Linux Driver Development for Embedded Processors

Raspberry Pi 3 Practical Labs

Building a Linux embedded system for the Raspberry Pi 3 Model B

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

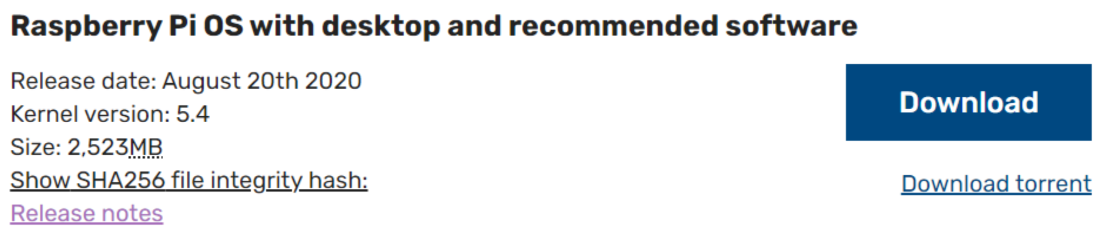
The ARM cores run at 1.2GHz, making the device about 50% faster tha n the Raspberry Pi 2. The VideoCore IV runs at 400MHz. You can see the documentation for BCM2836 at https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md and for BCM2835 at https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/README.md.

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

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 in 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

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:

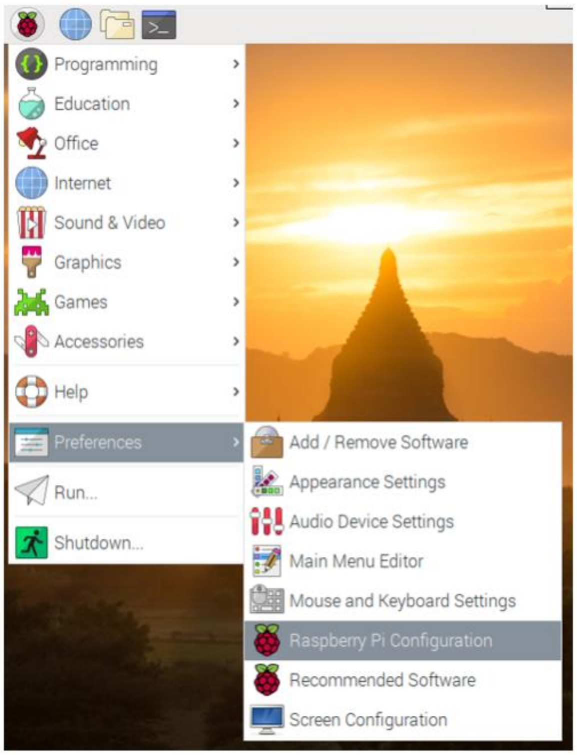
```
~$ lsblk
~$ mkdir ~/mnt
~$ mkdir ~/mnt/fat32
~$ mkdir ~/mnt/ext4
~$ sudo mount /dev/mmcblk0p1 ~/mnt/fat32
~$ ls -l ~/mnt/fat32/ /* see the files in the fat32 partition, check that
config.txt is included */
```

Update the config.txt file adding the next values:

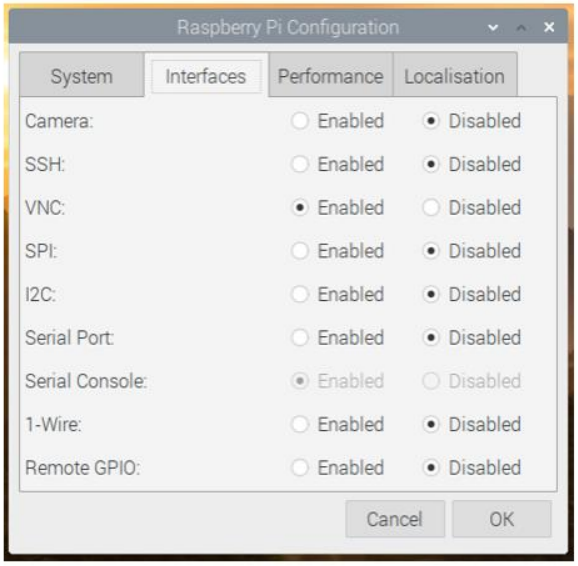
```
~$ cd ~/mnt/fat32/
~/mnt/fat32$ sudo nano config.txt
```

```
dtparam=i2c_arm=on
dtparam=spi=on
dtoverlay=spi0-cs
# Enable UART
enable_uart=1
kernel=kernel7.img
```

You can also update previous settings (after booting the Raspberry Pi 3 board) through the Raspberry Pi 3 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:



Raspberry Pi Configuration

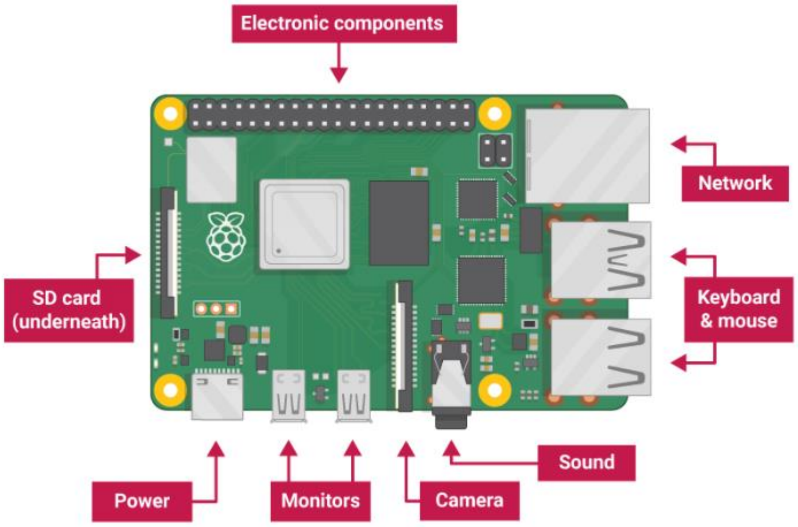
System | Interfaces | Performance | Localisation

Camera:	Enabled	[✓ Disabled]
SSH:	Enabled	[✓ Disabled]
VNC:	[✓ Enabled]	Disabled
SPI:	Enabled	[✓ Disabled]
I2C:	Enabled	[✓ Disabled]
Serial Port:	Enabled	[✓ Disabled]
Serial Console:	[✓ Enabled]	Disabled
1-Wire:	Enabled	[✓ Disabled]
Remote GPIO:	Enabled	[✓ Disabled]

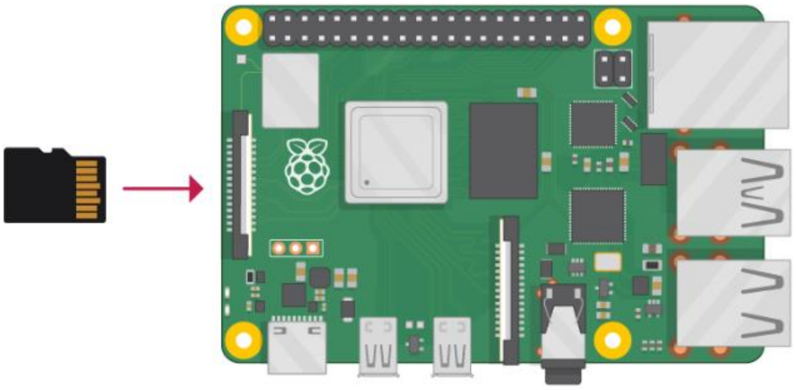
Cancel OK

Connect and set up hardware

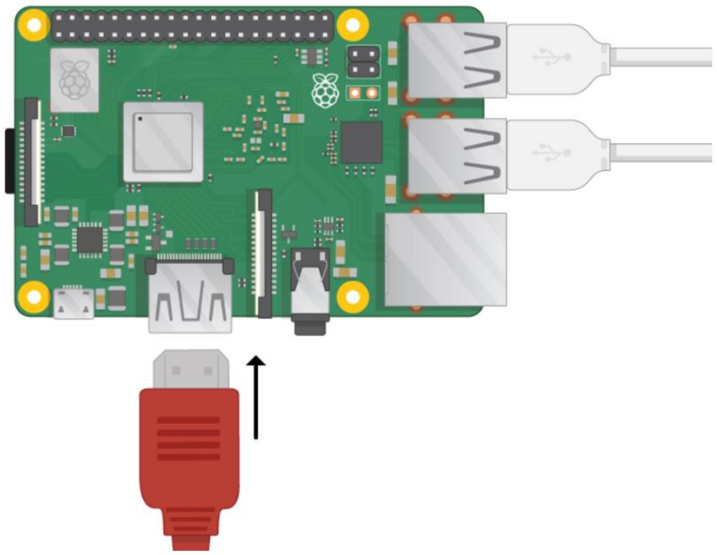
Now get everything connected to your Raspberry Pi 3. It's important to do this in the right order, so that all your components are safe.



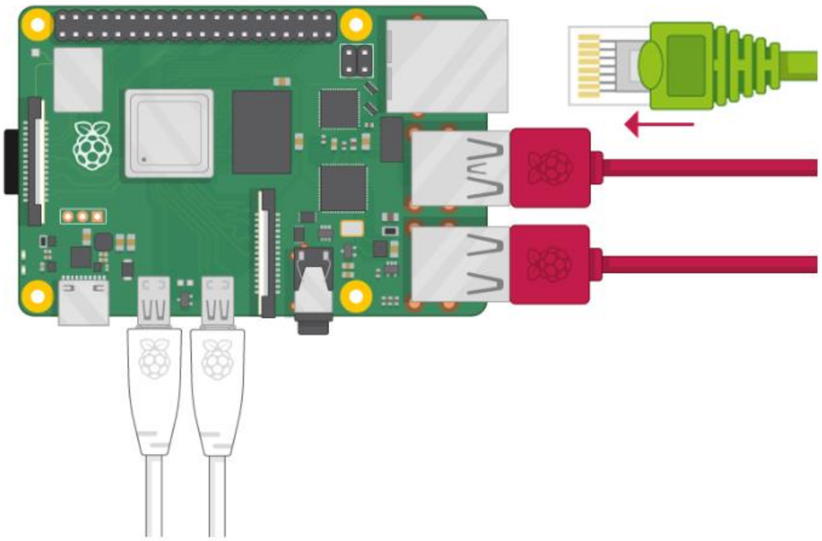
Insert the uSD card you've set up with **Raspberry Pi OS** into the microSD card slot on the underside of your Raspberry Pi 3.



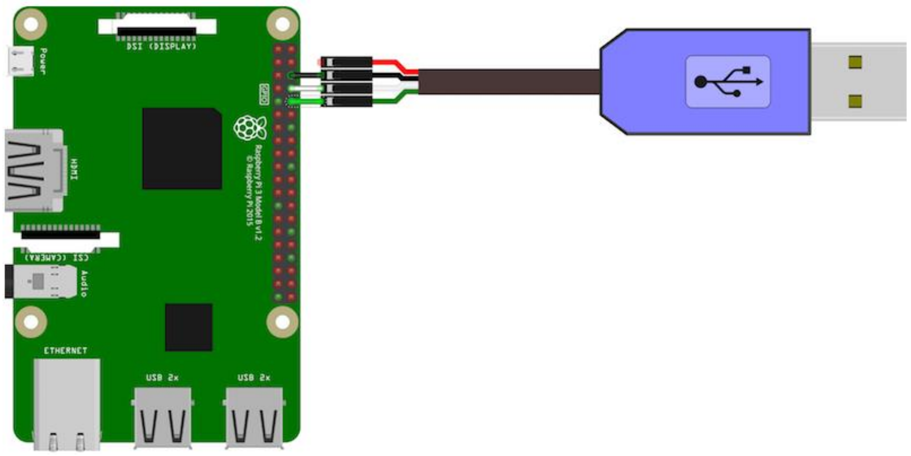
Connect your screen to the single Raspberry Pi 3's HDMI port. You can also connect a mouse to an USB port and keyboard in the same way.



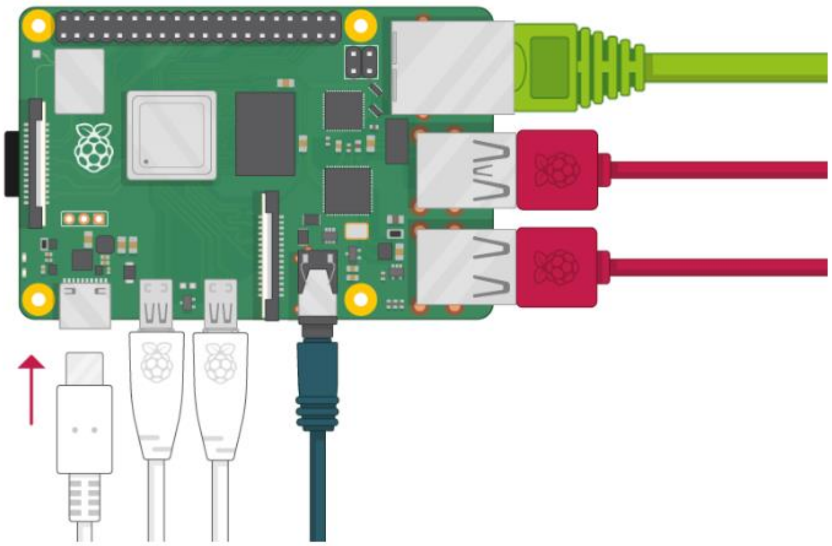
Connect your Raspberry Pi 3 to the internet via Ethernet, use an Ethernet cable to connect the Ethernet port on Raspberry Pi 3 to an Ethernet socket on the host PC.



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 3, which indicates that Raspberry Pi 3 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 3 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 3'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 3:

```
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 your Raspberry Pi 3 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 3 remotely from another computer or device on the same network using SSH. Make sure your Raspberry Pi 3 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 3 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** (also comment the line out). After editing the file type "Ctrl+x", then type "yes" and press "enter" to exit.

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

Building the Linux kernel

There are two main methods for building the kernel. You can build locally on the Raspberry Pi 3, 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=kernel7
```

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

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

```
~/linux$ 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/mmcblk0p1 ~/mnt/fat32/
~$ sudo mount /dev/mmcblk0p2 ~/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/kernel7.img ~/mnt/fat32/kernel7-backup.img
~/linux$ sudo cp arch/arm/boot/zImage ~/mnt/fat32/kernel7.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
5.4.80-v7+
```

If you modify later kernel or device tree files, you can copy them to the Raspberry Pi 3 remotely using SSH:

```
~/linux$ scp arch/arm/boot/zImage root@10.0.0.10:/boot/kernel7.img
~/linux$ scp arch/arm/boot/dts/bcm2710-rpi-4-b.dtb root@10.0.0.10:/boot/
```

Hardware descriptions for the Raspberry Pi 3 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 3 Model B labs

LAB 10.1 software description

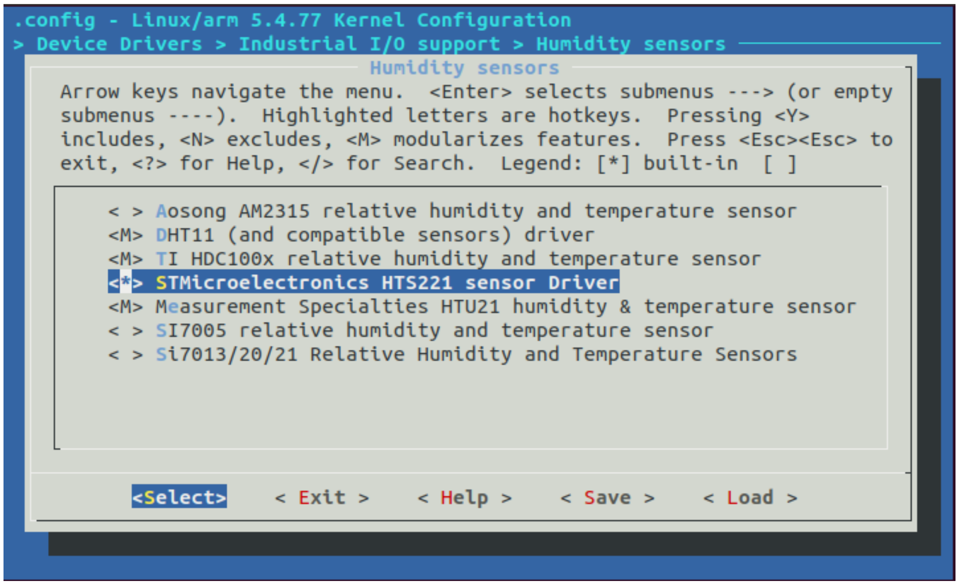
You have to install the evtest application to test this driver. Connect you Raspberry Pi 3 to the Internet and download the application:

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

LAB 12.1 software description

Functions for triggered buffer support are needed by this module. 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 that includes this triggered buffer support.

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



```
.config - Linux/arm 5.4.77 Kernel Configuration
> Device Drivers > Industrial I/O support > Humidity sensors
                          Humidity sensors
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

      < > Aosong AM2315 relative humidity and temperature sensor
      <M> DHT11 (and compatible sensors) driver
      <M> TI HDC100x relative humidity and temperature sensor
      <*> STMicroelectronics HTS221 sensor Driver
      <M> Measurement Specialties HTU21 humidity & temperature sensor
      < > SI7005 relative humidity and temperature sensor
      < > Si7013/20/21 Relative Humidity and Temperature Sensors

         <Select>    < Exit >    < Help >    < Save >    < Load >
```

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

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

In the Host build the IIO tools:

```
~/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 3 Model B board are included in the linux_5.4_rpi3_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 3 Model B board using Linux kernel version 5.4. The Raspberry Pi 3 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.

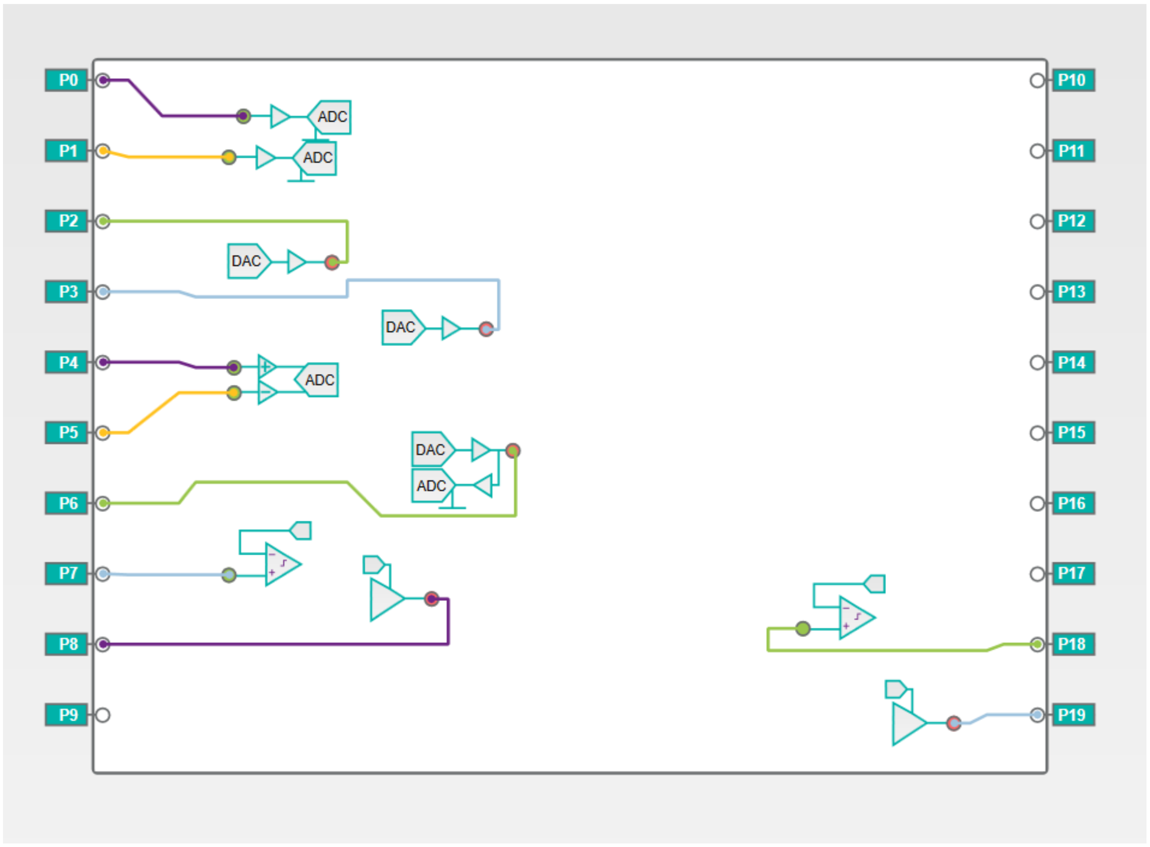
A new low cost evaluation board based on the MAX11300 device will be used, thus expanding the number of evaluation boards that can be adquired to practice with the theory explained in the Chapter 11.

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 3 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-3-model-b/?resellerType=home and https://www.mikroe.com/pixi-click

Before developing the driver, you can first create a custom design using the MAX11300 configuration GUI software. You will download this tool from Maxim's website. The MAX11300ConfigurationSetupV1.4.zip tool and the custom design used as a starting point for the development of the driver is included in the lab folder.

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

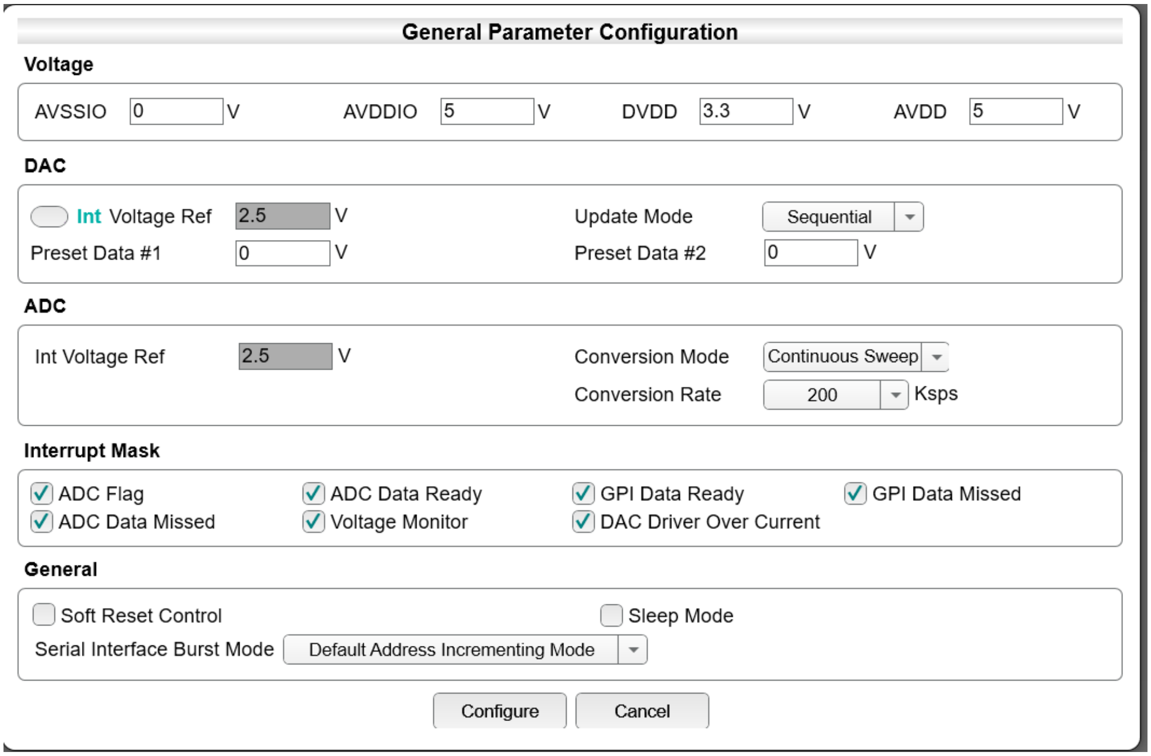


These are the parameters used during the configuration of the MAX11300 PIXI ports:

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

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

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


General Parameter Configuration

Voltage

AVSSIO 0 V AVDDIO 5 V DVDD 3.3 V AVDD 5 V

DAC

Int Voltage Ref 2.5 V Update Mode Sequential
Preset Data #1 0 V Preset Data #2 0 V

ADC

Int Voltage Ref 2.5 V Conversion Mode Continuous Sweep
Conversion Rate 200 Ksps

Interrupt Mask

✓ ADC Flag ✓ ADC Data Ready ✓ GPI Data Ready ✓ GPI Data Missed
✓ ADC Data Missed ✓ Voltage Monitor ✓ DAC Driver Over Current

General

☐ Soft Reset Control ☐ Sleep Mode
Serial Interface Burst Mode Default Address Incrementing Mode

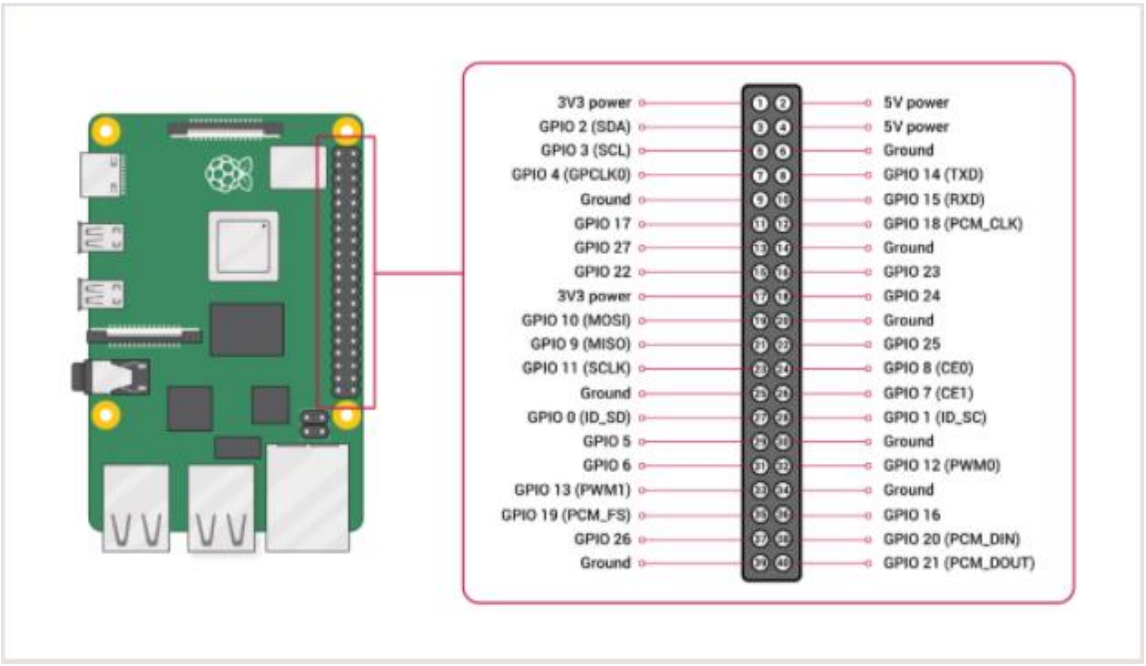
Not all the MAX11300 specifications were included during the development of this driver. These are the main specifications that have been included:

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

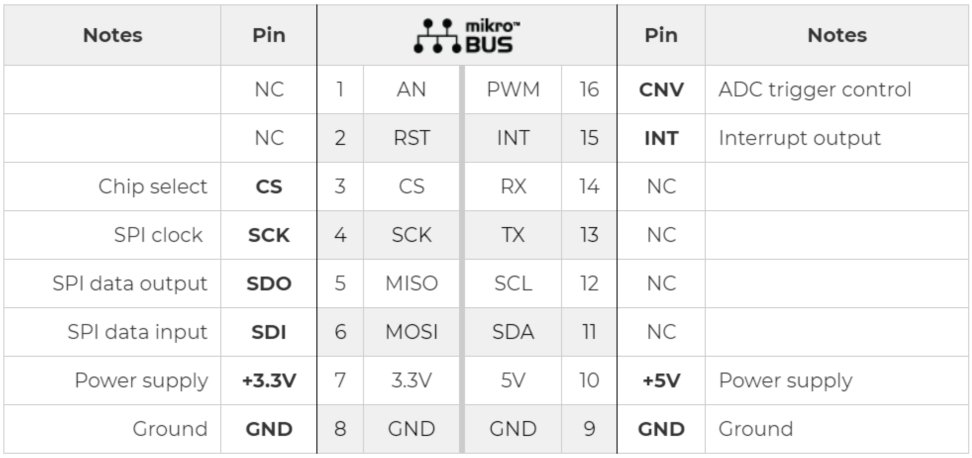
- DAC Update Mode: Sequential.
- ADC Conversion Mode: Continuous Sweep.
- Default ADC Conversion Rate of 200Ksps.
- Interrupts are masked.

LAB 11.5 hardware description

In this lab, you will use the SPI pins of the Raspberry Pi 3 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 3 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

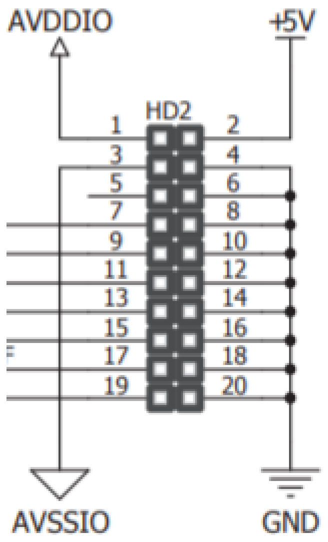
Connect the Raspberry Pi 3 Model B SPI pins to the MAX11300 SPI ones obtained from the PIXI™ CLICK mikroBUS™ socket:

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

Also connect the next power pins between the two boards:

- Connect Raspberry Pi 3 Model B **3.3V** to MAX11300 **3.3V** (Pin 7 of Mikrobus)
- Connect Raspberry Pi 3 Model B **5V** to MAX11300 **5V** (Pin 10 of Mikrobus)
- Connect Raspberry Pi 3 Model B **GNDs** to MAX11300 **GNDs** (Pin 9 and Pin 8 of Mikrobus)

Finally, find the HD2 connector in the PIXI™ CLICK schematic
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 bcm2710-rpi-3-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 bcm2710-rpi-3-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 serial device reg property included in all the channel childrens follows this specification 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 children 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 have been configured with the same parameters that were used during 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 bcm2710-rpi-3-b.dts DT file.

```
/dts-v1/;

#include "bcm2710.dtsi"
#include "bcm2709-rpi.dtsi"
#include "bcm283x-rpi-smsc9514.dtsi"
#include "bcm283x-rpi-csi1-2lane.dtsi"
#include "bcm283x-rpi-i2c0mux_0_44.dtsi"
#include "bcm271x-rpi-bt.dtsi"
#include <dt-bindings/iio/maxim,max11300.h>
```

The maxim,max11300.h file includes the values of the DT binding properties that will be used for the DT channel children nodes. You have to place the maxim,max11300.h file under the next iio folder inside 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 a 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 a SPI interaction

These are the main code sections:

1. Include the required header files:

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

2. Create a struct spi_driver structure:

   ```
   static struct spi_driver 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 matchs 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 struct 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 struct 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 DT channel children nodes of the DT max11300 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 struct 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 (entered through the user space via a IIO sysfs entry) to the selected port DAC data register using a SPI transaction.

When the max11300_read_adc() function receives the info_mask value [IIO_CHAN_INFO_RAW] = "raw", it first reads the received ADC channel address value to select the ADC port mode. Once the ADC port mode has been discovered, then max11300_reg_read() or max11300_reg_read_differential() functions are called, which get the value of the selected port ADC data register via a SPI transaction. 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.

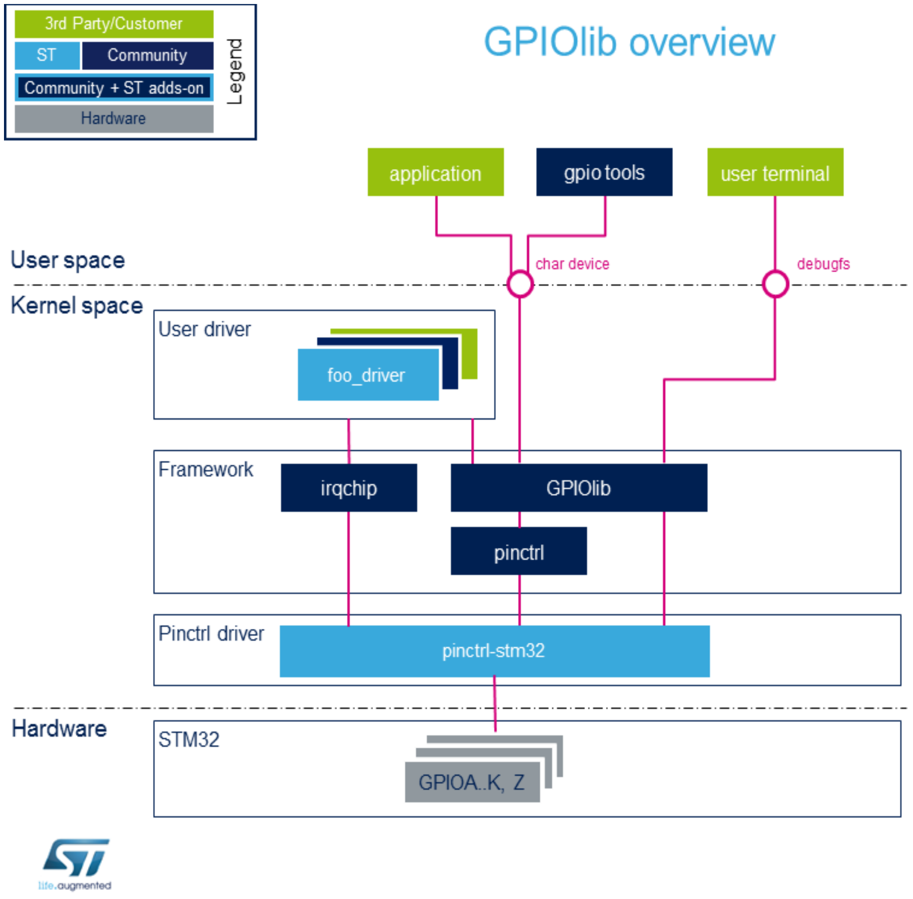
In the Chapter 5 of this book , you saw how to control GPIOs from kernel space using the GPIO descriptor consumer interface of the GPIOLib framework.

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 controllers 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 will configure the ports of the MAX11300 device as GPIOs, set the direction of the GPIOs (input or output) and control the ouput level of the GPIO lines (low or high ouput 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 3 Model B processor.

Note: The " IIO Mixed-Signal I/O Device" driver source code developed for the Raspberry Pi 3 Model B board is included in the linux_5.4_rpi3_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 3 to 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_rpi3_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_rpi3_drivers/linux_5.4_max11300_driver/
```

Compile and deploy the drivers to the Raspberry Pi 3 Model B board:

```
~/linux_5.4_rpi3_drivers/linux_5.4_max11300_driver$ make
```

```
~/linux_5.4_rpi3_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 dic  5 12:52 /dev/gpiochip0
crw-rw---- 1 root gpio 254, 1 dic  5 12:52 /dev/gpiochip1
crw-rw---- 1 root gpio 254, 2 dic  5 12:52 /dev/gpiochip2
crw-rw---- 1 root gpio 254, 3 dic  5 12:53 /dev/gpiochip3

/* Print information of all the lines of the gpiochip3 */
root@raspberrypi:/home# gpioinfo gpiochip3
gpiochip3 - 4 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
        line   2:      unnamed       unused   input  active-high
        line   3:      unnamed       unused   input  active-high
```

connect port19 (GPO) to port 18 (GPI)

```
/* Set port19 (GPO) to high */
```

```
root@raspberrypi:/home# gpioset gpiochip3 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 gpiochip3 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 gpiochip3 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 gpiochip3 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 gpiochip3 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 gpiochip3 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 gpiochip3 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 gpiochip3 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 gpiochip3 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 gpiochip3 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 gpiochip3 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 gpiochip3 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 3 Model B :~# gpioinfo gpiochip3
gpiochip3 - 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  output  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 using a device node and the second method will control the GPIO using the functions of the libgpiod library.

GPIO control through character device

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 struct 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 APIs in the user space that will control the GPIOs through ioctl calls on char device files /dev/gpiochipX, 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 syfs interface is deprecated, and is highly recommend to use the new interface. These are some of the advanteages 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 of the Color click eval board (https://www.mikroe.com/color-click), to see the red LED blinking.

Send the application to the Raspberry Pi 3 Model B board and compile on it:

```
~/linux_5.4_rpi3_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 flashing!

```
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/gpiochip3"
```

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

    struct gpiohandle_data data;
    struct gpiohandle_request req;

    /* open gpio device */
    fd = open(DEVICE_GPIO, 3);
    if (fd < 0) {
        fprintf(stderr, "Failed to open %s\n", DEVICE_GPIO);
        return -1;
    }

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

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

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

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

        }
        sleep(1);
    }

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

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

    return ret;
}
```

GPIO control through 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 of 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 3 Model B board:

```
~/linux_5.4_rpi3_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 3 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 flashing!

```
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

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

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

   /* open /dev/gpiochip3 */
   output_chip = gpiod_chip_open_by_number(3);
   if (!output_chip)
          return -1;

   /* get line 3 (port19) of the gpiochip3 device */
   output_line = gpiod_chip_get_line(output_chip, 3);
   if(!output_line) {
          gpiod_chip_close(output_chip);
          return -1;
   }
```

```
	/* config port19 (GPO) as output and set 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_rpi3_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 an 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 adquired to practice with the theory explained in Chapter 7.

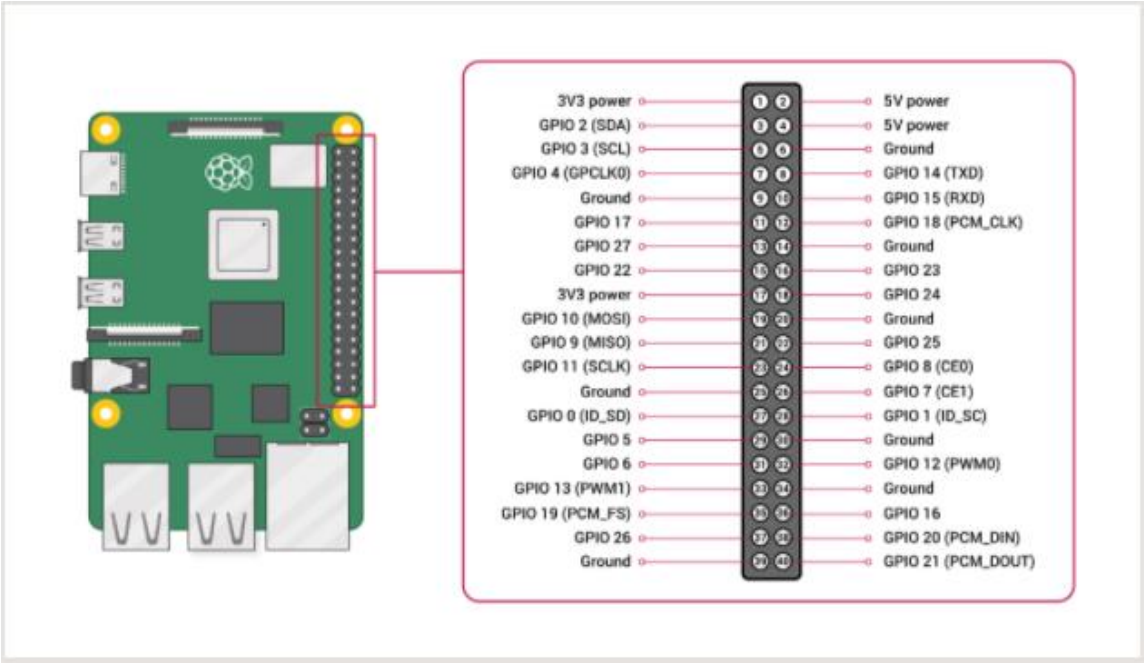
This new kernel module will control the Cypress CY8C9520A device. The CY8C9520A is a multi-port IO expander with on board user available EEPROM and several PWM outputs. The IO expander's data pins can be independently assigned as inputs, outputs, quasi-bidirectional input/outputs or PWM ouputs. The individual 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 3 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-3-model-b/?resellerType=home and https://www.mikroe.com/expand-6-click

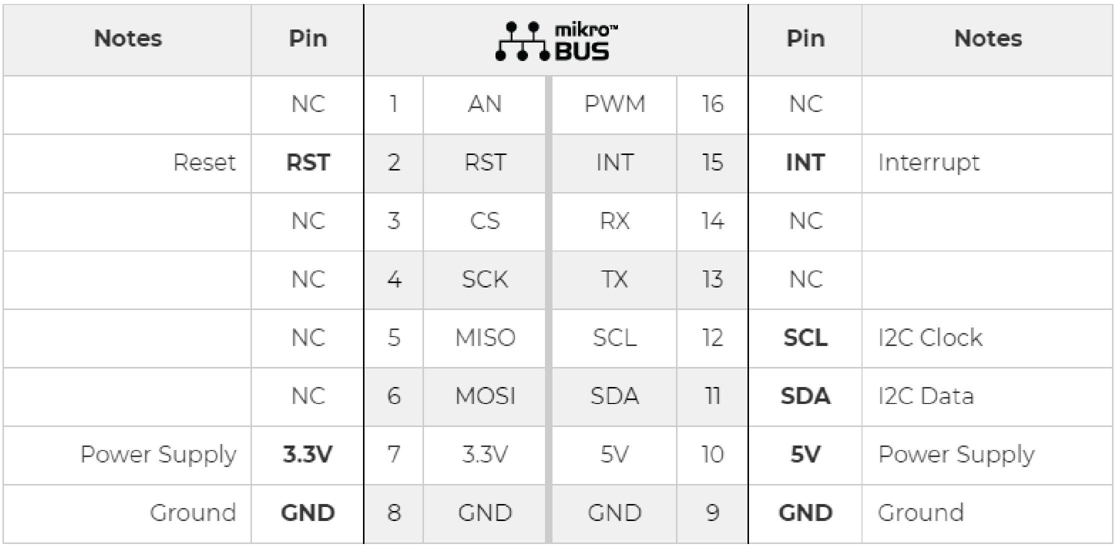
Not all the CY8C9520A features are included in this driver. The driver will configure the CY8C9520A port pins as input and outputs and will handle GPIO interrupts.

LAB 7.4 hardware description

In this lab, you will use the I2C pins of the Raspberry Pi 3 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 3 Model B connector:



And the EXPAND 6 Click mikroBUS™ socket:



Notes	Pin	mikro™ BUS				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 3 Model B I2C pins to the CY8C9520A I2C ones obtained from the EXPAND 6 Click mikroBUS™ socket:

- Connect Raspberry Pi 3 Model B **SCL** to CY8C9520A **SCL** (Pin 12 of Mikrobus)
- Connect Raspberry Pi 3 Model B **SDA** to CY8C9520A **SDA** (Pin 11 of Mikrobus)
- Connect Raspberry Pi 3 Model B **GPIO 23** to CY8C9520A **INT** (Pin 15 of Mikrobus)

Also connect the next power pins between the two boards:

- Connect Raspberry Pi 3 Model B **3.3V** to CY8C9520A **3.3V** (Pin 7 of Mikrobus)
- Connect Raspberry Pi 3 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 bcm2710-rpi-3-b.dts DT file and find the i2c1 controller master node. Inside the i2c1 node, you can see the pinctrl-0 property which configure the pins in I2C mode. The i2c1_pins are already defined in the bcm2710-rpi-3-b.dts file inside the gpio node property.

The i2c1 controller is enabled by writing "okay" to the status property. You will set to 100Khz the clock-frequency property. EXPAND 6 Click communicates with 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 from their parent node. The CY8C9520A Interrupt pin (INT) is connected to the GPIO 23 pin of the Raspberry Pi 3 Model B processor, so the interrupt parent of our device is the gpio peripheral of the Raspberry Pi 3 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 struct 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 matchs 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 struct 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_set_nested_irqchip() function sets up a nested cascaded irq handler for a gpio_chip from a parent IRQ. The The gpiochip_set_nested_irqchip() 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 cy8c9520a_irq_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 this 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 3 Model B processor.

Note: The "GPIO expander device" driver source code developed for the Raspberry Pi 3 Model B board is included in the linux_5.4_rpi3_drivers.zip file inside 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_rpi3.c

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

#define DRV_NAME                       "cy8c9520a"

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

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

/* definition of the global structure for the driver */
struct cy8c9520a {
   struct i2c_client *client;
   struct gpio_chip gpio_chip;
   struct gpio_desc *gpio;
   int irq;
   struct mutex lock;
   /* protect serialized access to the interrupt controller bus */
   struct mutex irq_lock;
   /* cached output registers */
   u8 outreg_cache[NPORTS];
   /* cached IRQ mask */
   u8 irq_mask_cache[NPORTS];
   /* IRQ mask to be applied */
   u8 irq_mask[NPORTS];
};

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

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

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

/*
 * struct gpio_chip get callback function.
 * It gets the input value of the GPIO line (0=low, 1=high)
 * accessing to the REG_INPUT_PORT register
 */
static int cy8c9520a_gpio_get(struct gpio_chip *chip,
                              unsigned int gpio)
{
   int ret;
   u8 port, in_reg;
   struct cy8c9520a *cygpio = gpiochip_get_data(chip);

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

   /* get the input port address 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

You will develop a GPIO child driver (int_rpi3_gpio) now, 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 bcm2710-rpi-3-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 matchs 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 3 Model B processor.

Note: The "GPIO child device" driver source code developed for the Raspberry Pi 3 Model B board is included in the linux_5.4_rpi3_drivers.zip, inside 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_rpi3_gpio.c

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

static char *INT_NAME = "P0_line0_INT";

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

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

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

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

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

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

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

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

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

	return 0;
}

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

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

MODULE_DEVICE_TABLE(of, my_of_ids);

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

module_platform_driver(my_platform_driver);

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

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

LAB 7.4 GPIO based IRQ application

In the previous section you have seen how to request and handle a GPIO 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 char device file /dev/gpiochip3.

Note: The "GPIO based IRQ application" source code developed for the Raspberry Pi 3 Model B board is included in the linux_5.4_rpi3_drivers.zip, inside 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

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

#define DEV_GPIO  "/dev/gpiochip3"

#define POLL_TIMEOUT -1 /* No timeout */

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

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

    /* open gpio */
    fd = open(DEV_GPIO, O_RDWR);
    if (fd < 0) {
        printf("ERROR: open %s ret=%d\n", DEV_GPIO, fd);
        return -1;
    }
    /* Request GPIO P0 first line interrupt */
    req.lineoffset = 0;
    req.handleflags = GPIOHANDLE_REQUEST_INPUT;
    req.eventflags  = GPIOEVENT_REQUEST_BOTH_EDGES;
    strncpy(req.consumer_label, "gpio_irq", sizeof(req.consumer_label) - 1);

    /* 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_rpi3_drivers.zip file from the github of the book and unzip it in the home folder of your Linux host:

```
~/linux_5.4_rpi3_drivers$ cd linux_5.4_CY8C9520A_driver
```

Compile and deploy the drivers and the application to the **Raspberry Pi 3 Model B** board:

```
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver$ make

~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver$ make deploy

~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver/linux_5.4_gpio_int_driver$
make

~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver/linux_5.4_gpio_int_driver$
make deploy

~/linux_5.4_rpi3_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
```

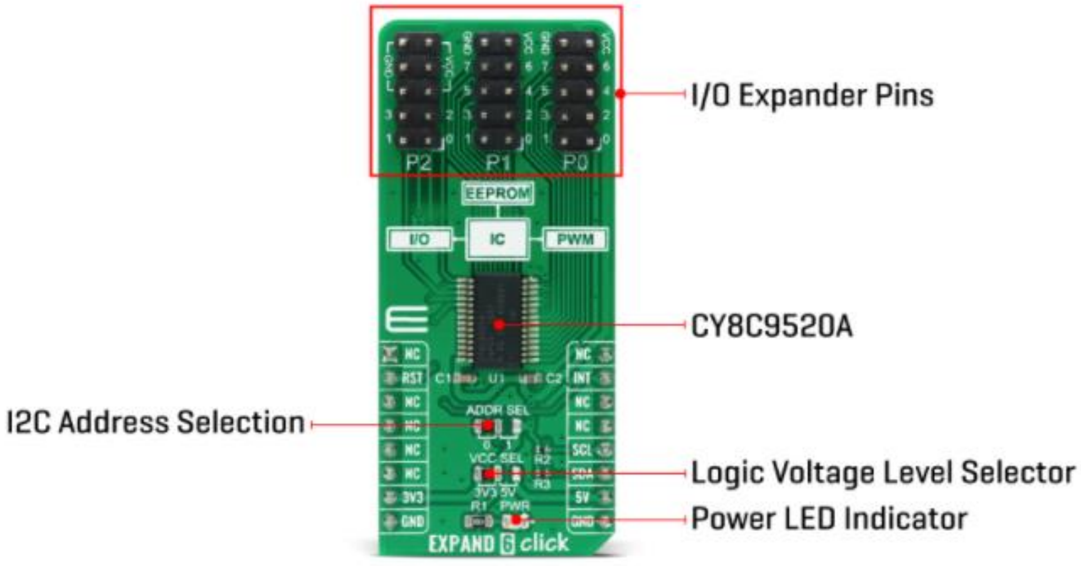
Follow the next instructions to test the drivers:

```
/* 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 gpiochip3 */
root@raspberrypi:/home# gpioinfo gpiochip3
gpiochip3 - 20 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
        line   2:      unnamed       unused   input  active-high
        line   3:      unnamed       unused   input  active-high
        line   4:      unnamed       unused   input  active-high
        line   5:      unnamed       unused   input  active-high
        line   6:      unnamed       unused   input  active-high
        line   7:      unnamed       unused   input  active-high
        line   8:      unnamed       unused   input  active-high
        line   9:      unnamed       unused   input  active-high
```

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

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

```
/* the gpio lines of the gpiochip3 are configured with internal pull-up to Vcc */
```



```
/* set to high level the pin 1 of P0 */
root@raspberrypi:/home# gpioset gpiochip3 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 gpiochip3 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 gpiochip3 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 gpiochip3 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
```
[  778.013607] int_gpio_expand soc:int_gpio: my_probe() function is called.
[  778.020652] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[  778.026584] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[  778.033184] cy8c9520a 1-0020: gpio 0 is unmasked
[  778.041794] the interrupt ISR has been entered
[  778.046411] cy8c9520a 1-0020: the REG_INTR_MASK value is 254
[  778.052254] int_gpio_expand soc:int_gpio: IRQ_using_platform_get_irq: 167
[  778.059498] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[  778.065665] cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
[  778.071591] cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
[  778.077394] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[  778.084696] int_gpio_expand soc:int_gpio: mydev: got minor 60
[  778.090599] int_gpio_expand soc:int_gpio: my_probe() function is exited.


/* check the gpio interrupt with Linux IRQ number 167 */
```
root@raspberrypi:/home# cat /proc/interrupts
```
CPU0       CPU1       CPU2       CPU3
 17:        878          0          0          0  ARMCTRL-level   1 Edge      3f
00b880.mailbox
 18:        720          0          0          0  ARMCTRL-level   2 Edge      VC
HIQ doorbell
 40:          0          0          0          0  ARMCTRL-level  48 Edge      bc
m2708_fb DMA
 42:        215          0          0          0  ARMCTRL-level  50 Edge      DM
```

```
A IRQ
 44:       6577          0          0          0  ARMCTRL-level  52 Edge      DM
A IRQ
 45:          0          0          0          0  ARMCTRL-level  53 Edge      DM
A IRQ
 48:          0          0          0          0  ARMCTRL-level  56 Edge      DM
A IRQ
 53:       1793          0          0          0  ARMCTRL-level  61 Edge      tt
yS0
 56:     221387          0          0          0  ARMCTRL-level  64 Edge      dw
c_otg, dwc_otg_pcd, dwc_otg_hcd:usb1
 77:         76          0          0          0  ARMCTRL-level  85 Edge      3f
804000.i2c
 78:          0          0          0          0  ARMCTRL-level  86 Edge      3f
204000.spi
 80:        475          0          0          0  ARMCTRL-level  88 Edge      mm
c0
 81:       4272          0          0          0  ARMCTRL-level  89 Edge      ua
rt-pl011
 86:       9185          0          0          0  ARMCTRL-level  94 Edge      mm
c1
161:          0          0          0          0  bcm2836-timer   0 Edge      ar
ch_timer
162:       4677       8130       4250       5834  bcm2836-timer   1 Edge      ar
ch_timer
165:          0          0          0          0  bcm2836-pmu   9 Edge      arm-
pmu
166:          1          0          0          0  pinctrl-bcm2835  23 Level
1-0020
167:          0          0          0          0  cy8c9520a-irq   0 Edge      P0
_line0_INT
FIQ:                usb_fiq
IPI0:          0          0          0          0  CPU wakeup interrupts
IPI1:          0          0          0          0  Timer broadcast interrupts
IPI2:       3927       8767       6914       7539  Rescheduling interrupts
IPI3:        416       1136       1334       1307  Function call interrupts
IPI4:          0          0          0          0  CPU stop interrupts
IPI5:       1568       2207        450        933  IRQ work interrupts
IPI6:          0          0          0          0  completion interrupts
Err:          0

/* Connect pin 0 of P0 to GND, then disconnect it from GND. Two interrupts are
fired */
root@raspberrypi:/home# [  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 gpiochip3 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 Linux CY8C9520A_pwm_pinctrl driver, that we will develop in this LAB 7.5 is an extension of the previous CY8C9520A_rpi4 driver, to which we 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´s 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 struct pinctrl_desc and registering it to the pinctrl subsystem with the devm_pinctrl_register() function. See below the setup of the pintrl_desc structure, done inside 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,
};
```

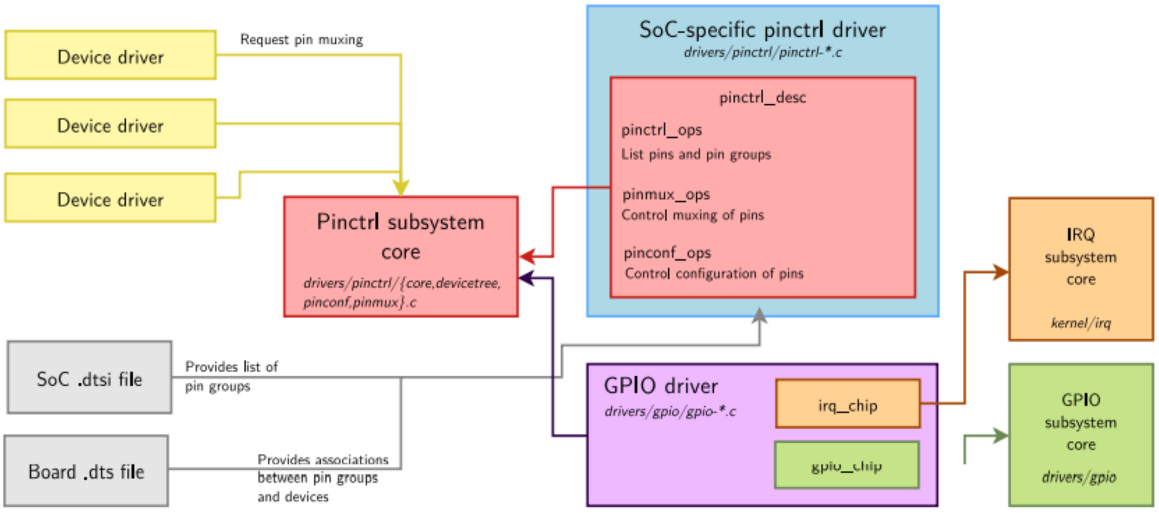
See below the code of the cygpio_pinconf_set callback function:

```
/* 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.

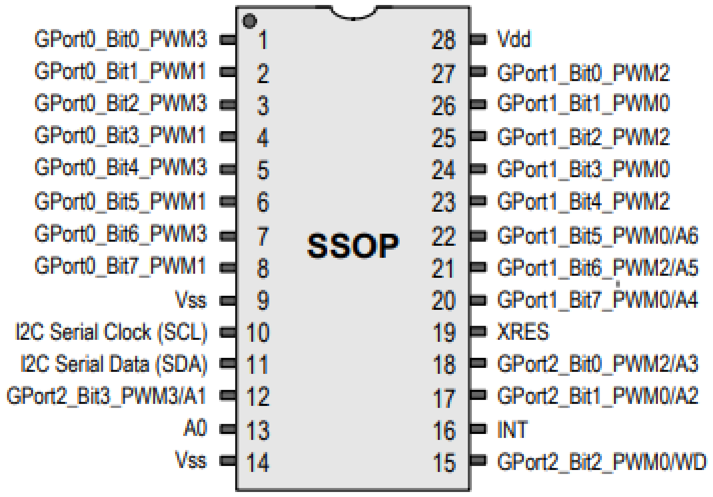
Finally, you will add the following lines in bold to the device-tree configuration of our cy8c9520a device:

```
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 to 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 3 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 3 Model B board is included in the linux_5.4_rpi3_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_rpi3_drivers.zip file from the github of the book and unzip it in the home folder of your Linux host:

```
~/linux_5.4_rpi3_drivers$ cd linux_5.4_CY8C9520A_pwm_pinctrl
```

Compile and deploy the drivers and the application to the **Raspberry Pi 3 Model B** board:

```
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_pwm_pinctrl$ make

~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_pwm_pinctrl$ make deploy

~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_pwm_pinctrl/linux_5.4_gpio_int_driver
$ make

~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_pwm_pinctrl/linux_5.4_gpio_int_driver
$ make deploy

~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_pwm_pinctrl/app$ make

~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_pwm_pinctrl/app$ make deploy
```

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
```