_info ("switch is OFF.\n");
        else
               pr_info ("bad value received\n");

        usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);
}
```

7. Add a struct usb_driver structure that will be registered to the USB core:

```
static struct usb_driver led_driver = {
        .name =        "usbled",
        .probe =       led_probe,
        .disconnect =  led_disconnect,
        .id_table =    id_table,
};
```

8. Register your driver with the USB bus:

```
module_usb_driver(led_driver);
```

9. Build the module and load it to the target processor:

See in the next **Listing 13-2** the "USB LED and Switch" driver source code (usb_urb_int_led.c).

Listing 13-2: usb_urb_int_led.c

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

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

#define USBLED_VENDOR_ID 0x04D8
#define USBLED_PRODUCT_ID0x003F

static void led_urb_out_callback(struct urb *urb);
static void led_urb_in_callback(struct urb *urb);

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
   { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
   { }
};
MODULE_DEVICE_TABLE(usb, id_table);

struct usb_led {
   struct usb_device *udev;
   struct usb_interface *intf;
   struct urb      *interrupt_out_urb;
   struct urb      *interrupt_in_urb;
   struct usb_endpoint_descriptor *interrupt_out_endpoint;
   struct usb_endpoint_descriptor *interrupt_in_endpoint;
   u8              irq_data;
   u8              led_number;
   u8              ibuffer;
   int             interrupt_out_interval;
   int ep_in;
   int ep_out;
};

static ssize_t led_show(struct device *dev, struct device_attribute *attr,
                   char *buf)
{
   struct usb_interface *intf = to_usb_interface(dev);
   struct usb_led *led = usb_get_intfdata(intf);

   return sprintf(buf, "%d\n", led->led_number);
}

static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                  const char *buf, size_t count)
{
   struct usb_interface *intf = to_usb_interface(dev);
   struct usb_led *led = usb_get_intfdata(intf);
   u8 val;
   int error, retval;
```

```
    dev_info(&intf->dev, "led_store() function is called.\n");

    /* transform char array to u8 value */
    error = kstrtou8(buf, 10, &val);
    if (error)
            return error;

    led->led_number = val;
    led->irq_data = val;

    if (val == 0)
            dev_info(&led->udev->dev, "read status\n");
    else if (val == 1 || val == 2 || val == 3)
            dev_info(&led->udev->dev, "led = %d\n", led->led_number);
     else {
            dev_info(&led->udev->dev, "unknown value %d\n", val);
            retval = -EINVAL;
            return retval;
     }

    /* send the data out */
    retval = usb_submit_urb(led->interrupt_out_urb, GFP_KERNEL);
    if (retval) {
            dev_err(&led->udev->dev,
                    "Couldn't submit interrupt_out_urb %d\n", retval);
            return retval;
    }

    return count;
}
static DEVICE_ATTR_RW(led);

static void led_urb_out_callback(struct urb *urb)
{
    struct usb_led *dev;

    dev = urb->context;

    dev_info(&dev->udev->dev, "led_urb_out_callback() function is called.\n");

    /* sync/async unlink faults aren't errors */
    if (urb->status) {
            if (!(urb->status == -ENOENT ||
                urb->status == -ECONNRESET ||
                urb->status == -ESHUTDOWN))
                    dev_err(&dev->udev->dev,
                            "%s - nonzero write status received: %d\n",
                            __func__, urb->status);
```

```
    }
}

static void led_urb_in_callback(struct urb *urb)
{
    int retval;
    struct usb_led *dev;

    dev = urb->context;

    dev_info(&dev->udev->dev, "led_urb_in_callback() function is called.\n");

    if (urb->status) {
            if (!(urb->status == -ENOENT ||
                urb->status == -ECONNRESET ||
                urb->status == -ESHUTDOWN))
                    dev_err(&dev->udev->dev,
                            "%s - nonzero write status received: %d\n",
                            __func__, urb->status);
    }

    if (dev->ibuffer == 0x00)
            pr_info ("switch is ON.\n");
    else if (dev->ibuffer == 0x01)
            pr_info ("switch is OFF.\n");
    else
            pr_info ("bad value received\n");

    retval = usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);
    if (retval)
            dev_err(&dev->udev->dev,
                    "Couldn't submit interrupt_in_urb %d\n", retval);
}

static int led_probe(struct usb_interface *intf,
                const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(intf);
    struct usb_host_interface *altsetting = intf->cur_altsetting;
    struct usb_endpoint_descriptor *endpoint;
    struct usb_led *dev = NULL;
    int ep;
    int ep_in, ep_out;
    int retval, size, res;

    dev_info(&intf->dev, "led_probe() function is called.\n");

    res = usb_find_last_int_out_endpoint(altsetting, &endpoint);
```

```
if (res) {
	dev_info(&intf->dev, "no endpoint found");
	return res;
}

ep = usb_endpoint_num(endpoint); /* value from 0 to 15, it is 1 */
size = usb_endpoint_maxp(endpoint);

/* Validate endpoint and size */
if (size <= 0) {
	dev_info(&intf->dev, "invalid size (%d)", size);
	return -ENODEV;
}

dev_info(&intf->dev, "endpoint size is (%d)", size);
dev_info(&intf->dev, "endpoint number is (%d)", ep);

ep_in = altsetting->endpoint[0].desc.bEndpointAddress;
ep_out = altsetting->endpoint[1].desc.bEndpointAddress;

dev_info(&intf->dev, "endpoint in address is (%d)", ep_in);
dev_info(&intf->dev, "endpoint out address is (%d)", ep_out);

dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);

if (!dev)
	return -ENOMEM;

dev->ep_in = ep_in;
dev->ep_out = ep_out;

dev->udev = usb_get_dev(udev);

dev->intf = intf;

/* allocate int_out_urb structure */
dev->interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);
if (!dev->interrupt_out_urb)
	goto error_out;

/* initialize int_out_urb */
usb_fill_int_urb(dev->interrupt_out_urb,
		dev->udev,
		usb_sndintpipe(dev->udev, ep_out),
		(void *)&dev->irq_data,
		1,
		led_urb_out_callback, dev, 1);
```

```
	/* allocate int_in_urb structure */
	dev->interrupt_in_urb = usb_alloc_urb(0, GFP_KERNEL);
	if (!dev->interrupt_in_urb)
		goto error_out;

	/* initialize int_in_urb */
	usb_fill_int_urb(dev->interrupt_in_urb,
			dev->udev,
			usb_rcvintpipe(dev->udev, ep_in),
			(void *)&dev->ibuffer,
			1,
			led_urb_in_callback, dev, 1);

	usb_set_intfdata(intf, dev);

	retval = device_create_file(&intf->dev, &dev_attr_led);
	if (retval)
		goto error_create_file;

	retval = usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);
	if (retval) {
		dev_err(&dev->udev->dev,
			"Couldn't submit interrupt_in_urb %d\n", retval);
		device_remove_file(&intf->dev, &dev_attr_led);
		goto error_create_file;
	}

	dev_info(&dev->udev->dev,"int_in_urb submitted\n");

	return 0;

error_create_file:
	usb_free_urb(dev->interrupt_out_urb);
	usb_free_urb(dev->interrupt_in_urb);
	usb_put_dev(udev);
	usb_set_intfdata(intf, NULL);

error_out:
	kfree(dev);
	return retval;
}

static void led_disconnect(struct usb_interface *interface)
{
	struct usb_led *dev;

	dev = usb_get_intfdata(interface);
```

```
    device_remove_file(&interface->dev, &dev_attr_led);
    usb_free_urb(dev->interrupt_out_urb);
    usb_free_urb(dev->interrupt_in_urb);
    usb_set_intfdata(interface, NULL);
    usb_put_dev(dev->udev);
    kfree(dev);

    dev_info(&interface->dev, "USB LED now disconnected\n");
}

static struct usb_driver led_driver = {
    .name =         "usbled",
    .probe =        led_probe,
    .disconnect =   led_disconnect,
    .id_table =     id_table,
};

module_usb_driver(led_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a led/switch usb controlled module with irq in/out
endpoints");
```

usb_urb_int_led.ko Demonstration

```
/*
 * Connect the PIC32MX470 Curiosity Development Board USB Micro-B port (J12) to
 * one of the four USB HostType-A connectors of the Raspberry Pi 4 Model B board.
 * Power the Raspberry Pi 4 Model B board to boot the processor. Keep the
 * PIC32MX470 board powered off
 */
```

```
root@raspberrypi:/home# insmod usb_urb_int_led.ko /* load the module */
```

```
usb_urb_int_led: loading out-of-tree module taints kernel.
usbcore: registered new interface driver usbled

/* power now the PIC32MX Curiosity board */
root@raspberrypi:/home# usb 1-1.3: new full-speed USB device number 4 using
dwc_otg
usb 1-1.3: New USB device found, idVendor=04d8, idProduct=003f, bcdDevice= 1.00
usb 1-1.3: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-1.3: Product: LED_USB HID Demo
usb 1-1.3: Manufacturer: Microchip Technology Inc.
usbled 1-1.3:1.0: led_probe() function is called.
usbled 1-1.3:1.0: endpoint size is (64)
usbled 1-1.3:1.0: endpoint number is (1)
usbled 1-1.3:1.0: endpoint in address is (129)
```

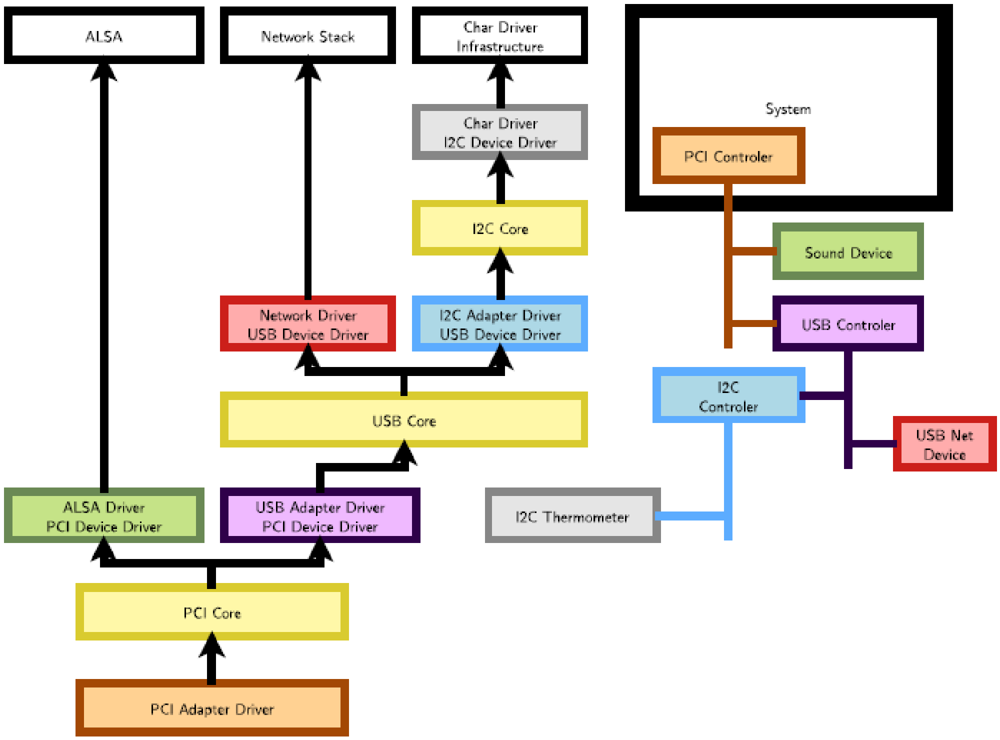
```
usbled 1-1.3:1.0: endpoint out address is (1)
usb 1-1.3: int_in_urb submitted

/* Go to the new created USB device */
root@raspberrypi:/home# cd /sys/bus/usb/devices/1-1.3:1.0

/* Switch on the LED1 of the PIC32MX Curiosity board */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 1 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 1
usb 1-1.3: led_urb_out_callback() function is called.

/* Switch on the LED2 of the PIC32MX Curiosity board */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 2 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 2
usb 1-1.3: led_urb_out_callback() function is called.

/* Switch on the LED3 of the PIC32MX Curiosity board */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 3 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 3
usb 1-1.3: led_urb_out_callback() function is called.

/* Keep pressed the S1 switch of PIC32MX Curiosity board and get SW status*/
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 0 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: read status
usb 1-1.3: led_urb_out_callback() function is called.
usb 1-1.3: led_urb_in_callback() function is called.
switch is ON.

/* Release the S1 switch of PIC32MX Curiosity board and get SW status */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 0 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: read status
usb 1-1.3: led_urb_out_callback() function is called.
usb 1-1.3: led_urb_in_callback() function is called.
switch is OFF.

root@raspberrypi:/home# rmmod usb_urb_int_led.ko /* remove the module */
usbcore: deregistering interface driver usbled
usb 1-1.3: led_urb_in_callback() function is called.
switch is OFF.
usb 1-1.3: Couldn't submit interrupt_in_urb -1
usbled 1-1.3:1.0: USB LED now disconnected
```

LAB 13.4: "I2C to USB Multidisplay LED" Module

In the lab 6.2 of this book, you implemented 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). In that lab 6.2, you controlled the LTC3206 device by using an I2C Linux driver. In this lab 13.4, you will write a Linux USB driver that is controlled from the user space by using the I2C Tools for Linux; to perform this task you will have to create a new I2C adapter within your created USB driver.

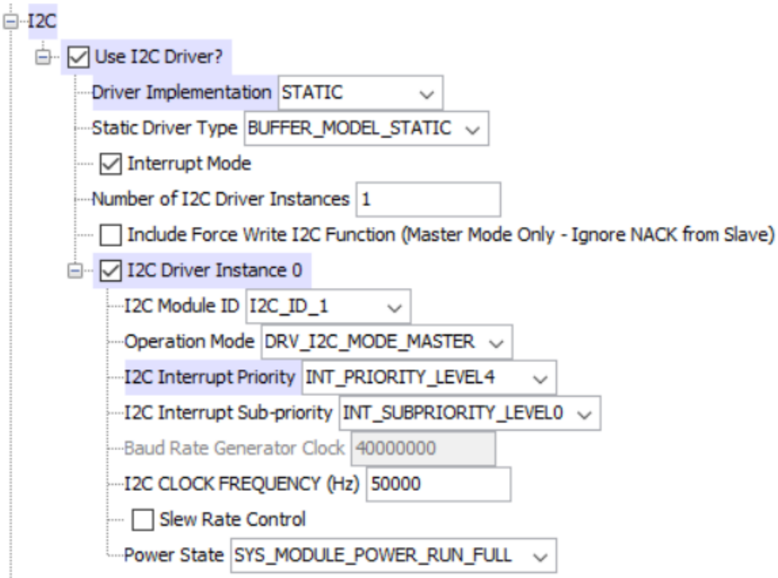
The driver model is recursive. In the following image, you can see all the needed drivers to control an I2C device through a PCI board that integrates a USB to I2C converter. These are the main steps to create this recursive driver model:

- First, you have to develop a PCI device driver that will create a USB adapter (the PCI device driver is the parent of the USB adapter driver).
- Second, you have to develop a USB device driver that will send USB data to the USB adapter driver through the USB Core; this USB device driver will also create an I2C adapter driver (the USB device driver is the parent of the I2C adapter driver).
- Finally, you will create an I2C device driver that will send data to the I2C adapter driver through the I2C Core and will create a `struct file_operations` structure to define driver´s functions which are called when the Linux user space reads, and writes to character devices.

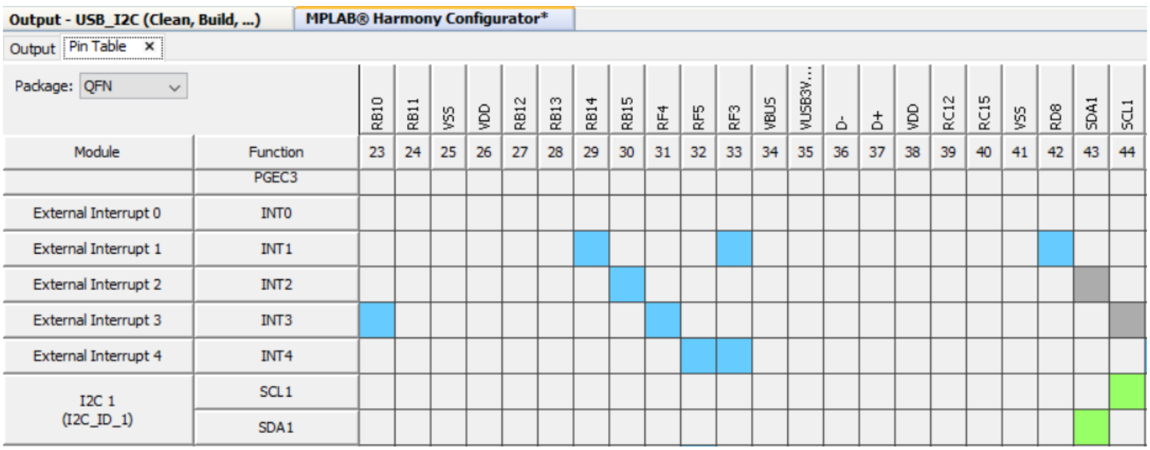


This recursive model will be simplified in the driver of lab 13.4, where you are only going to execute the second step of the three previously mentioned. In this driver, the communication between the host and the device is done asynchronously by using an interrupt OUT URB.

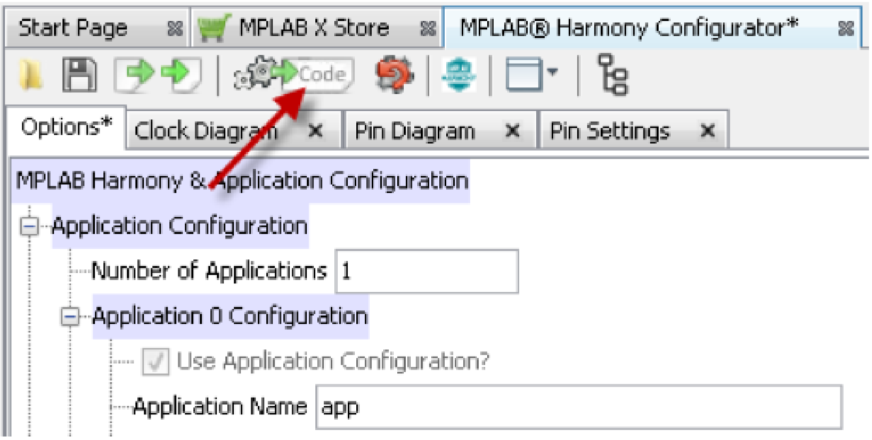
Before developing the Linux driver, you must first add new Harmony configurations to the previous project ones. You must select the I2C Drivers option inside the Harmony Framework Configuration:



In the Pin Table of the MPLAB Harmony Configurator activate the SCL1 and SDA1 pins of the I2C1 controller:



Generate the code, save the modified configuration, and generate the project:



Now, you have to modify the generated app.c code. Go to the USB_STATE_WAITING_FOR_DATA case inside the USB_Task() function. Basically, it is waiting for I2C data, which has been encapsulated inside a USB interrupt OUT URB; once the PIC32MX USB device receives the information, it forwards it via I2C to the LTC3206 device connected to the MikroBus 1 of the PIC32MX470 Curiosity Development Board.

```
static void USB_Task (void)
{
    if(appData.usbDeviceIsConfigured)
    {
        switch (appData.stateUSB)
        {
            case USB_STATE_INIT:

                appData.hidDataTransmitted = true;
                appData.txTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
                appData.rxTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
                appData.stateUSB = USB_STATE_SCHEDULE_READ;

                break;

            case USB_STATE_SCHEDULE_READ:

                appData.hidDataReceived = false;

                /* receive from Host (OUT endpoint). It is a write
                 command to the LTC3206 device */
                USB_DEVICE_HID_ReportReceive (USB_DEVICE_HID_INDEX_0,
                    &appData.rxTransferHandle, appData.receiveDataBuffer, 64 );
                appData.stateUSB = USB_STATE_WAITING_FOR_DATA;
```

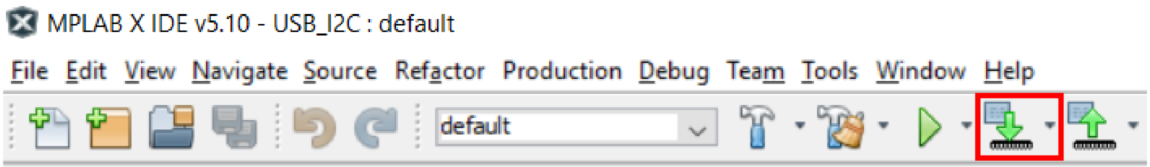
```
            break;

        case USB_STATE_WAITING_FOR_DATA:

            if( appData.hidDataReceived )
            {

                DRV_I2C_Transmit (appData.drvI2CHandle_Master,
                                  0x36,
                                  &appData.receiveDataBuffer[0],
                                  3,
                                  NULL);

                appData.stateUSB = USB_STATE_SCHEDULE_READ;
            }
            break;
    }
}
else
{

      appData.stateUSB = USB_STATE_INIT;

}
```

You also need to open the I2C driver inside the APP_Tasks() function:

```
    /* Application's initial state. */
    case APP_STATE_INIT:
    {
        bool appInitialized = true;

        /* Open the I2C Driver for Slave device */
        appData.drvI2CHandle_Master = DRV_I2C_Open(DRV_I2C_INDEX_0,
                                                   DRV_IO_INTENT_WRITE);
        if ( appData.drvI2CHandle_Master == (DRV_HANDLE)NULL )
        {
            appInitialized = false;
        }

        /* Open the device layer */
        if (appData.handleUsbDevice == USB_DEVICE_HANDLE_INVALID)
        {
            appData.handleUsbDevice = USB_DEVICE_Open( USB_DEVICE_INDEX_0,
                                                  DRV_IO_INTENT_READWRITE );
            if(appData.handleUsbDevice != USB_DEVICE_HANDLE_INVALID)
            {
```

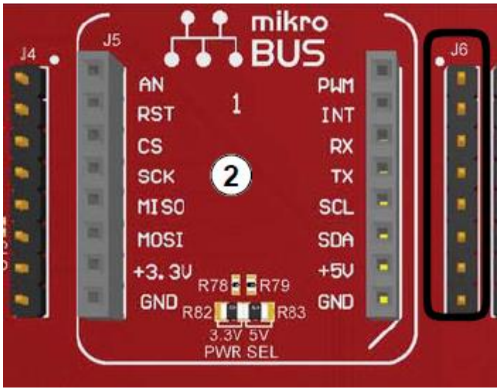
```
            appInitialized = true;
        }
        else
        {
            appInitialized = false;
        }
    }
```

Now, you must build the code and program the PIC32MX with the new application. You can download this new project from the book´s Github.



You will use the LTC3206 **DC749A - Demo Board** (http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/dc749a.html) to test the driver. You will connect the board to the MikroBUS 1 connector on the Curiosity PIC32MX470 Development Board. Connect the MikroBUS 1 SDA pin to the pin 7 (SDA) of the DC749A J1 connector and the MikroBUS 1 SCL pin to the pin 4 (SCL) of the DC749A J1 connector. Connect the MikroBUS 1 3.3V pin, the DC749A J20 DVCC pin and the DC749A pin 6 (ENRGB/S) to the DC749A Vin J2 pin. Do not forget to connect GND between the two boards.

Note: For the Curiosity PIC32MX470 Development Board, verify that the value of the series resistors mounted on the SCL and SDA lines of the mikroBUS 1 socket J5 are set to zero Ohms. If not, replace them with zero Ohm resistors. You can also take the SDA and SCL signals from the J6 connector if you do not want to replace the resistors.



LAB 13.4 Code Description of the "I2C to USB Multidisplay LED" Module

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

1. Include the function headers:

```
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/usb.h>
#include <linux/i2c.h>
```

2. Create the ID table to support hotplugging. The Vendor ID and Product ID values have to match with the ones used in the PIC32MX USB HID device.

```
#define USBLED_VENDOR_ID    0x04D8
#define USBLED_PRODUCT_ID   0x003F

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
      { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
      { }
};
MODULE_DEVICE_TABLE(usb, id_table);
```

3. Create a private structure that will store the driver´s data.

```
struct i2c_ltc3206 {
      u8 obuffer[LTC3206_OUTBUF_LEN];      /* USB write buffer */
      /* I2C/SMBus data buffer */
      u8 user_data_buffer[LTC3206_I2C_DATA_LEN]; /* LEN is 3 bytes */
      int ep_out;                        /* out endpoint */
      struct usb_device *usb_dev;   /* the usb device for this device */
```

```
        struct usb_interface *interface;/* the interface for this device */
        struct i2c_adapter adapter;  /* i2c related things */
        /* wq to wait for an ongoing write */
        wait_queue_head_t usb_urb_completion_wait;
        bool ongoing_usb_ll_op;             /* all is in progress */
        struct urb *interrupt_out_urb;      /* interrupt out URB */
};
```

4. See below an extract of the probe() routine with the main lines of code to configure the driver commented.

```
static int ltc3206_probe(struct usb_interface *interface,
                         const struct usb_device_id *id)
{
        /* Get the current altsetting of the USB interface */
        struct usb_host_interface *hostif = interface->cur_altsetting;
        struct i2c_ltc3206 *dev; /* the data structure */

        /* allocate data memory for our USB device and initialize it */
        kzalloc(sizeof(*dev), GFP_KERNEL);

        /* get interrupt ep_out address */
        dev->ep_out = hostif->endpoint[1].desc.bEndpointAddress;

        dev->usb_dev = usb_get_dev(interface_to_usbdev(interface));
        dev->interface = interface;

        /* declare dynamically a wait queue */
        init_waitqueue_head(&dev->usb_urb_completion_wait);

        /* save our data pointer in this USB interface device */
        usb_set_intfdata(interface, dev);

        /* setup I2C adapter description */
        dev->adapter.owner = THIS_MODULE;
        dev->adapter.class = I2C_CLASS_HWMON;
        dev->adapter.algo = &ltc3206_usb_algorithm;
        i2c_set_adapdata(&dev->adapter, dev);

        /* Attach the I2C adapter to the USB interface */
        dev->adapter.dev.parent = &dev->interface->dev;

        /* initialize the I2C device */
        ltc3206_init(dev);

        /* and finally attach the adapter to the I2C layer */
        i2c_add_adapter(&dev->adapter);

        return 0;
```

```
}
```

5. Write the ltc3206_init() function. Inside this function, you will allocate and initialize the interrupt OUT URB which is used for the communication between the host and the device. See below an extract of the ltc3206_init() routine:

```
static int ltc3206_init(struct i2c_ltc3206 *dev)
{
	/* allocate int_out_urb structure */
	interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);

	/* initialize int_out_urb structure */
	usb_fill_int_urb(dev->interrupt_out_urb, dev->usb_dev,
				usb_sndintpipe(dev->usb_dev,
						dev->ep_out),
				(void *)&dev->obuffer, LTC3206_OUTBUF_LEN,
				ltc3206_usb_cmpl_cbk, dev,
				1);

	return 0;
}
```

6. Create a struct i2c_algorithm that represents the I2C transfer method. You will initialize two variables inside this structure:

- **master_xfer**: Issues a set of i2c transactions to the given I2C adapter defined by the msgs array, with num messages available to transfer via the adapter specified by adap.
- **functionality**: Returns the flags that this algorithm/adapter pair supports from the I2C_FUNC_* flags.

```
static const struct i2c_algorithm ltc3206_usb_algorithm = {
	.master_xfer = ltc3206_usb_i2c_xfer,
	.functionality = ltc3206_usb_func,
};
```

7. Write the ltc3206_usb_i2c_xfer() function. This function will be called each time you write to the I2C adapter from the Linux user space. This function will call ltc32016_i2c_write() which stores the I2C data received from the Linux user space in the obuffer[] char array, then ltc32016_i2c_write() will call ltc3206_ll_cmd() which submits the interrupt OUT URB to the USB device and waits for the URB´s completion.

```
static int ltc3206_usb_i2c_xfer(struct i2c_adapter *adap,
		struct i2c_msg *msgs, int num)
{
	/* get the private data structure */
```

```
	struct i2c_ltc3206 *dev = i2c_get_adapdata(adap);
	struct i2c_msg *pmsg;
	int ret, count;

	pr_info("number of i2c msgs is = %d\n", num);

	for (count = 0; count < num; count++) {
		pmsg = &msgs[count];
		ret = ltc3206_i2c_write(dev, pmsg);
		if (ret < 0)
			goto abort;
	}

	/* if all the messages were transferred ok, return "num" */
	ret = num;
abort:
	return ret;
}

static int ltc3206_i2c_write(struct i2c_ltc3206 *dev,
				struct i2c_msg *pmsg)
{
	u8 ucXferLen;
	int rv;
	u8 *pSrc, *pDst;

	/* I2C write lenght */
	ucXferLen = (u8)pmsg->len;

	pSrc = &pmsg->buf[0];
	pDst = &dev->obuffer[0];
	memcpy(pDst, pSrc, ucXferLen);

	pr_info("oubuffer[0] = %d\n", dev->obuffer[0]);
	pr_info("oubuffer[1] = %d\n", dev->obuffer[1]);
	pr_info("oubuffer[2] = %d\n", dev->obuffer[2]);

	rv = ltc3206_ll_cmd(dev);
	if (rv < 0)
		return -EFAULT;

	return 0;
}

static int ltc3206_ll_cmd(struct i2c_ltc3206 *dev)
{
	int rv;

	/*
```

```
	 * tell everybody to leave the URB alone
	 * we are going to write to the LTC3206
	 */
	dev->ongoing_usb_ll_op = 1; /* doing USB communication */

	/* submit the interrupt out ep packet */
	if (usb_submit_urb(dev->interrupt_out_urb, GFP_KERNEL)) {
		dev_err(&dev->interface->dev,
				"ltc3206(ll): usb_submit_urb intr out failed\n");
		dev->ongoing_usb_ll_op = 0;
		return -EIO;
	}

	/* wait for its completion, the USB URB callback will signal it */
	rv = wait_event_interruptible(dev->usb_urb_completion_wait,
			(!dev->ongoing_usb_ll_op));
	if (rv < 0) {
		dev_err(&dev->interface->dev, "ltc3206(ll): wait
			interrupted\n");
		goto ll_exit_clear_flag;
	}

	return 0;

ll_exit_clear_flag:
	dev->ongoing_usb_ll_op = 0;
	return rv;
}
```

8. Create the interrupt OUT URB's completion callback. The completion callback checks the URB status and re-submits the URB if there was an error status. If the transmission was successful, the callback wakes up the sleeping process and returns.

```
static void ltc3206_usb_cmpl_cbk(struct urb *urb)
{
	struct i2c_ltc3206 *dev = urb->context;
	int status = urb->status;
	int retval;

	switch (status) {
	case 0:			/* success */
		break;
	case -ECONNRESET:	/* unlink */
	case -ENOENT:
	case -ESHUTDOWN:
		return;
	/* -EPIPE:  should clear the halt */
	default:		/* error */
```

```
                goto resubmit;
        }

        /*
         * wake up the waiting function
         * modify the flag indicating the ll status
         */
        dev->ongoing_usb_ll_op = 0; /* communication is OK */
        wake_up_interruptible(&dev->usb_urb_completion_wait);
        return;

resubmit:
        retval = usb_submit_urb(urb, GFP_ATOMIC);
        if (retval) {
                dev_err(&dev->interface->dev,
                        "ltc3206(irq): can't resubmit intrerrupt urb, retval
%d\n",
                        retval);
        }
}
```

9. Add a usb_driver structure that will be registered to the USB core:

```
static struct usb_driver ltc3206_driver = {
        .name = DRIVER_NAME,
        .probe = ltc3206_probe,
        .disconnect = ltc3206_disconnect,
        .id_table = ltc3206_table,
};
```

10. Register your driver with the USB bus:

```
module_usb_driver(ltc3206_driver);
```

11. Build the module and load it to the target processor:

See in the next **Listing 13-3** the "I2C to USB Multidisplay LED" driver source code (usb_ltc3206.c).

Listing 13-3: usb_ltc3206.c

```
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/usb.h>
#include <linux/i2c.h>

#define DRIVER_NAME      "usb-ltc3206"

#define USB_VENDOR_ID_LTC3206          0x04d8
```

```
#define USB_DEVICE_ID_LTC3206          0x003f

#define LTC3206_OUTBUF_LEN             3      /* USB write packet length */
#define LTC3206_I2C_DATA_LEN           3

/* Structure to hold all of our device specific stuff */
struct i2c_ltc3206 {
   u8 obuffer[LTC3206_OUTBUF_LEN];     /* USB write buffer */
   /* I2C/SMBus data buffer */
   u8 user_data_buffer[LTC3206_I2C_DATA_LEN];
   int ep_out;                  /* out endpoint */
   struct usb_device *usb_dev;  /* the usb device for this device */
   struct usb_interface *interface;/* the interface for this device */
   struct i2c_adapter adapter;  /* i2c related things */
   /* wq to wait for an ongoing write */
   wait_queue_head_t usb_urb_completion_wait;
   bool ongoing_usb_ll_op;             /* all is in progress */
   struct urb *interrupt_out_urb;
};

/*
 * Return list of supported functionality.
 */
static u32 ltc3206_usb_func(struct i2c_adapter *a)
{
   return I2C_FUNC_I2C | I2C_FUNC_SMBUS_EMUL |
          I2C_FUNC_SMBUS_READ_BLOCK_DATA | I2C_FUNC_SMBUS_BLOCK_PROC_CALL;
}

/* usb out urb callback function */
static void ltc3206_usb_cmpl_cbk(struct urb *urb)
{
   struct i2c_ltc3206 *dev = urb->context;
   int status = urb->status;
   int retval;

   switch (status) {
   case 0:              /* success */
          break;
   case -ECONNRESET:    /* unlink */
   case -ENOENT:
   case -ESHUTDOWN:
          return;
   /* -EPIPE:  should clear the halt */
   default:             /* error */
          goto resubmit;
   }
```

```
	/*
	 * wake up the waiting function
	 * modify the flag indicating the ll status
	 */
	dev->ongoing_usb_ll_op = 0; /* communication is OK */
	wake_up_interruptible(&dev->usb_urb_completion_wait);
	return;

resubmit:
	retval = usb_submit_urb(urb, GFP_ATOMIC);
	if (retval) {
		dev_err(&dev->interface->dev,
			"ltc3206(irq): can't resubmit intrerrupt urb, retval %d\n",
			retval);
	}
}

static int ltc3206_ll_cmd(struct i2c_ltc3206 *dev)
{
	int rv;

	/*
	 * tell everybody to leave the URB alone
	 * we are going to write to the LTC3206 device
	 */
	dev->ongoing_usb_ll_op = 1; /* doing USB communication */

	/* submit the interrupt out URB packet */
	if (usb_submit_urb(dev->interrupt_out_urb, GFP_KERNEL)) {
		dev_err(&dev->interface->dev,
				"ltc3206(ll): usb_submit_urb intr out failed\n");
		dev->ongoing_usb_ll_op = 0;
		return -EIO;
	}

	/* wait for the transmit completion, the USB URB callback will signal it */
	rv = wait_event_interruptible(dev->usb_urb_completion_wait,
			(!dev->ongoing_usb_ll_op));
	if (rv < 0) {
		dev_err(&dev->interface->dev, "ltc3206(ll): wait interrupted\n");
		goto ll_exit_clear_flag;
	}

	return 0;

ll_exit_clear_flag:
	dev->ongoing_usb_ll_op = 0;
	return rv;
```

```
}

static int ltc3206_init(struct i2c_ltc3206 *dev)
{
   int ret;

   /* initialize the LTC3206 */
   dev_info(&dev->interface->dev,
            "LTC3206 at USB bus %03d address %03d -- ltc3206_init()\n",
            dev->usb_dev->bus->busnum, dev->usb_dev->devnum);

   /* allocate the int out URB */
   dev->interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);
   if (!dev->interrupt_out_urb){
          ret = -ENODEV;
          goto init_error;
   }

   /* Initialize the int out URB */
   usb_fill_int_urb(dev->interrupt_out_urb, dev->usb_dev,
                      usb_sndintpipe(dev->usb_dev,
                                      dev->ep_out),
                      (void *)&dev->obuffer, LTC3206_OUTBUF_LEN,
                      ltc3206_usb_cmpl_cbk, dev,
                      1);

   ret = 0;
   goto init_no_error;

init_error:
   dev_err(&dev->interface->dev, "ltc3206_init: Error = %d\n", ret);
   return ret;

init_no_error:
   dev_info(&dev->interface->dev, "ltc3206_init: Success\n");
   return ret;
}

static int ltc3206_i2c_write(struct i2c_ltc3206 *dev,
                                struct i2c_msg *pmsg)
{
   u8 ucXferLen;
   int rv;
   u8 *pSrc, *pDst;

   if (pmsg->len > LTC3206_I2C_DATA_LEN)
   {
          pr_info ("problem with the lenght\n");
```

```
		return -EINVAL;
	}

	/* I2C write lenght */
	ucXferLen = (u8)pmsg->len;

	pSrc = &pmsg->buf[0];
	pDst = &dev->obuffer[0];
	memcpy(pDst, pSrc, ucXferLen);

	pr_info("oubuffer[0] = %d\n", dev->obuffer[0]);
	pr_info("oubuffer[1] = %d\n", dev->obuffer[1]);
	pr_info("oubuffer[2] = %d\n", dev->obuffer[2]);

	rv = ltc3206_ll_cmd(dev);
	if (rv < 0)
		return -EFAULT;

	return 0;
}

/* device layer, called from the I2C user app */
static int ltc3206_usb_i2c_xfer(struct i2c_adapter *adap,
		struct i2c_msg *msgs, int num)
{
	struct i2c_ltc3206 *dev = i2c_get_adapdata(adap);
	struct i2c_msg *pmsg;
	int ret, count;

	pr_info("number of i2c msgs is = %d\n", num);

	for (count = 0; count < num; count++) {
		pmsg = &msgs[count];
		ret = ltc3206_i2c_write(dev, pmsg);
		if (ret < 0)
			goto abort;
	}

	/* if all the messages were transferred ok, return "num" */
	ret = num;
abort:
	return ret;
}

static const struct i2c_algorithm ltc3206_usb_algorithm = {
	.master_xfer = ltc3206_usb_i2c_xfer,
	.functionality = ltc3206_usb_func,
};
```

```c
static const struct usb_device_id ltc3206_table[] = {
   { USB_DEVICE(USB_VENDOR_ID_LTC3206, USB_DEVICE_ID_LTC3206) },
   { }
};
MODULE_DEVICE_TABLE(usb, ltc3206_table);

static void ltc3206_free(struct i2c_ltc3206 *dev)
{
   usb_put_dev(dev->usb_dev);
   usb_set_intfdata(dev->interface, NULL);
   kfree(dev);
}

static int ltc3206_probe(struct usb_interface *interface,
                      const struct usb_device_id *id)
{
   struct usb_host_interface *hostif = interface->cur_altsetting;
   struct i2c_ltc3206 *dev;
   int ret;

   dev_info(&interface->dev, "ltc3206_probe() function is called.\n");

   /* allocate memory for our device and initialize it */
   dev = kzalloc(sizeof(*dev), GFP_KERNEL);
   if (dev == NULL) {
          pr_info("i2c-ltc3206(probe): no memory for device state\n");
          ret = -ENOMEM;
          goto error;
   }

   /* get ep_out address */
   dev->ep_out = hostif->endpoint[1].desc.bEndpointAddress;

   dev->usb_dev = usb_get_dev(interface_to_usbdev(interface));
   dev->interface = interface;

   init_waitqueue_head(&dev->usb_urb_completion_wait);

   /* save our data pointer in this interface device */
   usb_set_intfdata(interface, dev);

   /* setup I2C adapter description */
   dev->adapter.owner = THIS_MODULE;
   dev->adapter.class = I2C_CLASS_HWMON;
   dev->adapter.algo = &ltc3206_usb_algorithm;
   i2c_set_adapdata(&dev->adapter, dev);
```

```
    snprintf(dev->adapter.name, sizeof(dev->adapter.name),
             DRIVER_NAME " at bus %03d device %03d",
             dev->usb_dev->bus->busnum, dev->usb_dev->devnum);

    dev->adapter.dev.parent = &dev->interface->dev;

    /* initialize the ltc3206 device */
    ret = ltc3206_init(dev);
    if (ret < 0) {
            dev_err(&interface->dev, "failed to initialize adapter\n");
            goto error_init;
    }

    /* and finally attach to I2C layer */
    ret = i2c_add_adapter(&dev->adapter);
    if (ret < 0) {
            dev_info(&interface->dev, "failed to add I2C adapter\n");
            goto error_i2c;
    }

    dev_info(&dev->interface->dev,
                    "ltc3206_probe() -> chip connected -> Success\n");
    return 0;

error_init:
    usb_free_urb(dev->interrupt_out_urb);

error_i2c:
    usb_set_intfdata(interface, NULL);
    ltc3206_free(dev);
error:
    return ret;
}

static void ltc3206_disconnect(struct usb_interface *interface)
{
    struct i2c_ltc3206 *dev = usb_get_intfdata(interface);

    i2c_del_adapter(&dev->adapter);

    usb_kill_urb(dev->interrupt_out_urb);
    usb_free_urb(dev->interrupt_out_urb);

    usb_set_intfdata(interface, NULL);
    ltc3206_free(dev);

    pr_info("i2c-ltc3206(disconnect) -> chip disconnected");
}
```

```
static struct usb_driver ltc3206_driver = {
   .name = DRIVER_NAME,
   .probe = ltc3206_probe,
   .disconnect = ltc3206_disconnect,
   .id_table = ltc3206_table,
};

module_usb_driver(ltc3206_driver);

MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a usb controlled i2c ltc3206 device");
MODULE_LICENSE("GPL");
```

usb_ltc3206.ko Demonstration

```
/*
 * Connect the PIC32MX470 Curiosity Development Board USB Micro-B port (J12) to
 * one of the four USB HostType-A connectors of the Raspberry Pi 4 Model B board.
 * Power the Raspberry Pi 4 Model B board to boot the processor. Keep the
 * PIC32MX470 board powered off
 */

/* check the i2c adapters of the Raspberry Pi 4 Model B board */
root@raspberrypi:/home# i2cdetect -l
i2c-1   i2c             bcm2835 (i2c@7e804000)                  I2C adapter

root@raspberrypi:/home# insmod usb_ltc3206.ko  /* load the module */
usb_ltc3206: loading out-of-tree module taints kernel.
usbcore: registered new interface driver usb-ltc3206

/* power now the PIC32MX Curiosity board */
root@raspberrypi:/home# usb 1-1.3: new full-speed USB device number 4 using
dwc_otg
usb 1-1.3: New USB device found, idVendor=04d8, idProduct=003f, bcdDevice= 1.00
usb 1-1.3: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-1.3: Product: USB to I2C demo
usb 1-1.3: Manufacturer: Microchip Technology Inc.
usb-ltc3206 1-1.3:1.0: ltc3206_probe() function is called.
usb-ltc3206 1-1.3:1.0: LTC3206 at USB bus 001 address 004 -- ltc3206_init()
usb-ltc3206 1-1.3:1.0: ltc3206_init: Success
usb-ltc3206 1-1.3:1.0: ltc3206_probe() -> chip connected -> Success


/* check again the i2c adapters of the Raspberry Pi 4 Model B board, find the new
one */

root@raspberrypi:/home# i2cdetect -l

i2c-1   i2c             bcm2835 (i2c@7e804000)                  I2C adapter
```

```
i2c-11  i2c             usb-ltc3206 at bus 001 device 004       I2C adapter

root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# ls
authorized         bInterfaceProtocol  ep_01     power
bAlternateSetting  bInterfaceSubClass  ep_81     subsystem
bInterfaceClass    bNumEndpoints       i2c-4     supports_autosuspend
bInterfaceNumber   driver              modalias  uevent

/*
 * verify the communication between the host and device
 * these commands toggle the three leds of the PIC32MX board and
 * set maximum brightness of the LTC3206 LED BLUE
 */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0x00 0xf0 0x00
i
number of i2c msgs is = 1
oubuffer[0] = 0
oubuffer[1] = 240
oubuffer[2] = 0

/* set maximum brightness of the LTC3206 LED RED */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0xf0 0x00 0x00
i

/* decrease brightness of the LTC3206 LED RED */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0x10 0x00 0x00
i

/* set maximum brightness of the LTC3206 LED GREEN */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0x00 0x0f 0x00
i

/* set maximum brightness of the LTC3206 LED GREEN and SUB display */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0x00 0x0f 0x0f
i

/* set maximum brightness of the LTC3206 MAIN display */
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0x00 0x00 0xf0
i

root@raspberrypi:/home# rmmod usb_ltc3206.ko /* remove the module */
usbcore: deregistering interface driver usb-ltc3206

/* Power off the PIC32MX Curiosity board */
root@raspberrypi:/home# i2c-ltc3206(disconnect) -> chip disconnected
usb 1-1.3: USB disconnect, device number 4
```