

---

# Linux Driver Development for Embedded Processors

ST STM32MP1 Practical Labs

---

# Building a Linux embedded system for the ST STM32MP1 processor

The STM32MP1 microprocessor series is based on a heterogeneous single or dual Arm Cortex-A7 and Cortex-M4 cores architecture, strengthening its ability to support multiple and flexible applications, achieving the best performance and power figures at any time. The Cortex-A7 core provides access to open-source operating systems (Linux/Android) while the Cortex-M4 core leverages the STM32 MCU ecosystem.

You can check all the info related to this family at

<https://www.st.com/en/microcontrollers-microprocessors/stm32mp1-series.html#overview>

For the development of the labs the **STM32MP157C-DK2** Discovery kit will be used. The documentation of this board can be found at

<https://www.st.com/en/evaluation-tools/stm32mp157c-dk2.html>

## Connect and set up hardware

To set up the STM32MP15 Discovery kit connections follow the steps indicated in the STM32 MPU wiki section located at

[https://wiki.st.com/stm32mpu/wiki/Getting\\_started/STM32MP1\\_boards/STM32MP157x-DK2](https://wiki.st.com/stm32mpu/wiki/Getting_started/STM32MP1_boards/STM32MP157x-DK2)

## Creating the structure for the STM32MPU embedded software distribution

The STM32MPU embedded software distribution for STM32 microprocessor platforms supports three software packages.

- The **Starter Package** to quickly and easily start with any STM32MPU microprocessor device. The Starter Package is generated from the Distribution Package.
- The **Developer Package** to add your own developments on top of the STM32MPU Embedded Software distribution, or to replace the Starter Package pre-built binaries. The Developer Package is generated from the Distribution Package.
- The **Distribution Package** to create your own Linux® distribution, your own Starter Package and your own Developer Package.

Create your <working directory> and assign a unique name to it (for example by including the release name).

```
PC:~$ mkdir STM32MP15-Ecosystem-v2.0.0
```

```
PC:~$ cd STM32MP15-Ecosystem-v2.0.0
```

Create the first-level directories that will host the software packages delivered through the STM32MPU embedded software distribution release note.

```
PC:~/STM32MP15-Ecosystem-v2.0.0$ mkdir Starter-Package
```

```
PC:~/STM32MP15-Ecosystem-v2.0.0$ mkdir Developer-Package
```

```
PC:~/STM32MP15-Ecosystem-v2.0.0$ mkdir Distribution-Package
```

## Populate the target and boot the image

To populate the STM32MP15 Discovery kit with the Starter Package follow the steps indicated in the STM32 MPU wiki section located at

[https://wiki.st.com/stm32mpu/wiki/Getting\\_started/STM32MP1\\_boards/STM32MP157x-DK2/Let%27s\\_start/Populate\\_the\\_target\\_and\\_boot\\_the\\_image](https://wiki.st.com/stm32mpu/wiki/Getting_started/STM32MP1_boards/STM32MP157x-DK2/Let%27s_start/Populate_the_target_and_boot_the_image)

## Installing the SDK for the developer package

To download the STM32MP1 Developer Package SDK for the STM32MP15-Ecosystem-v2.0.0 release follow the steps indicated in the STM32 MPU wiki section located at

[https://wiki.st.com/stm32mpu/wiki/STM32MP1\\_Developer\\_Package](https://wiki.st.com/stm32mpu/wiki/STM32MP1_Developer_Package)

Follow the next steps to install the SDK:

1. Uncompress the tarball file to get the SDK installation script and make it executable.

```
PC:~$ mkdir -p $HOME/STM32MPU_workspace/tmp
```

```
PC:~$ mkdir -p $SDK_ROOT/SDK
```

```
PC:~/STM32MPU_workspace/tmp$ tar xvf en.SDK-x86_64-stm32mp1-openstlinux-5.4-dunfell-mp1-20-06-24.tar.xz
```

```
PC:~/STM32MPU_workspace/tmp$ chmod +x stm32mp1-openstlinux-5.4-dunfell-mp1-20-06-24/sdk/st-image-weston-openstlinux-weston-stm32mp1-x86_64-toolchain-3.1-openstlinux-5.4-dunfell-mp1-20-06-24.sh
```

2. Add the following line to .bashrc.

```
PC:~$ echo "export SDK_ROOT=$HOME/STM32MP15-Ecosystem-v2.0.0/Developer-Package" >> $HOME/.bashrc
```

3. Install the SDK.

```
PC:~/STM32MPU_workspace/tmp$ ./stm32mp1-openstlinux-5.4-dunfell-mp1-20-06-24/sdk/st-image-weston-openstlinux-weston-stm32mp1-x86_64-toolchain-3.1-openstlinux-5.4-dunfell-mp1-20-06-24.sh -d $SDK_ROOT/SDK
ST OpenSTLinux - Weston - (A Yocto Project Based Distro) SDK installer version 3.1-openstlinux-5.4-dunfell-mp1-20-06-24
```

```

=====
=====
You are about to install the SDK to "/home/alberto/STM32MP15-Ecosystem-
v2.0.0/Developer-Package/SDK". Proceed [Y/n]? y
Extracting
SDK.....
.....done
Setting it up...done
SDK has been successfully set up and is ready to be used.

Each time you wish to use the SDK in a new shell session, you need to source the
environment setup script:

PC:~$ source $SDK_ROOT/SDK/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-
linux-gnueabi

```

## Installing and compiling the Linux kernel for the developer package

To download the STM32MP1 Linux kernel for the STM32MP15-Ecosystem-v2.0.0 release follow the steps indicated in the STM32 MPU wiki section located at [https://wiki.st.com/stm32mpu/wiki/STM32MP1\\_Developer\\_Package](https://wiki.st.com/stm32mpu/wiki/STM32MP1_Developer_Package)

Follow the next steps to install and compile the Linux kernel:

1. Extract the kernel source code.

```

PC:~$ cd $SDK_ROOT

PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package$ tar xvf en.SOURCES-kernel-
stm32mp1-openstlinux-5-4-dunfell-mp1-20-06-24.tar.xz

PC:~$ cd $SDK_ROOT/stm32mp1-openstlinux-5.4-dunfell-mp1-20-06-24/sources/arm-
ostl-linux-gnueabi/linux-stm32mp-5.4.31-r0

PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-r0$
tar xvf linux-5.4.31.tar.xz

PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-r0$ cd
linux-5.4.31

```

2. To initialize a pad in GPIO mode with a bias (internal pull-up, pull-down..), it is needed to disable the strict mode of pinctrl. You have to change the strict variable of the struct pinmux\_ops to false. You can find within the kernel sources the struct pinmux\_ops structure; it is included in the /drivers/pinctrl/stm32/pinctrl-stm32.c file.

```

static const struct pinmux_ops stm32_pmx_ops = {
    .get_functions_count = stm32_pmx_get_funcs_cnt,
    .get_function_name   = stm32_pmx_get_func_name,

```

```

        .get_function_groups = stm32_pmx_get_func_groups,
        .set_mux             = stm32_pmx_set_mux,
        .gpio_set_direction  = stm32_pmx_gpio_set_direction,
        .strict               = false,
};

```

3. Prepare and configure kernel source code.

```

PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-
r0/linux-5.4.31$ for p in `ls -1 ../*.patch`; do patch -p1 < $p; done

```

```

PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-
r0/linux-5.4.31$ make multi_v7_defconfig fragment*.config

```

```

PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-
r0/linux-5.4.31$ for f in `ls -1 ../fragment*.config`; do
scripts/kconfig/merge_config.sh -m -r .config $f; done

```

```

PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-
r0/linux-5.4.31$ yes '' | make ARCH=arm oldconfig

```

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

```

PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-
r0/linux-5.4.31$ make ARCH=arm menuconfig

```

```

Device drivers >
  <*> Industrial I/O support --->
    *- Enable buffer support within IIO
    *- Industrial I/O buffering based on kfifo
  <*> Enable IIO configuration via configfs
  *- Enable triggered sampling support
  <*> Enable software IIO device support
  <*> Enable software triggers support
    Triggers - standalone --->
      <*> High resolution timer trigger
      <*> SYSFS trigger

```

```

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
```

5. Compile kernel source code and kernel modules.

```
PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-  
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-  
r0/linux-5.4.31$ make -j4 ARCH=arm uImage vmlinux dtbs LOADADDR=0xC2000040
```

```
PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-  
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-  
r0/linux-5.4.31$ make ARCH=arm modules
```

```
PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-  
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-  
r0/linux-5.4.31$ mkdir -p $PWD/install_artifact/
```

```
PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-  
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-  
r0/linux-5.4.31$ make ARCH=arm INSTALL_MOD_PATH="$PWD/install_artifact"  
modules_install
```

6. Boot the STM32MP1 target and open a new terminal on the host, for example "minicom". Set the following configuration: "115.2 kbaud, 8 data bits, 1 stop bit, no parity".

```
PC:~$ minicom -D /dev/ttyACM0
```

7. Connect Ethernet cable between host and eval board and verify the connection.

```
root@stm32mp1:~# ifconfig eth0 down
```

```
root@stm32mp1:~# ifconfig eth0 up
```

```
root@stm32mp1:~# ifconfig eth0 10.0.0.10
```

```
root@stm32mp1:~# ping 10.0.0.1
```

8. Deploy the compiled Linux kernel image and the kernel modules to the target STM32MP1 device.

```
PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-  
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-  
r0/linux-5.4.31$ scp arch/arm/boot/uImage root@10.0.0.10:/boot
```

```
PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-  
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-  
r0/linux-5.4.31$ rm install_artifact/lib/modules/5.4.31/build  
install_artifact/lib/modules/5.4.31/source
```

```
PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-r0/linux-5.4.31$ find install_artifact/ -name "*.ko" | xargs $STRIP --strip-debug --remove-section=.comment --remove-section=.note --preserve-dates
```

```
PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-r0/linux-5.4.31$ scp -r install_artifact/lib/modules/* root@10.0.0.10:/lib/modules
```

9. Re-generate the list of module dependencies (modules.dep) and the list of symbols provided by modules (modules.symbols), synchronize data on disk with memory and reboot the board.

```
root@stm32mp1:~# /sbin/depmod -a
```

```
root@stm32mp1:~# sync
```

```
root@stm32mp1:~# modinfo vivid
```

```
root@stm32mp1:~# reboot
```

## Compile and deploy the Linux kernel drivers

Download the linux\_5.4\_stm32mp1\_drivers.zip file from the github of the book and unzip it in the STM32MP15-Ecosystem-v2.0.0 folder of the Linux host:

```
PC:~$ cd ~/STM32MP15-Ecosystem-v2.0.0/
```

Compile and deploy the drivers to the **STM32MP157C-DK2** Discovery kit:

```
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_stm32mp1_drivers$ make
```

```
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_stm32mp1_drivers$ make deploy
```

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

adx1345_stm32mp1.ko	100%	12KB	12.3KB/s	00:00
adx1345_stm32mp1_iio.ko	100%	12KB	12.4KB/s	00:00
hellokeys_stm32mp1.ko	100%	7024	6.9KB/s	00:00
helloworld_stm32mp1.ko	100%	4008	3.9KB/s	00:00
helloworld_stm32mp1_char_driver.ko	100%	6184	6.0KB/s	00:00
helloworld_stm32mp1_class_driver.ko	100%	7724	7.5KB/s	00:00
helloworld_stm32mp1_with_parameters.ko	100%	4604	4.5KB/s	00:00
helloworld_stm32mp1_with_timing.ko	100%	5688	5.6KB/s	00:00
i2c_stm32mp1_accel.ko	100%	7216	7.1KB/s	00:00
int_stm32mp1_key.ko	100%	7812	7.6KB/s	00:00
int_stm32mp1_key_wait.ko	100%	10KB	9.9KB/s	00:00
io_stm32mp1_expander.ko	100%	9664	9.4KB/s	00:00
keyled_stm32mp1_class.ko	100%	16KB	16.2KB/s	00:00
ledRGB_stm32mp1_class_platform.ko	100%	9524	9.3KB/s	00:00
ledRGB_stm32mp1_platform.ko	100%	11KB	10.9KB/s	00:00

led_stm32mp1_UIO_platform.ko	100%	6912	6.8KB/s	00:00
linkedlist_stm32mp1_platform.ko	100%	9460	9.2KB/s	00:00
ltc2422_stm32mp1_dual.ko	100%	7344	7.2KB/s	00:00
ltc2422_stm32mp1_trigger.ko	100%	9840	9.6KB/s	00:00
ltc2607_stm32mp1_dual_device.ko	100%	8056	7.9KB/s	00:00
ltc3206_stm32mp1_led_class.ko	100%	11KB	11.1KB/s	00:00
misc_stm32mp1_driver.ko	100%	5780	5.6KB/s	00:00
sdma_stm32mp1_m2m.ko	100%	12KB	11.7KB/s	00:00
sdma_stm32mp1_mmap.ko	100%	12KB	11.7KB/s	00:00

~/STM32MP15-Ecosystem-v2.0.0/linux\_5.4\_stm32mp1\_drivers\$

Verify that the drivers are now in the STM32MP157C-DK2 Discovery kit:

```

root@stm32mp1:~# ls
adxl345_stm32mp1.ko          keyedled_stm32mp1_class.ko
adxl345_stm32mp1_iio.ko     ledRGB_stm32mp1_class_platform.ko
hellokeys_stm32mp1.ko       ledRGB_stm32mp1_platform.ko
helloworld_stm32mp1.ko      led_stm32mp1_UIO_platform.ko
helloworld_stm32mp1_char_driver.ko linkedlist_stm32mp1_platform.ko
helloworld_stm32mp1_class_driver.ko ltc2422_stm32mp1_dual.ko
helloworld_stm32mp1_with_parameters.ko ltc2422_stm32mp1_trigger.ko
helloworld_stm32mp1_with_timing.ko ltc2607_stm32mp1_dual_device.ko
i2c_stm32mp1_accel.ko       ltc3206_stm32mp1_led_class.ko
int_stm32mp1_key.ko         misc_stm32mp1_driver.ko
int_stm32mp1_key_wait.ko    sdma_stm32mp1_m2m.ko
io_stm32mp1_expander.ko     sdma_stm32mp1_mmap.ko
root@stm32mp1:~#

```

The stm32mp15xx-dkx.dtsi and stm32mp15-pinctrl.dtsi files with all the needed modifications to run the drivers are stored in the device\_tree folder inside the linux\_5.4\_stm32mp1\_drivers.zip file.

During the development of the drivers you will modify these device tree files, then build and copy them to the STM32MP1 board.

```

PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-r0/linux-
5.4.31$ make dtbs

```

```

PC:~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-
dunfell-mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-r0/linux-
5.4.31$ scp arch/arm/boot/dts/stm32mp157c-dk2.dtb root@10.0.0.10:/boot

```

## Hardware and device tree descriptions for the ST STM32MP1 labs

In the next sections it will be described the different hardware and device tree configurations for the labs where external hardware connected to the processor is controlled by the drivers. The

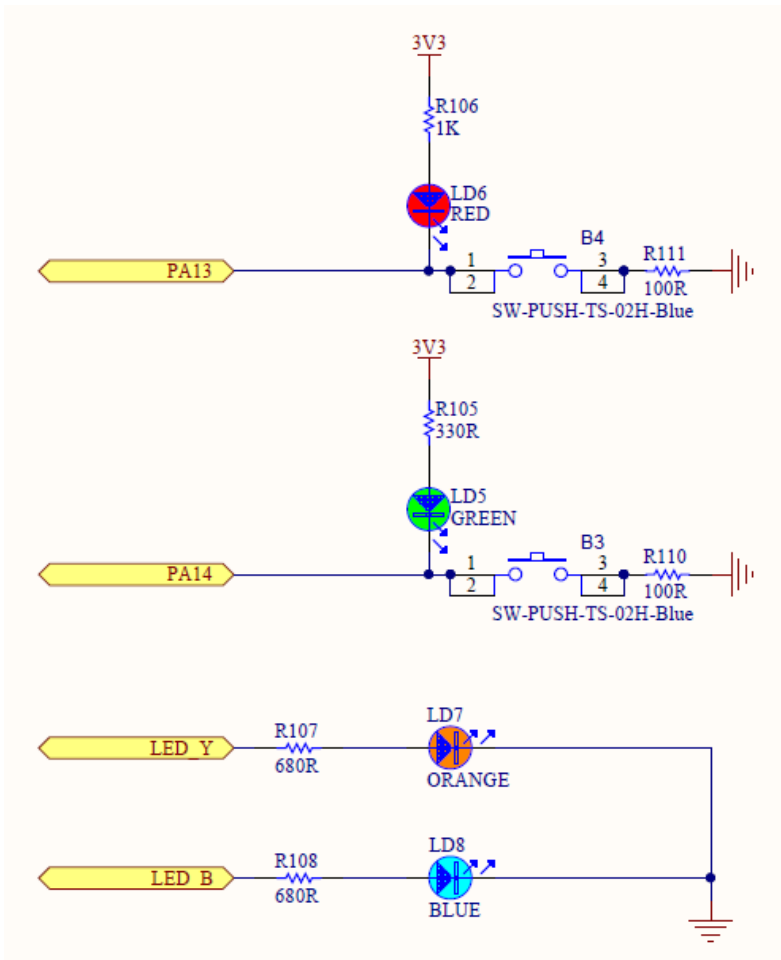


schematic of the STM32MP157C-DK2 Discovery kit is included inside the linux\_5.4\_stm32mp1\_drivers.zip file that can be downloaded from the github of the book.

## LAB 5.2, 5.3 and 5.4 hardware and device tree descriptions

During the development of these drivers you will use the LD6 RED, LD5 GREEN and LD8 BLUE leds included in the STM32MP157C-DK2 Discovery kit. Go to the pag.13 of the schematic to see them. Each LED is individually controlled by a processor pin programmed as GPIO output. The pins are PA13, PA14, and PD11. The PD11 pin is used by the “gpio-leds” driver, therefore you’ll have to disable it in the stm32mp15xx-dkx.dtsi file to avoid conflicts with your developed drivers.

```
/*led {  
    compatible = "gpio-leds";  
    blue {  
        label = "heartbeat";  
        gpios = <&gpiod 11 GPIO_ACTIVE_HIGH>;  
        linux,default-trigger = "heartbeat";  
        default-state = "off";  
    };  
};*/
```



This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the driver for the LAB 5.2:

```
ledRGB {
    compatible = "arrow,RGBleds";
    clocks = <&rcc GPIOA>,
            <&rcc GPIOD>;

    clock-names = "GPIOA", "GPIOD";

    red {
        label = "ledred";
```

```

};

green {
    label = "ledgreen";
};

blue {
    label = "ledblue";
};
};

```

This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the driver for the LAB 5.3:

```

ledclassRGB {
    compatible = "arrow,RGBclasssleds";
    reg = <0x50002000 0x400>,
        <0x50005000 0x400>;

    clocks = <&rcc GPIOA>,
        <&rcc GPIOD>;

    clock-names = "GPIOA", "GPIOD";

    red {
        label = "ledred";
    };

    green {
        label = "ledgreen";
    };

    blue {
        label = "ledblue";
    };
};

```

This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the driver for the LAB 5.4:

```

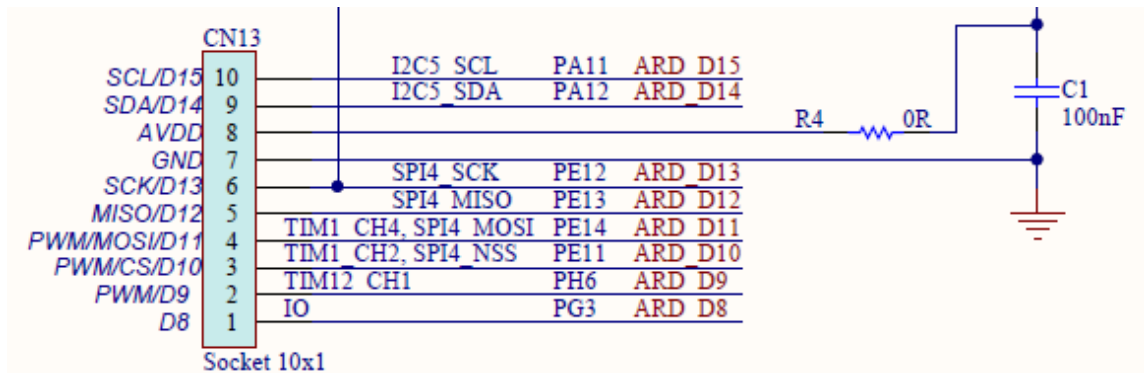
UIO {
    compatible = "arrow,UIO";
    reg = <0x50002000 0x1000>;
    clocks = <&rcc GPIOA>;
};

```

## LAB 6.1 hardware and device tree descriptions

In this lab the driver will be able to manage several PCF8574 I/O expander devices connected to the I2C bus. You can use one of the multiples boards based on this device to develop this lab, for example, the next one <https://www.waveshare.com/pcf8574-io-expansion-board.htm>.

You will take the I2C5 bus from the CN13 connector of the STM32MP157C-DK2 Discovery kit. Go to the pag.10 of the STM32MP157C-DK2 schematic to see the connector.



You can take the 3V3 and GND signals from the CN16 connector of the STM32MP157C-DK2 board. Go to the pag.10 of the STM32MP157C-DK2 schematic to see the connector.

This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the driver for the LAB 6.1:

```
&i2c5 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c5_pins_a>;
    pinctrl-1 = <&i2c5_pins_sleep_a>;
    i2c-scl-rising-time-ns = <185>;
    i2c-scl-falling-time-ns = <20>;
    clock-frequency = <400000>;
    /delete-property/dmas;
    /delete-property/dma-names;
    status = "okay";

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

```
};  
  
};
```

## LAB 6.2 hardware and device tree descriptions

To test this driver you will use the DC749A - Demo Board (<http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/dc749a.html>).

In this lab you will use the I2C5 pins of the STM32MP157C-DK2 CN13 connector to connect to the DC749A - Demo Board. Connect the pin 9 (I2C5\_SDA) of the CN13 connector to the pin 7 (SDA) of the DC749A J1 connector and the pin 10 (I2C5\_SCL) of the CN13 connector to the pin 4 (SCL) of the DC749A J1 connector. Connect the 3.3V pin from the STM32MP157C-DK2 CN16 connector to the DC749A Vin J2 pin and to the DC749A J20 DVCC connector. Connect the pin 1 (PG3 pad) of the CN13 connector to the pin 6 (ENRGB/S) of the DC749A J1 connector. Do not forget to connect GND between the two boards.

This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the driver for the LAB 6.2:

```
&i2c5 {  
    pinctrl-names = "default", "sleep";  
    pinctrl-0 = <&i2c5_pins_a>;  
    pinctrl-1 = <&i2c5_pins_sleep_a>;  
    i2c-scl-rising-time-ns = <185>;  
    i2c-scl-falling-time-ns = <20>;  
    clock-frequency = <400000>;  
    /delete-property/dmas;  
    /delete-property/dma-names;  
    status = "okay";  
  
    ltc3206: ltc3206@1b {  
        compatible = "arrow,ltc3206";  
        reg = <0x1b>;  
        gpios = <&gpio3 3 GPIO_ACTIVE_LOW>;  
  
        led1r {  
            label = "red";  
        };  
  
        led1b {  
            label = "blue";  
        };  
  
        led1g {  
            label = "green";  
        };  
    };  
};
```

```

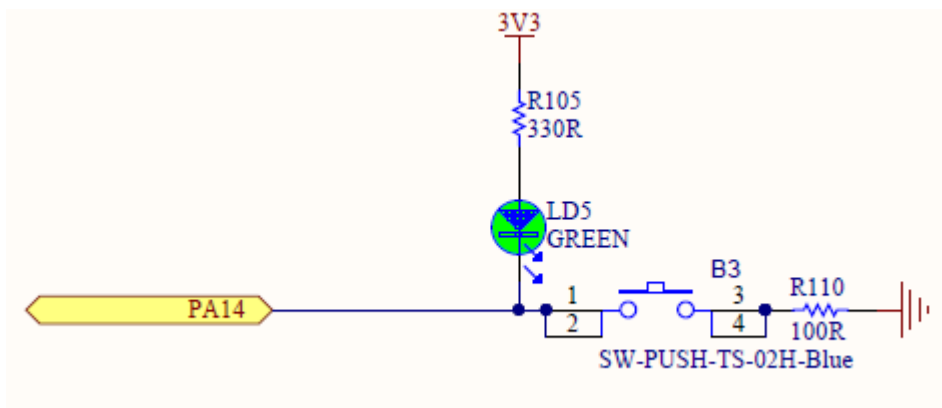
    ledmain {
        label = "main";
    };

    ledsub {
        label = "sub";
    };
};

```

## LAB 7.1 and 7.2 hardware and device tree descriptions

In these two labs you will use the “USER” button (B3) of the STM32MP157C-DK2 board. The button is connected to PA14 pin. The pin will be programmed as an input generating an interrupt. You will also have to ensure the mechanical key is debounced. Open the STM32MP157C-DK2 schematic and find the button B3 in pag.13.



These are the device tree nodes that should be included in the stm32mp15xx-dkx.dtsi file to run the drivers for the LAB 7.1 and the LAB 7.2:

```

int_key {
    compatible = "arrow,intkey";
    pinctrl-names = "default";
    pinctrl-0 = <&key_pins>;
    label = "PB_USER";
    gpios = <&gpioa 14 GPIO_ACTIVE_LOW>;
    interrupt-parent = <&gpioa>;
    interrupts = <14 IRQ_TYPE_EDGE_FALLING>;
};

```

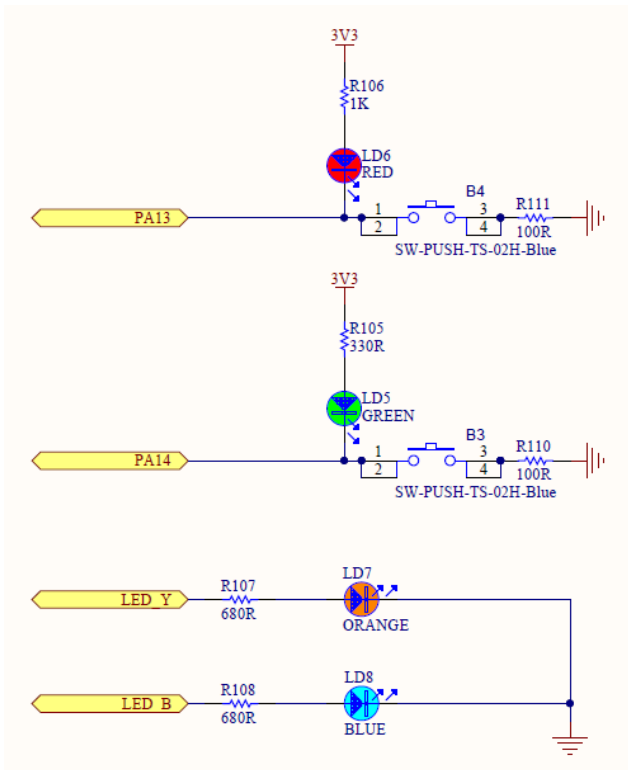
```

int_key_wait {
    compatible = "arrow,intkeywait";
    pinctrl-names = "default";
    pinctrl-0 = <&key_pins>;
    label = "PB_USER";
    gpios = <&gpioa 14 GPIO_ACTIVE_LOW>;
    interrupt-parent = <&gpioa>;
    interrupts = <14 IRQ_TYPE_EDGE_FALLING>;
};

```

## LAB 7.3 hardware and device tree descriptions

In this lab you will use the LD7 ORANGE and the LD8 BLUE leds included in the STM32MP157C-DK2 Discovery kit. Go to the pag.13 of the STM32MP157C-DK2 schematic to see them. Each LED is individually controlled by a processor pin programmed as GPIO output. The pins are PH7 and PD11. Currently the PD11 pin is used by the “gpio-leds” driver, therefore you’ll have to disable it in the stm32mp15xx-dkx.dtsi file. In this lab you will also use the buttons B4 and B3. The button B4 is connected to PA13 pin and the button B3 is connected to the PA14 pin. Both pins will be programmed as an input generating an interrupt. You will also have to ensure the mechanical key is debounced. Open the STM32MP157C-DK2 schematic and find the B4 and B3 buttons in pag.13.



This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the driver for the LAB 7.3:

```
ledpwm {
    compatible = "arrow,ledpwm";

    pinctrl-names = "default";
    pinctrl-0 = <&keyleds_pins>;

    bp1 {
        label = "KEY_1";
        gpios = <&gpioa 13 GPIO_ACTIVE_LOW>; // B4:USER2
        trigger = "falling";
    };

    bp2 {
        label = "KEY_2";
        gpios = <&gpioa 14 GPIO_ACTIVE_LOW>; // B3:USER1
    };
}
```



```

        trigger = "falling";
    };

    ledorange {
        label = "led";
        colour = "orange";
        gpios = <&gpioh 7 GPIO_ACTIVE_LOW>;
    };

    ledblue {
        label = "led";
        colour = "blue";
        gpios = <&gpiod 11 GPIO_ACTIVE_LOW>;
    };

};

```

This is the device tree node that should be included in the stm32mp15-pinctrl.dtsi file to run the driver for the LAB 7.3:

```

keyleds_pins: keyleds-0 {
    pins1 {
        pinmux = <STM32_PINMUX('H', 7, GPIO)>,
                <STM32_PINMUX('D', 11, GPIO)>;
        drive-push-pull;
        bias-pull-down;
    };

    pins2 {
        pinmux = <STM32_PINMUX('A', 13, GPIO)>;
        drive-push-pull;
        bias-pull-up;
    };

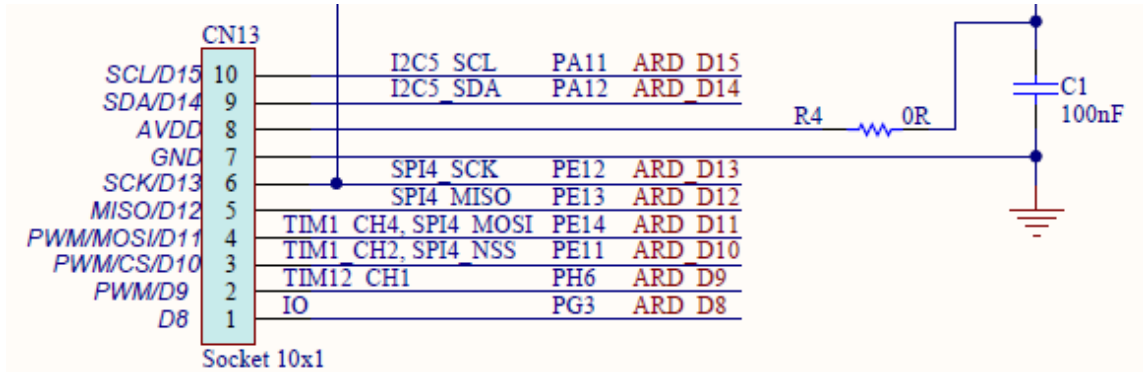
    pins3 {
        pinmux = <STM32_PINMUX('A', 14, GPIO)>;
        drive-push-pull;
        bias-pull-up;
    };
};

```

## LAB 10.1,10.2 and 12.1 hardware and device tree descriptions

In these labs you will control an accelerometer board connected to the I2C and SPI buses of the processor. You will use the ADXL345 Accel click mikroBUS™ accessory board to develop the drivers; you will access to the schematic of the board at <http://www.mikroe.com/click/accel/>.

For the LAB 10.1 you will connect the accelerometer board to the I2C5 pins of the STM32MP157C-DK2 CN13 connector. For the LAB 10.2 and the LAB 12.1 you will connect the accelerometer board to the SPI4 pins of the CN13 connector.



The pin 1 of the CN13 connector (PG3 pad) will be programmed as an input generating an interrupt for the LAB 10.2 and the LAB 12.1.

This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the driver for the LAB 10.1:

```
&i2c5 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c5_pins_a>;
    pinctrl-1 = <&i2c5_pins_sleep_a>;
    i2c-scl-rising-time-ns = <185>;
    i2c-scl-falling-time-ns = <20>;
    clock-frequency = <400000>;
    /delete-property/dmas;
    /delete-property/dma-names;
    status = "okay";

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

This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the drivers for the LAB 10.2 and the LAB 12.1:

```
&spi4 {
    pinctrl-names = "default", "sleep";
```

```

pinctrl-0 = <&spi4_pins_a>;
pinctrl-1 = <&spi4_sleep_pins_a>;
cs-gpios = <&gpioe 11 0>;
status = "okay";

Accel: ADXL345@0 {
    compatible = "arrow,adxl345";
    pinctrl-names = "default";
    pinctrl-0 = <&accel_pins>;
    spi-max-frequency = <5000000>;
    spi-cpol;
    spi-cpha;
    reg = <0>;
    int-gpios = <&gpiog 3 GPIO_ACTIVE_LOW>;
    interrupt-parent = <&gpiog>;
    interrupts = <3 IRQ_TYPE_LEVEL_HIGH>;
};
};

```

This is the device tree node that should be included in the stm32mp15-pinctrl.dtsi file to run the drivers for the LAB 10.2 and the LAB 12.1:

```

accel_pins: accel-0 {
    pins {
        pinmux = <STM32_PINMUX('G', 3, GPIO)>;
        drive-push-pull;
        bias-pull-down;
    };
};

```

## LAB 11.1 hardware and device tree descriptions

In this lab you will control the Analog Devices LTC2607 internal DACs individually or both DACA + DACB in a simultaneous mode. You will use the DC934A evaluation board; you can download the schematics at

<https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/dc934a.html>

For this LAB 11.1 you will connect the I2C5 pins of the STM32MP157C-DK2 CN13 connector to the SDA and SCL pins of the LTC2607 DC934A evaluation board. You are going to power the LTC2607 with the 3.3V pin of the STM32MP157C-DK2 CN16 connector, connecting it to V+, pin 1 of the DC934A's connector J1. Also connect GND between the DC934A (i.e., pin 3 of connector J1) and GND pin of the STM32MP157C-DK2 Discovery kit.

This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the driver for the LAB 11.1:

```

&i2c5 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c5_pins_a>;
    pinctrl-1 = <&i2c5_pins_sleep_a>;
    i2c-scl-rising-time-ns = <185>;
    i2c-scl-falling-time-ns = <20>;
    clock-frequency = <400000>;
    /delete-property/dmas;
    /delete-property/dma-names;
    status = "okay";

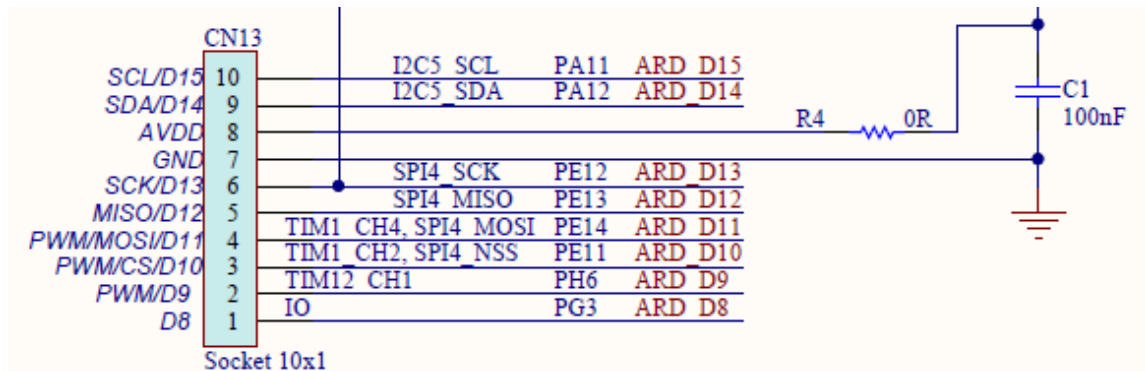
    ltc2607@72 {
        compatible = "arrow,ltc2607";
        reg = <0x72>;
    };

    ltc2607@73 {
        compatible = "arrow,ltc2607";
        reg = <0x73>;
    };
};

```

## LAB 11.2, LAB 11.3 and LAB 11.4 hardware and device tree descriptions

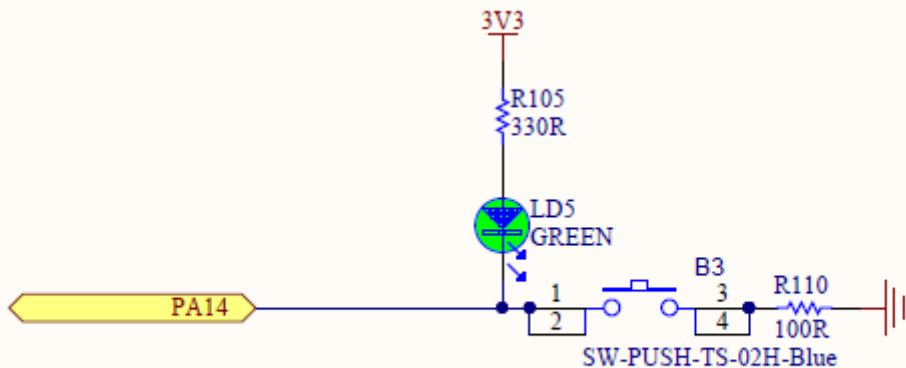
In these three labs you will reuse the hardware description of the LAB 11.1 and will use the SPI4 pins of the STM32MP157C-DK2 CN13 connector to connect to the LTC2422 dual ADC SPI device that is included in the DC934A board.



Open the STM32MP157C-DK2 schematic to see the CN13 connector and look for the SPI pins. The CS, SCK and MISO (Master In, Slave Out) signals will be used. The MOSI (Master out, Slave in) signal won't be needed, as you are only going to receive data from the LTC2422 device. Connect the next CN13 SPI4 pins to the LTC2422 SPI ones obtained from the DC934A board J1 connector:

- Connect the STM32MP157C-DK2 **SPI4\_NSS** (CS) to LTC2422 **CS**
- Connect the STM32MP157C-DK2 **SPI4\_SCK** (SCK) to LTC2422 **SCK**
- Connect the STM32MP157C-DK2 **SPI4\_MISO** (MISO) to LTC2422 **MISO**

In the lab 11.4 you will also use the "USER" button (B3). The button is connected to the PA14 pin. The pin will be programmed as an input generating an interrupt.



These are the device tree nodes that should be included in the stm32mp15xx-dkx.dtsi file to run the drivers for the LAB 11.2, LAB 11.3 and LAB 11.4:

```
&spi4 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&spi4_pins_a>;
    pinctrl-1 = <&spi4_sleep_pins_a>;
    cs-gpios = <&gpioe 11 0>;
    status = "okay";

    /* spidev@0 {
        compatible = "spidev";
        spi-max-frequency = <2000000>;
        reg = <0>;
```

```

    }; */

    ADC: ltc2422@0 {
        compatible = "arrow,ltc2422";
        spi-max-frequency = <2000000>;
        reg = <0>;
    };

    ADC: ltc2422@0 {
        compatible = "arrow,ltc2422";
        spi-max-frequency = <2000000>;
        reg = <0>;
        pinctrl-names = "default";
        pinctrl-0 = <&key_pins>;
        int-gpios = <&gpioa 14 GPIO_ACTIVE_LOW>;
    };
};

```

This is the device tree node that should be included in the stm32mp15-pinctrl.dtsi file to run the driver for the LAB 11.4:

```

key_pins: key-0 {
    pins {
        pinmux = <STM32_PINMUX('A', 14, GPIO)>;
        drive-push-pull;
        bias-pull-up;
    };
};

```

The kernel 5.4 modules developed for the STM32MP157C-DK2 board are included in the linux\_5.4\_STM32MP1\_drivers.zip file and can be downloaded from the GitHub repository at [https://github.com/ALIBERA/linux\\_book\\_2nd\\_edition](https://github.com/ALIBERA/linux_book_2nd_edition)

## 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 application to control GPIOs from user space.

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

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

The hardware platforms used in this lab are the STM32MP157C-DK2 board from ST and the PIXI™ CLICK from MIKROE. The documentation of these boards can be found at <https://www.st.com/en/evaluation-tools/stm32mp157c-dk2.html> 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

Configure

Cancel

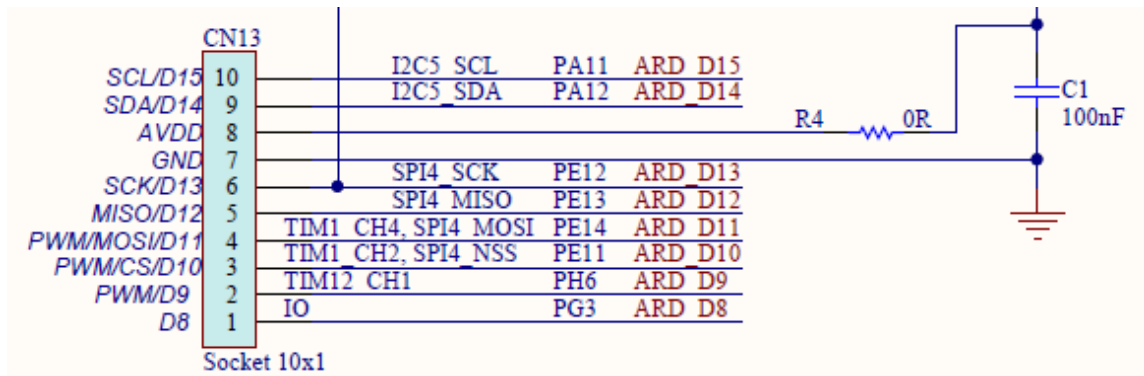
Not all the MAX11300 specifications were included during the development of this driver. These are the main specifications that have been included:

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


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

## LAB 11.5 hardware description

In this lab you will use the SPI4 pins of the STM32MP157C-DK2 CN13 connector to connect to the PIXI™ CLICK mikroBUS™ socket. See below the STM32MP157C-DK2 CN13 connector:



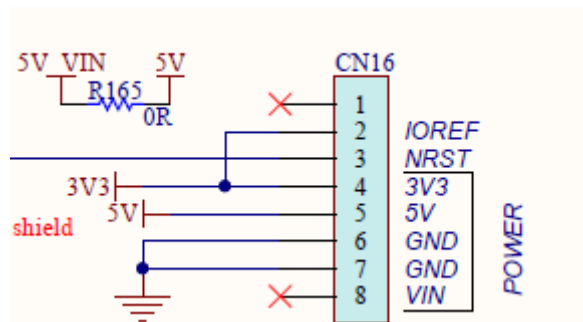
And the PIXI™ CLICK mikroBUS™ socket:

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

Open the STM32MP157C-DK2 schematic to see the CN13 connector and look for the SPI pins. The CS, SCK and MISO (Master In, Slave Out) and MOSI (Master out, Slave in) signals will be used. Connect the next CN13 SPI4 pins to the MAX11300 SPI ones obtained from the PIXI™ CLICK mikroBUS™ socket:

- Connect the STM32MP157C-DK2 **SPI4\_NSS** (Pin 3 of CN13) to MAX11300 **CS** (Pin 3 of Mikrobus)
- Connect the STM32MP157C-DK2 **SPI4\_SCK** (Pin 6 of CN13) to MAX11300 **SCK** (Pin 4 of Mikrobus)
- Connect the STM32MP157C-DK2 **SPI4\_MOSI** (Pin 4 of CN13) to MAX11300 **MOSI** (Pin 6 of Mikrobus)
- Connect the STM32MP157C-DK2 **SPI4\_MISO** (Pin 5 of CN13) to MAX11300 **MISO** (Pin 5 of Mikrobus)
- Connect STM32MP157C-DK2 **GND** (Pin 7 of CN13) to MAX11300 **GND** (Pin 9 of Mikrobus)

Now, find the CN16 connector in the STM32MP157C-DK2 schematic:

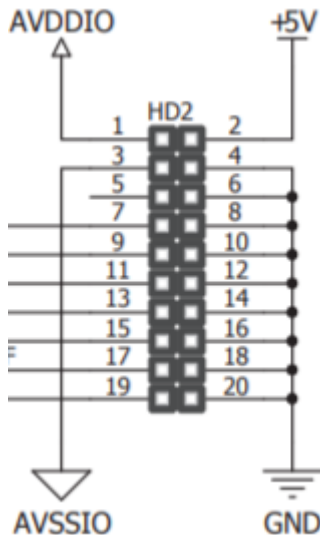


And connect the next power pins between the two boards:

- Connect the Pin 4 of CN16 (3.3V) to MAX11300 3.3V (Pin 7 of Mikrobus)
- Connect the Pin 5 of CN16 (5V) to MAX11300 5V (Pin 10 of Mikrobus)
- Connect the Pin 6 of CN16 (GND) to MAX11300 GND (Pin 9 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 `stm32mp15xx-dkx.dtsi` DT file and find the `spi4` controller master node. Inside the `spi4` node, you can see the `pinctrl` properties which configure the pins in SPI mode when the system runs and into a different state (ANALOG) when the system suspends to RAM. Both `spi4_pins_a` and `spi4_sleep_pins_a` are already defined in the `stm32mp15-pinctrl.dtsi` file.

The `cs-gpios` property specifies the gpio pins to be used for chip selects. In the `spi4` node, you can see that there is only one chip select enabled. The `spi4` controller is enabled by writing "okay" to the status property. Comment out all the sub-nodes included in the `spi4` node coming from previous labs.

Now, you will add to the `spi4` 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:

```
&spi4 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&spi4_pins_a>;
    pinctrl-1 = <&spi4_sleep_pins_a>;
    cs-gpios = <&gpioe 11 0>;
    status = "okay";

    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>;
    };
};

```

```

/* spidev@0 {
    compatible = "spidev";
    spi-max-frequency = <2000000>;
    reg = <0>;
}; */

/*Accel: ADXL345@0 {
    compatible = "arrow,adxl345";
    pinctrl-names = "default";
    pinctrl-0 = <&accel_pins>;
    spi-max-frequency = <5000000>;
    spi-cpol;
    spi-cpha;
    reg = <0>;
    int-gpios = <&gpio3 3 GPIO_ACTIVE_LOW>;
    interrupt-parent = <&gpio3>;
    interrupts = <3 IRQ_TYPE_LEVEL_HIGH>;
};*/

/*ADC: ltc2422@0 {
    compatible = "arrow,ltc2422";
    spi-max-frequency = <2000000>;
    reg = <0>;
};

ADC: ltc2422@0 {
    compatible = "arrow,ltc2422";
    spi-max-frequency = <2000000>;
    reg = <0>;
    pinctrl-names = "default";
    pinctrl-0 = <&key_pins>;
    int-gpios = <&gpioa 14 GPIO_ACTIVE_LOW>;
};*/
};

```

You also have to include the next header file at the beginning of the stm32mp15xx-dkx.dtsi DT file.

```
#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:

```

~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/stm32mp1-openstlinux-5.4-dunfell-
mp1-20-06-24/sources/arm-ostl-linux-gnueabi/linux-stm32mp-5.4.31-r0/linux-
5.4.31/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 matches with the compatible property of the max11300 DT node:

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

5. Define an array of 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.

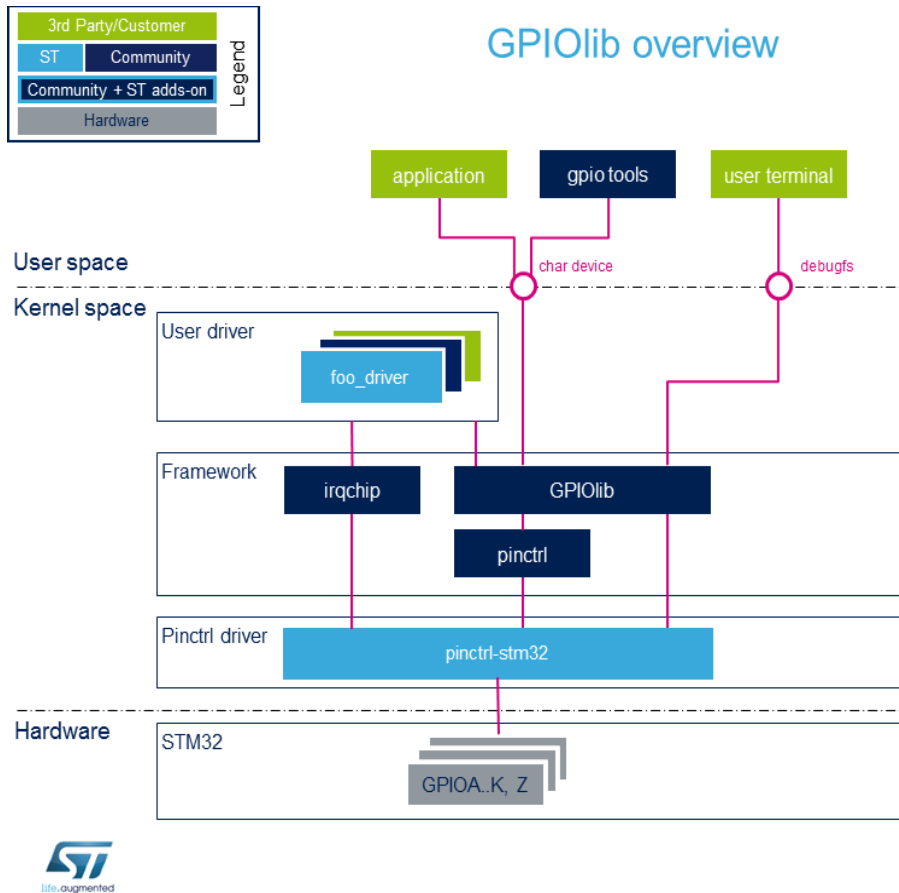
Most processors today use composite pin controllers. These composite pin controllers will control the GPIOs of the processor, generate interrupts on top of the GPIO functionality and allow pin multiplexing using the I/O pins of the processor as GPIOs or as one of several peripheral functions. The STM32MP1 from ST is one of these processors, including composite pin controllers, which are configured with the `pinctrl-stm32` driver:

<https://elixir.bootlin.com/linux/v5.4.64/source/drivers/pinctrl/stm32>

The `pinctrl-stm32` 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-stm32` 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](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 output level of the GPIO lines (low or high output level).

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

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

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

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

```
static int max11300_gpio_init(struct max11300_state *st)
{
    st->gpiochip.label = "gpio-max11300";
    st->gpiochip.base = -1;
    st->gpiochip.ngpio = st->num_gpios;
    st->gpiochip.parent = st->dev;
    st->gpiochip.can_sleep = true;
    st->gpiochip.direction_input = max11300_gpio_direction_input;
    st->gpiochip.direction_output = max11300_gpio_direction_output;
    st->gpiochip.get = max11300_gpio_get;
    st->gpiochip.set = max11300_gpio_set;
    st->gpiochip.owner = THIS_MODULE;

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

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

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

    mutex_lock(&st->gpio_lock);

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

```

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

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

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

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

    val = (int) read_val;

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

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

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

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

```



```

{
    struct max11300_state *st = gpiochip_get_data(chip);
    u8 reg;
    unsigned int val = 0;

    mutex_lock(&st->gpio_lock);

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

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

    mutex_unlock(&st->gpio_lock);
}

/*

```

```

* struct gpio_chip direction_input callback function.
* It configures the GPIO port as an input (GPI)
* writing to the PORT_CFG register of the max11300
*/
static int max11300_gpio_direction_input(struct gpio_chip *chip,
                                         unsigned int offset)
{
    struct max11300_state *st = gpiochip_get_data(chip);
    int ret;
    u8 reg;
    u16 port_mode, val;

    mutex_lock(&st->gpio_lock);

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

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

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

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

    mdelay(1);

err_unlock:
    mutex_unlock(&st->gpio_lock);

    return ret;
}

/*
* struct gpio_chip direction_output callback function.
* It configures the GPIO port as an output (GPO) writing to
* the PORT_CFG register of the max11300 and 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 STM32MP1 processor.

**Note:** The " IIO Mixed-Signal I/O Device" driver source code developed for the STM32MP157C-DK2 board is included in the linux\_5.4\_max11300\_driver.zip file and can be downloaded from the GitHub repository at [https://github.com/ALIBERA/linux\\_book\\_2nd\\_edition](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 output is set high and port_number is %d. Pin is > 0x0F\n",
            st->gpio_offset[offset]);
        val |= BIT(st->gpio_offset[offset]);
        val = val >> 16;
        reg = GPO_DATA_19_TO_16_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 0 && (st->gpio_offset[offset] > 0x0F)) {
        dev_info(st->dev,
            "The GPIO output is set low and port_number is %d. Pin is > 0x0F\n",
            st->gpio_offset[offset]);
        val &= ~BIT(st->gpio_offset[offset]);
        val = val >> 16;
        reg = GPO_DATA_19_TO_16_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 1 && (st->gpio_offset[offset] < 0x0F)) {
        dev_info(st->dev,
            "The GPIO output is set high and port_number is %d. Pin is < 0x0F\n",
            st->gpio_offset[offset]);
        val |= BIT(st->gpio_offset[offset]);
        reg = GPO_DATA_15_TO_0_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 0 && (st->gpio_offset[offset] < 0x0F)) {
        dev_info(st->dev,
            "The GPIO output is set low and port_number is %d. Pin is < 0x0F\n",
            st->gpio_offset[offset]);
        val &= ~BIT(st->gpio_offset[offset]);
        reg = GPO_DATA_15_TO_0_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else
        dev_info(st->dev, "the gpio %d cannot accept this value\n", offset);

    mutex_unlock(&st->gpio_lock);
}

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

```

[illegible]

```

    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 = IIIO_VOLTAGE;
    chan->indexed = 1;
    chan->address = port_mode;
    chan->output = output;
    chan->channel = id;
    chan->info_mask_separate = BIT(IIIO_CHAN_INFO_RAW);
    chan->scan_type.sign = 'u';
    chan->scan_type.realbits = 12;
    chan->scan_type.storagebits = 16;
    chan->scan_type.endianness = IIIO_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 = IIIO_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(IIIO_CHAN_INFO_RAW);
    chan->scan_type.sign = 's';
    chan->scan_type.realbits = 12;
    chan->scan_type.storagebits = 16;
    chan->scan_type.endianness = IIIO_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 for the IIO device */
iio_dev = devm_iio_device_alloc(dev, sizeof(*st));
if (!iio_dev)
    return -ENOMEM;

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

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

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

/* 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/>

You will use the 1.4.3 version of the library and tools during this demonstration section:

libgpiod	libgpiod	1.4.3	LGPLv2.1+	C library and tools for interacting with the linux GPIO character device
libgpiod	libgpiod-tools	1.4.3	LGPLv2.1+	C library and tools for interacting with the linux GPIO character device

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_max11300_driver.zip` file from the github of the book and unzip it in the `STM32MP15-Ecosystem-v2.0.0` folder of the Linux host:

```
PC:~$ cd ~/STM32MP15-Ecosystem-v2.0.0/
```

Compile and deploy the drivers to the **STM32MP157C-DK2** Discovery kit:

```
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_max11300_driver$ make
```

```
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_max11300_driver$ make deploy
```

Follow the next instructions to test the driver:

```
/* load the module */
root@stm32mp1:~# insmod max11300-base.ko
[ 49.999595] max11300_base: loading out-of-tree module taints kernel.
root@stm32mp1:~# insmod max11300.ko
[ 53.414477] max11300 spi0.0: max11300_probe() function is called
[ 53.419065] max11300 spi0.0: the value of DCR_ADDRESS is 10
[ 53.443251] max11300 spi0.0: the value of reset is 8000
[ 53.447408] max11300 spi0.0: read SE channel
[ 53.463302] max11300 spi0.0: the value of device ID is 424
[ 53.467382] max11300 spi0.0: the value of DACREF_CONT_SWEEP is 43
[ 53.483879] max11300 spi0.0: the setup of the device is done
[ 53.488095] max11300 spi0.0: the channel 0 is set in port mode 7
[ 53.513303] max11300 spi0.0: the channel 1 is set in port mode 7
[ 53.517860] max11300 spi0.0: the channel 2 is set in port mode 5
[ 53.543299] max11300 spi0.0: the channel 3 is set in port mode 5
[ 53.547856] max11300 spi0.0: the channel 4 is set in port mode 8
[ 53.558583] max11300 spi0.0: the channel 5 is set in port mode 9
[ 53.573303] max11300 spi0.0: there are 1 channels in mode_6
[ 53.577414] max11300 spi0.0: the channel 6 is set in port mode 6
[ 53.603435] max11300 spi0.0: the channel 7 is set in port mode 1
[ 53.607979] max11300 spi0.0: the gpio number 7 is using the gpio offset number
0
```

```

[ 53.633269] max11300 spi0.0: the channel 8 is set in port mode 3
[ 53.637995] max11300 spi0.0: the gpio number 8 is using the gpio offset number
1
[ 53.653305] max11300 spi0.0: the channel 9 is set in default port mode_0
[ 53.658550] max11300 spi0.0: the channel 10 is set in default port mode_0
[ 53.683352] max11300 spi0.0: the channel 11 is set in default port mode_0
[ 53.703354] max11300 spi0.0: the channel 12 is set in default port mode_0
[ 53.708682] max11300 spi0.0: the channel 13 is set in default port mode_0
[ 53.733264] max11300 spi0.0: the channel 14 is set in default port mode_0
[ 53.738596] max11300 spi0.0: the channel 15 is set in default port mode_0
[ 53.753306] max11300 spi0.0: the channel 16 is set in default port mode_0
[ 53.758638] max11300 spi0.0: the channel 17 is set in default port mode_0
[ 53.783352] max11300 spi0.0: the channel 18 is set in port mode 1
[ 53.787984] max11300 spi0.0: the gpio number 18 is using the gpio offset number
2
[ 53.813258] max11300 spi0.0: the channel 19 is set in port mode 3
[ 53.817891] max11300 spi0.0: the gpio number 19 is using the gpio offset number
3
[ 53.843381] max11300 spi0.0: the port 0 is configured as MODE 7
[ 53.847839] max11300 spi0.0: the port 1 is configured as MODE 7
[ 53.873361] max11300 spi0.0: the port 2 is configured as MODE 5
[ 53.877825] max11300 spi0.0: the port 3 is configured as MODE 5
[ 53.893290] max11300 spi0.0: the port 4 is configured as MODE 8
[ 53.897752] max11300 spi0.0: bad port mode for channel 5
[ 53.903040] max11300 spi0.0: the port 6 is configured as MODE 6
[ 53.933290] max11300 spi0.0: the channel 7 is set in port mode_1
[ 53.937836] max11300 spi0.0: the channel 8 is set in port mode_3
[ 53.963201] max11300 spi0.0: the channel is set in default port mode_0
[ 53.968395] max11300 spi0.0: the channel is set in default port mode_0
[ 53.993241] max11300 spi0.0: the channel is set in default port mode_0
[ 53.998314] max11300 spi0.0: the channel is set in default port mode_0
[ 54.013253] max11300 spi0.0: the channel is set in default port mode_0
[ 54.018322] max11300 spi0.0: the channel is set in default port mode_0
[ 54.041409] max11300 spi0.0: the channel is set in default port mode_0
[ 54.063302] max11300 spi0.0: the channel is set in default port mode_0
[ 54.068369] max11300 spi0.0: the channel is set in default port mode_0
[ 54.083404] max11300 spi0.0: the channel 18 is set in port mode_1
[ 54.088038] max11300 spi0.0: the channel 19 is set in port mode_3
[ 54.113297] max11300 spi0.0: the value of adc cfg addr for channel 0 in port
mode 7 is 20
[ 54.120010] max11300 spi0.0: the channel 0 is set in port mode 7
[ 54.143298] max11300 spi0.0: the value of adc cfg val for channel 0 in port
mode 7 is 7100
[ 54.164512] max11300 spi0.0: the value of adc cfg addr for channel 1 in port
mode 7 is 21
[ 54.171232] max11300 spi0.0: the channel 1 is set in port mode 7
[ 54.193247] max11300 spi0.0: the value of adc cfg val for channel 1 in port
mode 7 is 71e0

```



```

[ 54.214426] max11300 spi0.0: the value of adc cfg addr for channel 2 in port
mode 5 is 22
[ 54.221142] max11300 spi0.0: the channel 2 is set in port mode 5
[ 54.243258] max11300 spi0.0: the value of adc cfg val for channel 2 in port
mode 5 is 5100
[ 54.264524] max11300 spi0.0: the value of adc cfg addr for channel 3 in port
mode 5 is 23
[ 54.271238] max11300 spi0.0: the channel 3 is set in port mode 5
[ 54.293253] max11300 spi0.0: the value of adc cfg val for channel 3 in port
mode 5 is 5100
[ 54.314402] max11300 spi0.0: the value of adc cfg addr for channel 4 in port
mode 8 is 24
[ 54.321121] max11300 spi0.0: the channel 4 is set in port mode 8
[ 54.343410] max11300 spi0.0: the value of adc cfg val for channel 4 in port
mode 8 is 8105
[ 54.364616] max11300 spi0.0: the value of adc cfg addr for channel 5 in port
mode 9 is 25
[ 54.371335] max11300 spi0.0: the channel 5 is set in port mode 9
[ 54.393306] max11300 spi0.0: the value of adc cfg val for channel 5 in port
mode 9 is 9100
[ 54.414374] max11300 spi0.0: the value of adc cfg addr for channel 6 in port
mode 6 is 26
[ 54.421092] max11300 spi0.0: the channel 6 is set in port mode 6
[ 54.443469] max11300 spi0.0: the value of adc cfg val for channel 6 in port
mode 6 is 6100
[ 54.464637] max11300 spi0.0: the port 7 is set in port mode_1
[ 54.468921] max11300 spi0.0: the port 8 is set in port mode_3
[ 54.493295] max11300 spi0.0: the port 9 is set in default port mode_0
[ 54.498273] max11300 spi0.0: the port 10 is set in default port mode_0
[ 54.523486] max11300 spi0.0: the port 11 is set in default port mode_0
[ 54.528547] max11300 spi0.0: the port 12 is set in default port mode_0
[ 54.543431] max11300 spi0.0: the port 13 is set in default port mode_0
[ 54.548497] max11300 spi0.0: the port 14 is set in default port mode_0
[ 54.573339] max11300 spi0.0: the port 15 is set in default port mode_0
[ 54.578402] max11300 spi0.0: the port 16 is set in default port mode_0
[ 54.603446] max11300 spi0.0: the port 17 is set in default port mode_0
[ 54.608512] max11300 spi0.0: the port 18 is set in port mode_1
[ 54.633300] max11300 spi0.0: the port 19 is set in port mode_3

```

```

root@stm32mp1:~# cd /sys/bus/iio/devices/iio:device0/

```

```

root@stm32mp1:/sys/devices/platform/soc/44005000.spi/spi_master/spi0/spi0.0/iio:de
vice0#

```

```

/* check the IIO sysfs entries under the IIO MAX11300 device */

```

```

root@stm32mp1:/sys/devices/platform/soc/44005000.spi/spi_master/spi0/spi0.0/iio:de
vice0# ls

```

dev	in_voltage1_mode_7_ADC_raw
in_voltage6_mode_6_DAC_ADC_raw	of_node
out_voltage3_mode_5_DAC_raw	power uevent
in_voltage0_mode_7_ADC_raw	in_voltage4-voltage5_mode_8_ADC_raw name
out_voltage2_mode_5_DAC_raw	out_voltage6_mode_6_DAC_ADC_raw subsystem

### Connect port2 (DAC) to port0 (ADC)

```
/* write to the port2 (DAC) */
root@stm32mp1:/sys/devices/platform/soc/44005000.spi/spi_master/spi0/spi0.0/iio:device0# echo 1000 > out_voltage2_mode_5_DAC_raw
[ 813.600342] max11300 spi0.0: the DAC data register is 62
[ 813.604560] max11300 spi0.0: the value in the DAC data register is 3e8
```

```
/* read the port0 (ADC) */
root@stm32mp1:/sys/devices/platform/soc/44005000.spi/spi_master/spi0/spi0.0/iio:device0# cat in_voltage0_mode_7_ADC_raw
[ 835.930969] max11300 spi0.0: read SE channel
1001
```

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

```
/* set 5V output in the port2 (DAC) */
root@stm32mp1:/sys/devices/platform/soc/44005000.spi/spi_master/spi0/spi0.0/iio:device0# echo 2047 > out_voltage2_mode_5_DAC_raw
[ 282.286001] max11300 spi0.0: the DAC data register is 62
[ 282.289852] max11300 spi0.0: the value in the DAC data register is 7ff
```

```
/* set 2.5V in the port3 (DAC) */
root@stm32mp1:/sys/devices/platform/soc/44005000.spi/spi_master/spi0/spi0.0/iio:device0# echo 1024 > out_voltage3_mode_5_DAC_raw
[ 314.356308] max11300 spi0.0: the DAC data register is 63
[ 314.361039] max11300 spi0.0: the value in the DAC data register is 400
```

```
/* read differential input (port4_port5): 2.5V */
root@stm32mp1:/sys/devices/platform/soc/44005000.spi/spi_master/spi0/spi0.0/iio:device0# cat in_voltage4-voltage5_mode_8_ADC_raw
[ 335.131855] max11300 spi0.0: read differential channel
513
```

```
/* set DAC and read ADC in port mode 6 */
root@stm32mp1:/sys/devices/platform/soc/44005000.spi/spi_master/spi0/spi0.0/iio:device0# echo 1024 > out_voltage6_mode_6_DAC_ADC_raw
[11090.790511] max11300 spi0.0: the DAC data register is 66
[11090.794478] max11300 spi0.0: the value in the DAC data register is 400
root@stm32mp1:/sys/devices/platform/soc/44005000.spi/spi_master/spi0/spi0.0/iio:device0# cat in_voltage6_mode_6_DAC_ADC_raw
```

```
[11095.169444] max11300 spi0.0: read SE channel  
1022
```

```
/* check the gpio chip controllers */
```

```
root@stm32mp1:~# ls -l /dev/gpiochip*  
crw----- 1 root root 254, 0 Feb 7 15:50 /dev/gpiochip0  
crw----- 1 root root 254, 1 Feb 7 15:50 /dev/gpiochip1  
crw----- 1 root root 254, 10 Feb 7 16:07 /dev/gpiochip10  
crw----- 1 root root 254, 2 Feb 7 15:50 /dev/gpiochip2  
crw----- 1 root root 254, 3 Feb 7 15:50 /dev/gpiochip3  
crw----- 1 root root 254, 4 Feb 7 15:50 /dev/gpiochip4  
crw----- 1 root root 254, 5 Feb 7 15:50 /dev/gpiochip5  
crw----- 1 root root 254, 6 Feb 7 15:50 /dev/gpiochip6  
crw----- 1 root root 254, 7 Feb 7 15:50 /dev/gpiochip7  
crw----- 1 root root 254, 8 Feb 7 15:50 /dev/gpiochip8  
crw----- 1 root root 254, 9 Feb 7 15:50 /dev/gpiochip9  
root@stm32mp1:~#
```

```
/* active-high means that 0 value sets output line low */
```

```
/* Print information of all the lines of the gpiochip10 */
```

```
root@stm32mp1:~# gpioinfo gpiochip10  
gpiochip10 - 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 (GP0) to port 18 (GPI)**

```
/* Set port19 (GP0) to high */
```

```
root@stm32mp1:~# gpioset gpiochip10 3=1  
[ 62.435888] max11300 spi0.0: The GPIO is set as an output  
[ 62.450060] max11300 spi0.0: The GPIO ouput is set  
[ 62.453531] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.  
Pin is > 0x0F
```

```
/* Read port 18 (GPI) */
```

```
root@stm32mp1:~# gpioget gpiochip10 2  
[ 84.553859] max11300 spi0.0: The GPIO is set as an input  
[ 84.559241] max11300 spi0.0: The GPIO input is get  
[ 84.562564] max11300 spi0.0: read SE channel  
1
```

```
/* Set port19 (GP0) to low */
```

```
root@stm32mp1:~# gpioset gpiochip10 3=0  
[ 237.579351] max11300 spi0.0: The GPIO is set as an output  
[ 237.586048] max11300 spi0.0: The GPIO ouput is set
```

```
[ 237.589376] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.  
Pin is > 0x0F
```

```
/* Read port 18 (GPI) */
```

```
root@stm32mp1:~# gpioget gpiochip10 2  
[ 242.972241] max11300 spi0.0: The GPIO is set as an input  
[ 242.977719] max11300 spi0.0: The GPIO input is get  
[ 242.981045] max11300 spi0.0: read SE channel  
0
```

**connect port19 (GPO) to port 7 (GPI)**

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

```
root@stm32mp1:~# gpiowrite gpiochip10 3=1  
[ 353.390612] max11300 spi0.0: The GPIO is set as an output  
[ 353.397354] max11300 spi0.0: The GPIO ouput is set  
[ 353.400681] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.  
Pin is > 0x0F
```

```
/* Read port7 (GPI) */
```

```
root@stm32mp1:~# gpioget gpiochip10 0  
[ 360.911737] max11300 spi0.0: The GPIO is set as an input  
[ 360.917224] max11300 spi0.0: The GPIO input is get  
[ 360.920549] max11300 spi0.0: read SE channel  
1
```

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

```
root@stm32mp1:~# gpiowrite gpiochip10 3=0  
[ 395.411163] max11300 spi0.0: The GPIO is set as an output  
[ 395.417793] max11300 spi0.0: The GPIO ouput is set  
[ 395.423392] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.  
Pin is > 0x0F
```

```
/* Read port7 (GPI) */
```

```
root@stm32mp1:~# gpioget gpiochip10 0  
[ 398.715539] max11300 spi0.0: The GPIO is set as an input  
[ 398.720941] max11300 spi0.0: The GPIO input is get  
[ 398.724369] max11300 spi0.0: read SE channel  
0
```

**connect port8 (GPO) to port 7 (GPI)**

```
/* Set port8 (GPO) to high */
```

```
root@stm32mp1:~# gpiowrite gpiochip10 1=1  
[ 513.866874] max11300 spi0.0: The GPIO is set as an output  
[ 513.877063] max11300 spi0.0: The GPIO ouput is set  
[ 513.880397] max11300 spi0.0: The GPIO ouput is set high and port_number is 8.  
Pin is < 0x0F
```

```

/* Read port7 (GPI) */
root@stm32mp1:~# gpioget gpiochip10 0
[ 524.255066] max11300 spi0.0: The GPIO is set as an input
[ 524.260480] max11300 spi0.0: The GPIO input is get
[ 524.264006] max11300 spi0.0: read SE channel
1

/* Set port8 (GPO) to low */
root@stm32mp1:~# gpioset gpiochip10 1=0
[ 549.280354] max11300 spi0.0: The GPIO is set as an output
[ 549.287047] max11300 spi0.0: The GPIO output is set
[ 549.290375] max11300 spi0.0: The GPIO output is set low and port_number is 8.
Pin is < 0x0F

/* Read port7 (GPI) */
root@stm32mp1:~# gpioget gpiochip10 0
[ 553.596437] max11300 spi0.0: The GPIO is set as an input
[ 553.601859] max11300 spi0.0: The GPIO input is get
[ 553.606632] max11300 spi0.0: read SE channel
0

/* check the new direction of the gpio lines */
root@stm32mp1:~# gpioinfo gpiochip10
gpiochip10 - 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@stm32mp1:~# rmmod max11300.ko
root@stm32mp1:~# 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 `sysfs` interface is deprecated, and is highly recommend to use the new interface. These are some of the advantages of using the new character device user API:

- One device file for each `gpiochip`:  
`/dev/gpiochip0`, `/dev/gpiochip1`, `/dev/gpiochipX...`
- Similar to other kernel interfaces: `ioctl()` + `poll()` + `read()` + `close()`
- Possible to set/read multiple GPIOs at once.
- Possible to find GPIO lines by name.

You can find the userspace API for the GPIO character devices in the `linux/gpio.h` file. In the STM32MP1 SDK you can find this file in the following path:

```
~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/SDK/syroots/cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi/usr/include/linux/gpio.h
```

The following application toggles ten times the GPIO PA14 connected to the green LED of the STM32MP1 eval board.

## 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>

#define DEVICE_GPIO "/dev/gpiochip0"

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

    /* request GPIO line PA14 as an output (green LED) */
    req.lineoffsets[0] = 14;
    req.lines = 1;
    req.flags = GPIOHANDLE_REQUEST_OUTPUT;
    strcpy(req.consumer_label, "led_gpio_a_14");

    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_green 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;
}

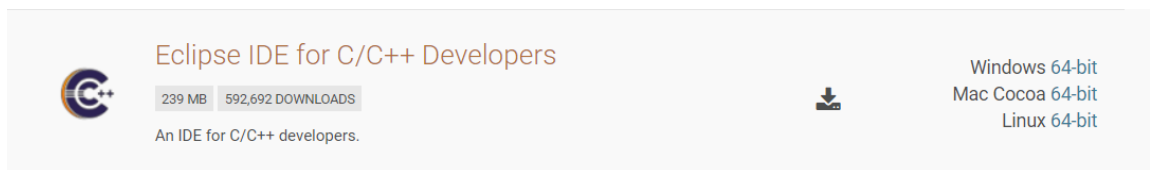
```

## Compiling with Eclipse

Now, you will see how to configure Eclipse for building the previous application. Go to <https://www.eclipse.org/downloads/eclipse-packages/> and select the latest version of Eclipse IDE for C/C++ Developers. At the moment of writing, the current version is the "Eclipse IDE 2020-06" release. Installing Eclipse is simple. Just download a proper version from the web page and untar it. The system must have the proper version of Java SDK installed. Ubuntu allows multiple packages providing a Java Virtual Machine to be installed. For Neon is 8.

**PC:~\$** sudo apt-get install openjdk-8-jdk

Download the "Eclipse IDE for C/C++ Developers" tarball for Linux 64-bit and copy it into the /opt host folder.



```

PC:~$ sudo tar xf eclipse-cpp-2020-06-R-linux-gtk-x86_64.tar.gz -C /opt/
PC:~$ sudo chown -R root:root /opt/eclipse/
PC:~$ cd /opt/eclipse/

```

Launch Eclipse:



```
PC:/opt/eclipse$ ./eclipse &
```

## Configuring Eclipse for cross-development

In this section, you will learn how to configure the Eclipse tool options to successfully cross-compile an application using the toolchain of the Yocto SDK. You will create a single file project in Eclipse by adding the `gpio_device_app.c` source file.

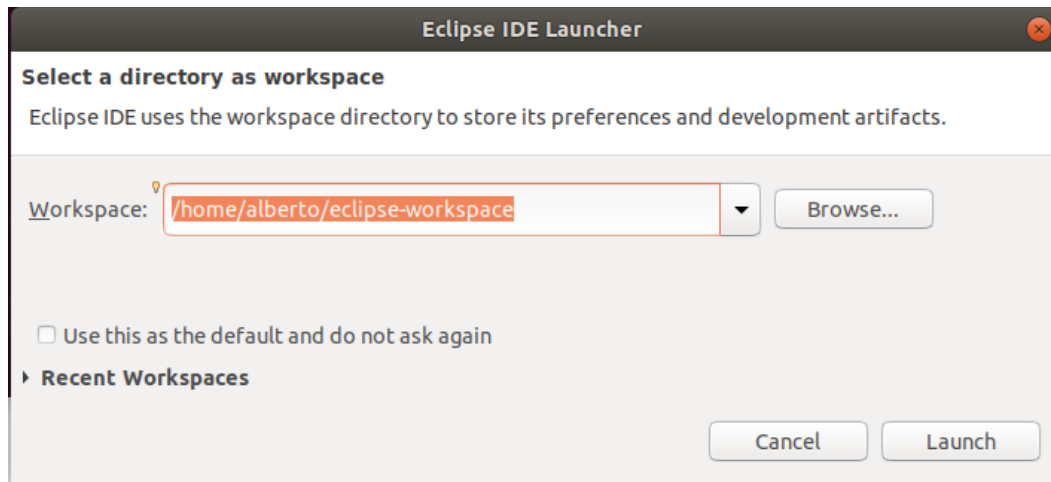
### Create a new C project

You will first create a C project with support for a cross-compiled toolchain, then create a new file and insert into it the code of the `gpio_device_app.c` application. You will explore and change a number of compiler and linker settings in order to get the toolchain to work correctly.

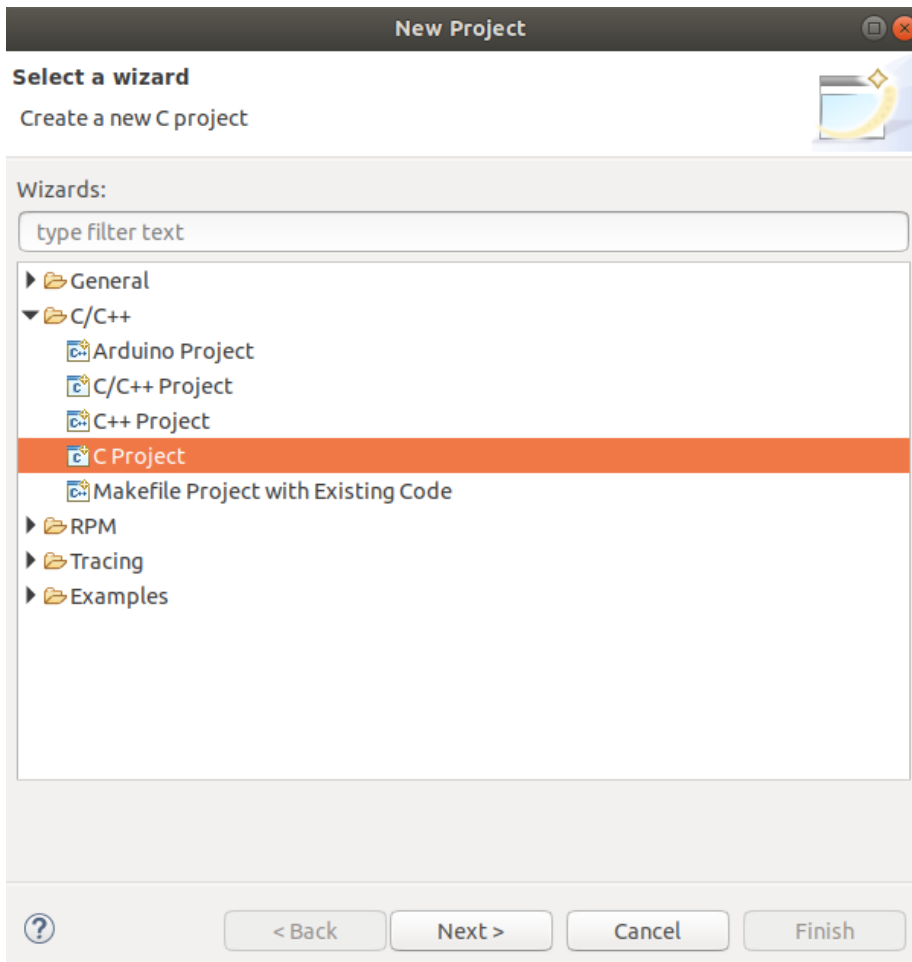
Launch Eclipse:

```
PC:/opt/eclipse$ ./eclipse &
```

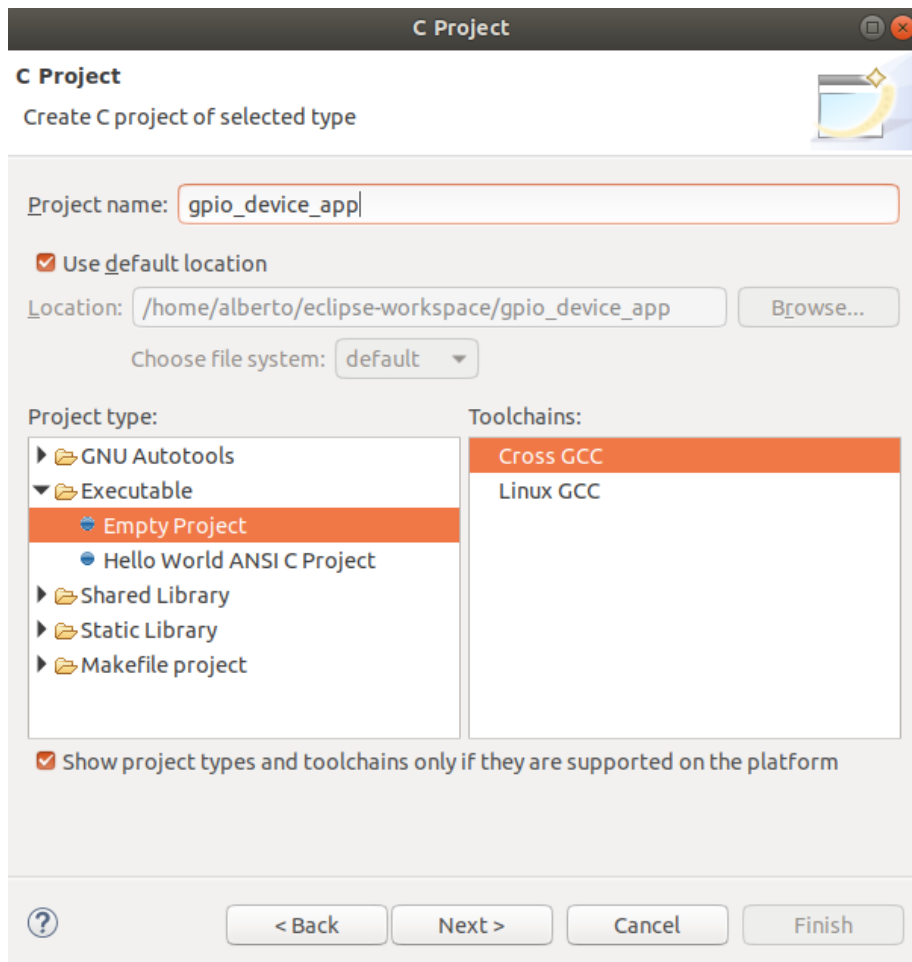
Select a workspace directory:



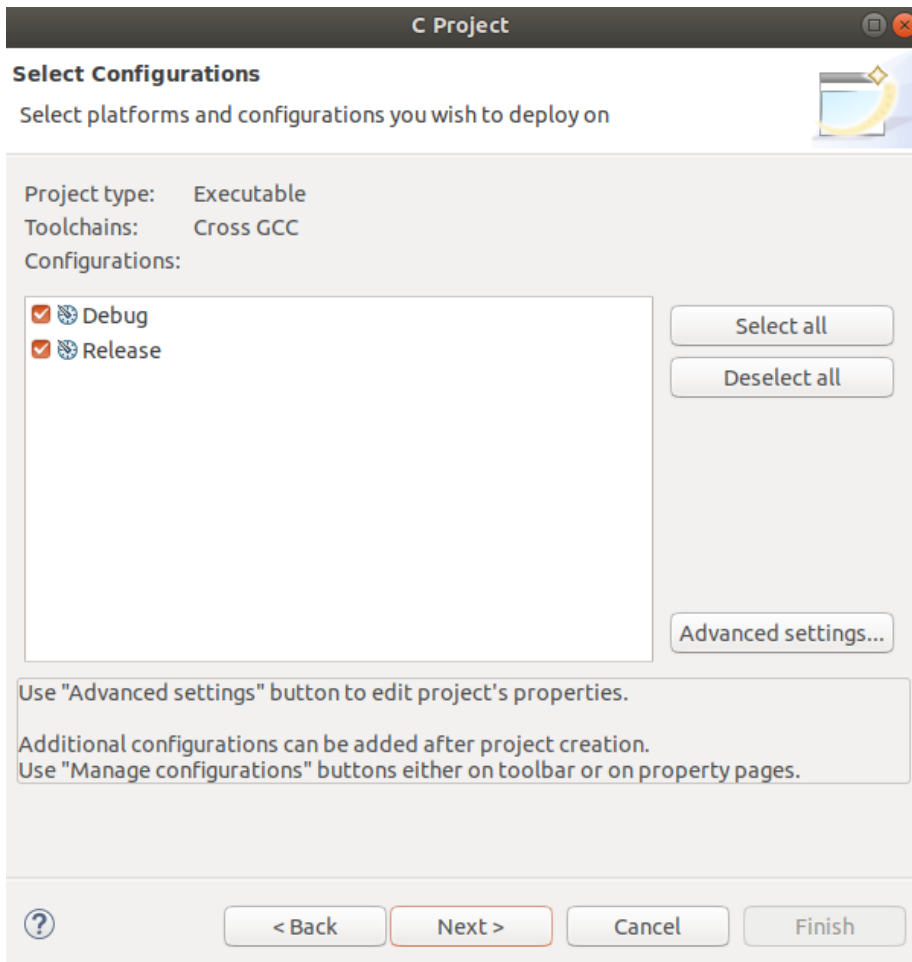
Using the Eclipse menus select **File->New->Project**, then select **C project** and click **Next>**



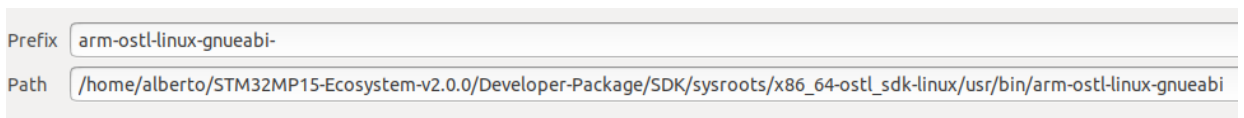
Give the project a name: **gpio\_device\_app**, Select Project type **Executable->Empty Project** and select the **Toolchains: Cross GCC**, click **Next>**.



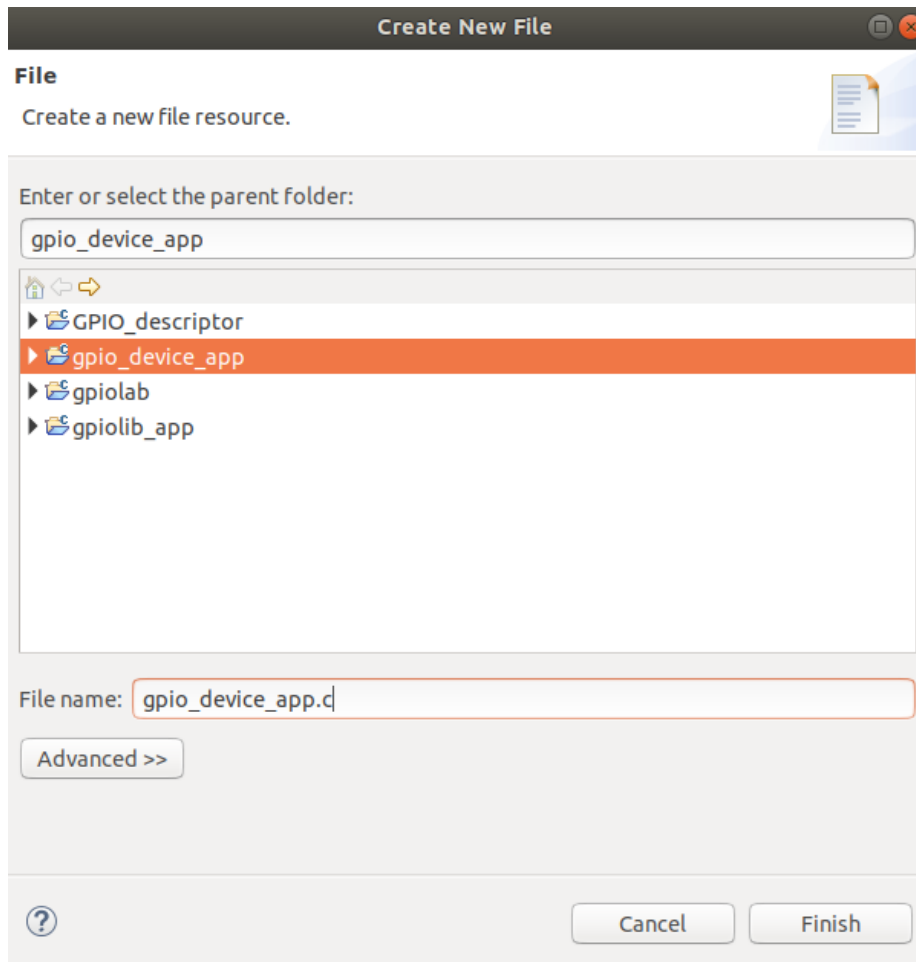
On the **Select Configurations** screen accept the default **Debug** and **Release** build and click **Next>**



On the next screen, enter arm-ostl-linux-gnueabi- as a **Cross compiler prefix** and /home/alberto/STM32MP15-Ecosystem-v2.0.0/Developer-Package/SDK/sysroots/x86\_64-ostl\_sdk-linux/usr/bin/arm-ostl-linux-gnueabi as a **Cross compiler path**, then click **Finish**.

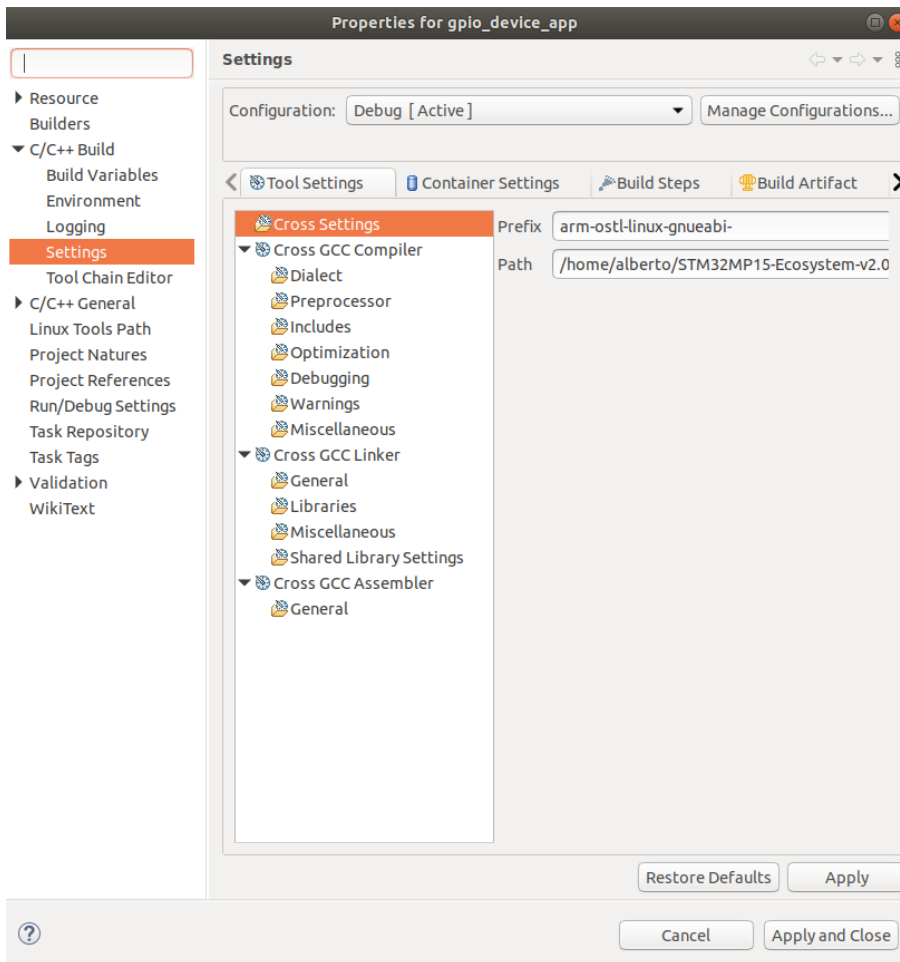


Eclipse will prompt to open the default C perspective. In the newly opened project you will now create the project file. Right click on the gpio\_device\_app project and select **New->File**. Insert **File name:** gpio\_device\_app.c, then click **Finish**.



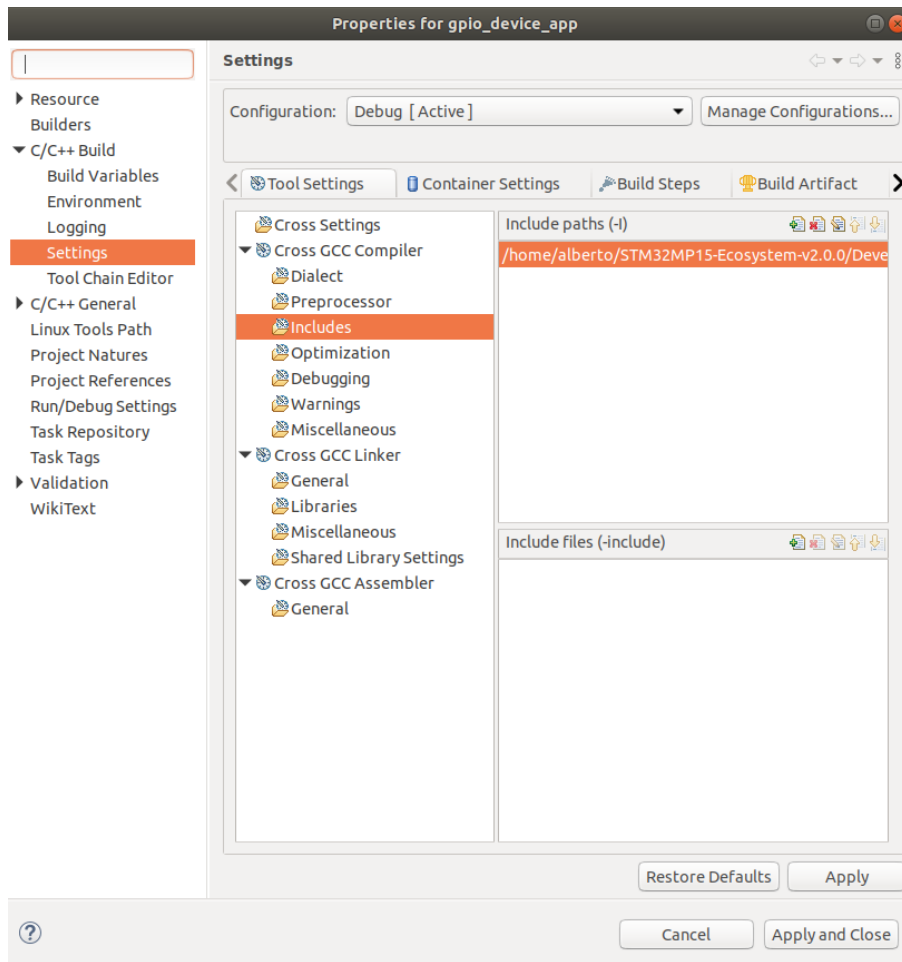
Now, insert the code of the Listing 11-10 into the `gpio_device_app.c` file.

Right click on the `gpio_device_app` project and select **Properties**. Expand the **C/C++ Build** item and click on **Settings**. This is where we will configure the toolchain settings. Under the **Cross settings** item at the top you can see the tool **Prefix** and **Path** that you entered earlier:



Select the **Cross GCC Compiler->Include** option and add this include path:

```
/home/alberto/STM32MP15-Ecosystem-v2.0.0/Developer-  
Package/SDK/sysroots/cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi/usr/include
```



Now, you will enter the compiler flags. You can get these flags sourcing the SDK environment setup script and checking the environment variables CC and CFLAGS:

```
PC:~$ source $SDK_ROOT/SDK/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

```
PC:~$ echo $CC
```

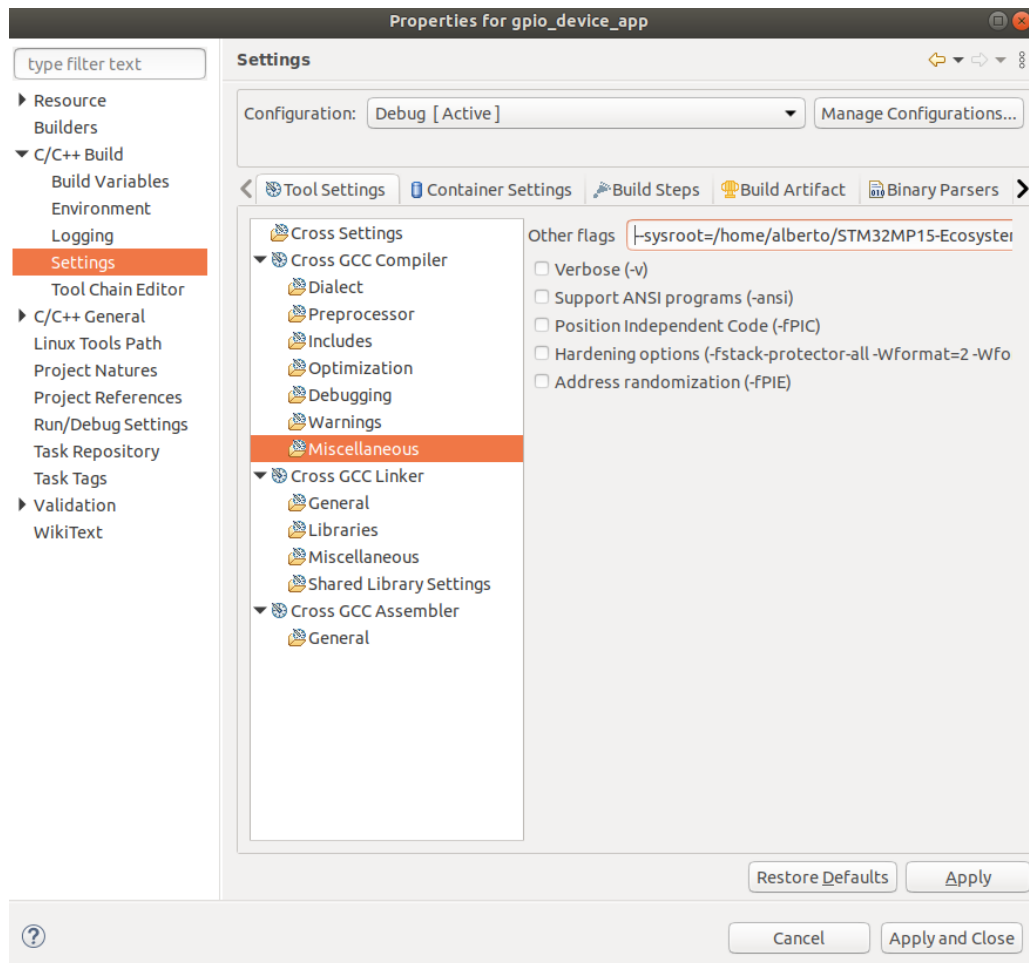
```
arm-ostl-linux-gnueabi-gcc -mthumb -mfpv=neon-vfpv4 -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/home/alberto/STM32MP15-Ecosystem-v2.0.0/Developer-Package/SDK/sysroots/cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

```
PC:~$ echo $CFLAGS
```

```
-O2 -pipe -g -feliminate-unused-debug-types
```

Under the **Cross GCC Compiler -> Miscellaneous** change the **Other flags** to be:

```
--sysroot=/home/alberto/STM32MP15-Ecosystem-v2.0.0/Developer-  
Package/SDK/sysroots/cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi -mthumb -  
mfpu=neon-vfpv4 -mfloat-abi=hard -mcpu=cortex-a7 -O2 -pipe -g -feliminate-unused-  
debug-types -c -fmessage-length=0
```



Now, you will enter the linker flags. You can get these flags sourcing the SDK environment setup script and checking the environment variable **LD\_FLAGS**:



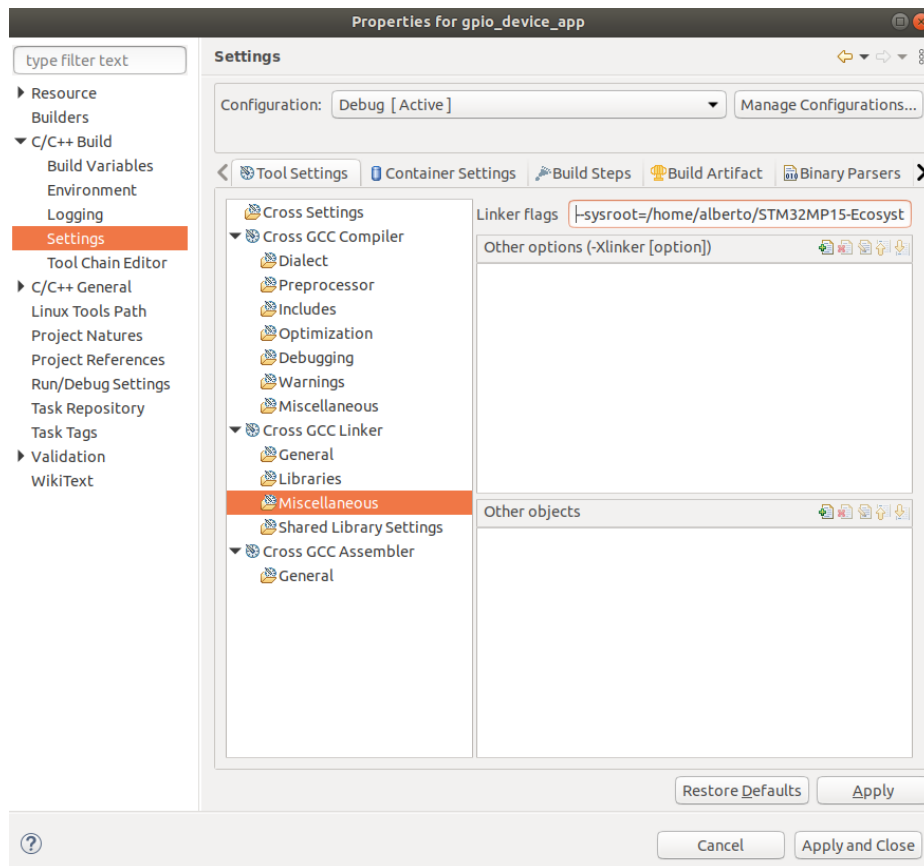
```
PC:~$ source $SDK_ROOT/SDK/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
```

```
PC:~$ echo $LDFLAGS
```

```
-Wl,-O1 -Wl,--hash-style=gnu -Wl,--as-needed
```

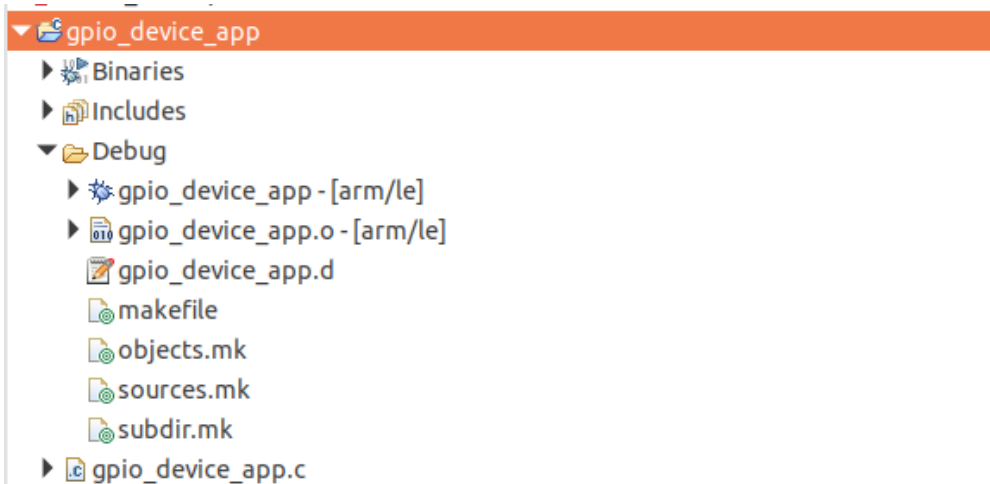
Under the **Cross GCC Linker->Miscellaneous** set the **Linker flags** to be (add linker + compiler flags):

```
--sysroot=/home/alberto/STM32MP15-Ecosystem-v2.0.0/Developer-Package/SDK/sysroots/cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi/ -Wl,-O1 -Wl,--hash-style=gnu -Wl,--as-needed -mthumb -mfpu=neon-vfpv4 -mfloat-abi=hard -mcpu=cortex-a7
```

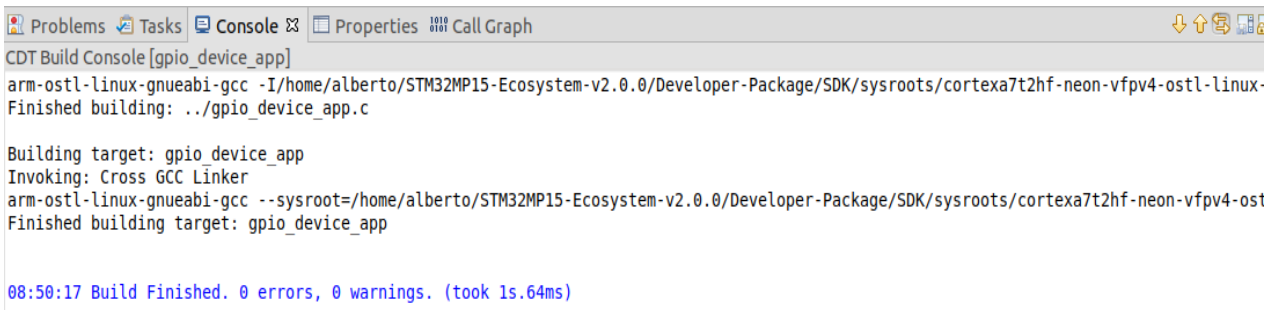


Now, click **Apply and Close**.

Finally, you can build your application using the hammer icon at the top of the Eclipse Window. If everything has been set up correctly your application will build.



You can see the building output in the **Console** Tab.



Deploy the application to the target STM32MP1:

```
PC:/eclipse-workspace/gpio_device_app/Debug$ scp gpio_device_app  
root@10.0.0.10:/home/root
```

Finally, execute the application on the target. You can see the green LED flashing!

```
root@stm32mp1:~# ./gpio_device_app
```

# GPIO control through gpiolibd library

In this section, you will see how to control GPIOs using the functions of the libgpiod library. You can find the userspace API for the libgpiod library in the linux/gpiod.h file. In the STM32MP1 SDK you can find this file in the following path:

```
~/STM32MP15-Ecosystem-v2.0.0/Developer-Package/SDK/syroots/cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi/usr/include/linux/gpiod.h
```

The following libgpiod\_app application has the same behaviour than the gpio\_device\_app one, toggling ten times the GPIO PA14 connected to the green LED of the STM32MP1 eval board, but this time you will use the libgpiod library instead of the “gpio char device” method to control the green LED.

## Listing 11-11: libgpiod\_app.c

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

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

    /* open /dev/gpiochip0 */
    output_chip = gpiod_chip_open_by_number(0);
    if (!output_chip)
        return -1;

    /* get PA14 pin (green LED) */
    output_line = gpiod_chip_get_line(output_chip, 14);
    if(!output_line) {
        gpiod_chip_close(output_chip);
        return -1;
    }

    /* config PA14 as output and set a description */
    if (gpiod_line_request_output(output_line, "green Led",
                                   GPIOD_LINE_ACTIVE_STATE_HIGH) == -1) {
        gpiod_line_release(output_line);
        gpiod_chip_close(output_chip);
        return -1;
    }
}
```

```

    }

    /* toggle 10 times the LED */
    for (int i=0; i < flash; i++) {
        line_value = !line_value;
        ret = gpio_line_set_value(output_line, line_value);
        if (ret == -1) {
            ret = -errno;
            gpio_line_release(output_line);
            gpio_chip_close(output_chip);
            return ret;
        }
        sleep(1);
    }

    gpio_line_release(output_line);
    gpio_chip_close(output_chip);

    return 0;
}

```

You will also use Eclipse to configure a new project. You will keep the same compiler and linker options than the previous example, but an additional configuration to add the libgpiod library. Of course you have to create a new source file with the new example code.

Before you begin creating the new configuration, you will explore a very useful tool to find library dependencies. The name of this tool is pkg-config. Instead of locating the individual include directories and library dependencies and adding them to the build configuration in Eclipse you can use the pkg-config tool to automate this process.

Before opening Eclipse, you have to source the Yocto SDK environment script to set environment variables for pkg-config:

```

PC:~$ source $SDK_ROOT/SDK/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-
gnueabi

PC:~$ which pkg-config
/home/alberto/STM32MP15-Ecosystem-v2.0.0/Developer-Package/SDK/sysroots/x86_64-
ostl_sdk-linux/usr/bin/pkg-config

```

Show the list of include directory locations and the libraries prefixed by -l:

```

PC:~$ pkg-config libgpiod --cflags --libs
-lgpiod

```

Only the library gpiod is shown in the list.

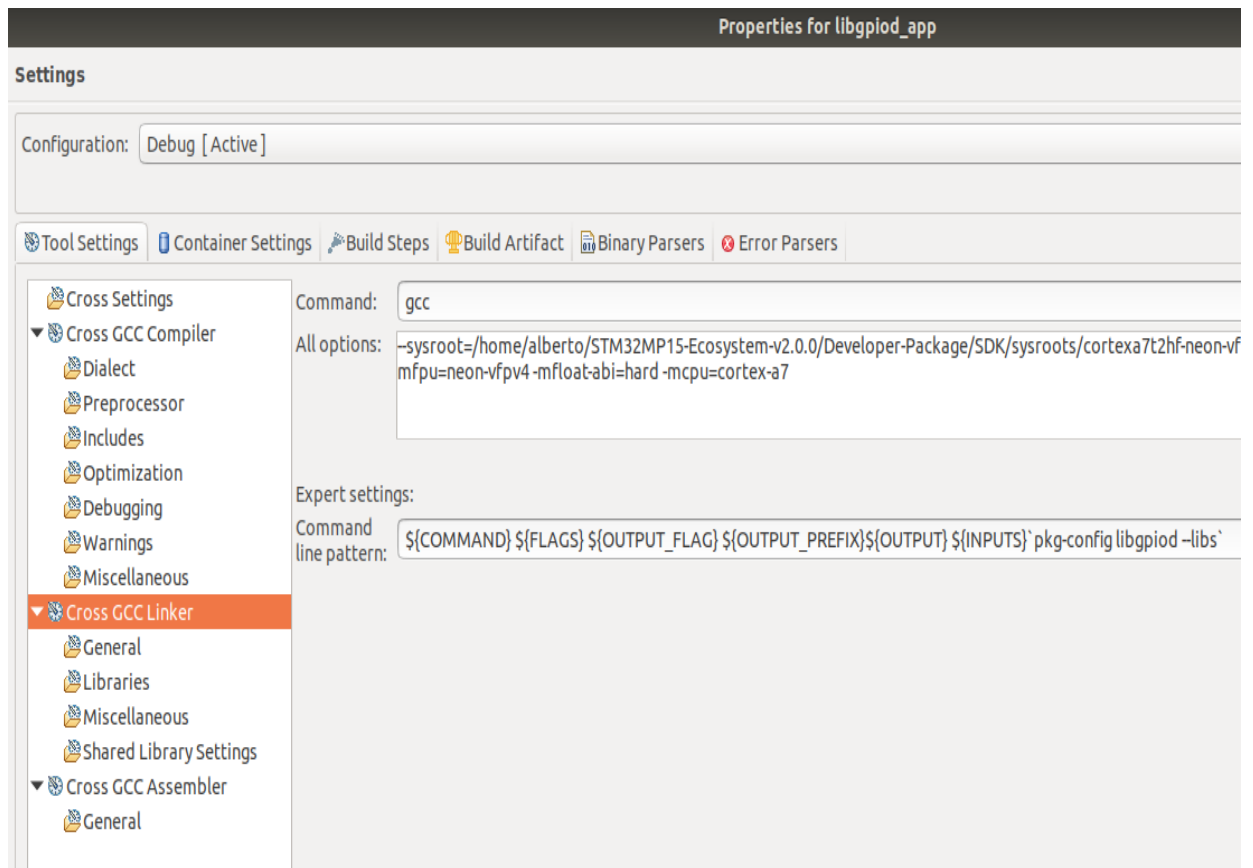
Launch Eclipse in the terminal where you have sourced the environment script and create a new project keeping the same compiler and linker configuration as the previous example:

**PC:/opt/eclipse\$ ./eclipse &**

Create a new source file and insert the code of Listing 11-11: libgpod\_app.c

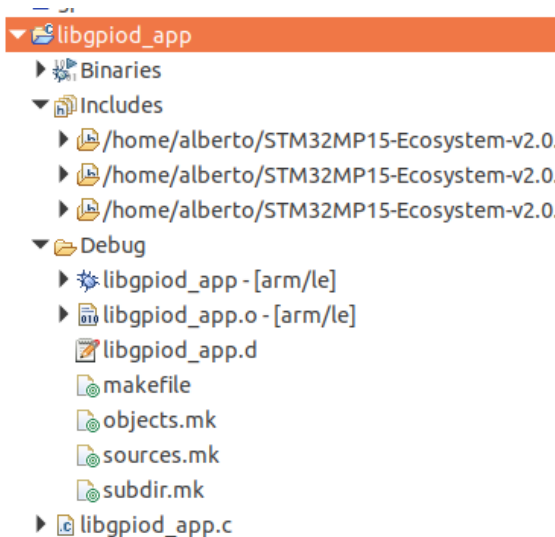
Now, you have to include the gpod library in the Eclipse configuration. Under **Cross GCC Linker->Expert settings** add `pkg-config libgpod --libs` to the **Command line pattern**. This will pass the location of the required libgpod dependencies to the linker. Note the use of the backtick quotes.

```
${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`pkg-config libgpod --libs`
```



Now, click **Apply and Close**.

Finally, build your application using the hammer icon at the top of the Eclipse Window.



Deploy the application to the target STM32MP1:

```
PC:/eclipse-workspace/gpio_device_app/Debug$ scp libgpiod_app  
root@10.0.0.10:/home/root
```

Finally, execute the application on the target. You can see the green LED flashing!

```
root@stm32mp1:~# ./libgpiod_app
```

Now, you can do a slight modification to the previous libgpiod\_app.c application code to control the **port19 (GPO)** of the **MAX11300** device. Create a new project in Eclipse and set the same configuration of the previous libgpiod\_app project. In the code below, you can see in bold the differences respect to the previous libgpiod\_app.c source file.

## Listing 11-11: libgpiod\_max11300\_app.c

```
#include <errno.h>  
#include <stdio.h>  
#include <unistd.h>  
#include <gpiod.h>  
  
int main(int argc, char *argv[])  
{  
    struct gpiod_chip *output_chip;  
    struct gpiod_line *output_line;  
    int line_value = 1;  
    int flash = 10;
```

```

int ret;

/* open /dev/gpiochip10 */
output_chip = gpiod_chip_open_by_number(10);
if (!output_chip)
    return -1;

/* get line 3 (port19) of the gpiochip10 device */
output_line = gpiod_chip_get_line(output_chip, 3);
if(!output_line) {
    gpiod_chip_close(output_chip);
    return -1;
}

/* config port19 (GP0) as output and set output to high level */
if (gpiod_line_request_output(output_line, "Port19_GP0",
                             GPIO_LINE_ACTIVE_STATE_HIGH) == -1) {
    gpiod_line_release(output_line);
    gpiod_chip_close(output_chip);
    return -1;
}

/* toggle 10 times the port19 (GP0) 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;
}

```

Deploy the application to the target STM32MP1:

```

PC:/eclipse-workspace/gpio_max11300_device_app/Debug$ scp libgpiod_max11300_app
root@10.0.0.10:/home/root

```

Load the “max11300” Linux driver:

```

root@stm32mp1:~# insmod max11300-base.ko

```

```
root@stm32mp1:~# insmod max11300.ko
```

Launch on the STM32MP1 the libgpiod\_max11300\_app. You can see the port19 of the MAX11300 device toggling ten times. You can connect a LED in the port19 to reproduce the same behaviour of the previous applications.

```
root@stm32mp1:~# ./libgpiod_max11300_app
[ 292.190891] max11300 spi0.0: The GPIO is set as an output
[ 292.197664] max11300 spi0.0: The GPIO output is set
[ 292.200991] max11300 spi0.0: The GPIO output is set high and port_number is 1F
[ 292.217663] max11300 spi0.0: The GPIO output is set
[ 292.220996] max11300 spi0.0: The GPIO output is set low and port_number is 19F
[ 293.234311] max11300 spi0.0: The GPIO output is set
[ 293.237645] max11300 spi0.0: The GPIO output is set high and port_number is 1F
[ 294.250366] max11300 spi0.0: The GPIO output is set
[ 294.253793] max11300 spi0.0: The GPIO output is set low and port_number is 19F
[ 295.262513] max11300 spi0.0: The GPIO output is set
[ 295.265972] max11300 spi0.0: The GPIO output is set high and port_number is 1F
[ 296.274859] max11300 spi0.0: The GPIO output is set
[ 296.278195] max11300 spi0.0: The GPIO output is set low and port_number is 19F
[ 297.291950] max11300 spi0.0: The GPIO output is set
[ 297.295331] max11300 spi0.0: The GPIO output is set high and port_number is 1F
[ 298.304457] max11300 spi0.0: The GPIO output is set
[ 298.307792] max11300 spi0.0: The GPIO output is set low and port_number is 19F
[ 299.317951] max11300 spi0.0: The GPIO output is set
[ 299.321284] max11300 spi0.0: The GPIO output is set high and port_number is 1F
[ 300.334107] max11300 spi0.0: The GPIO output is set
[ 300.337471] max11300 spi0.0: The GPIO output is set low and port_number is 19F
[ 301.346559] max11300 spi0.0: The GPIO output is set
[ 301.349895] max11300 spi0.0: The GPIO output is set high and port_number is 1F
root@stm32mp1:~#
```

Connect the **port19** (line 3 of the gpiochip10 device) to the **port18** (line 2 of the gpiochip10 device) of the MAX11300 device. The port19 is set to high level after executing libgpiod\_max11300\_app application. You can check it by reading the port18:

```
root@stm32mp1:~# gpioget gpiochip10 2
[ 602.872172] max11300 spi0.0: The GPIO is set as an input
[ 602.877739] max11300 spi0.0: The GPIO input is get
[ 602.881066] max11300 spi0.0: read SE channel
1
```

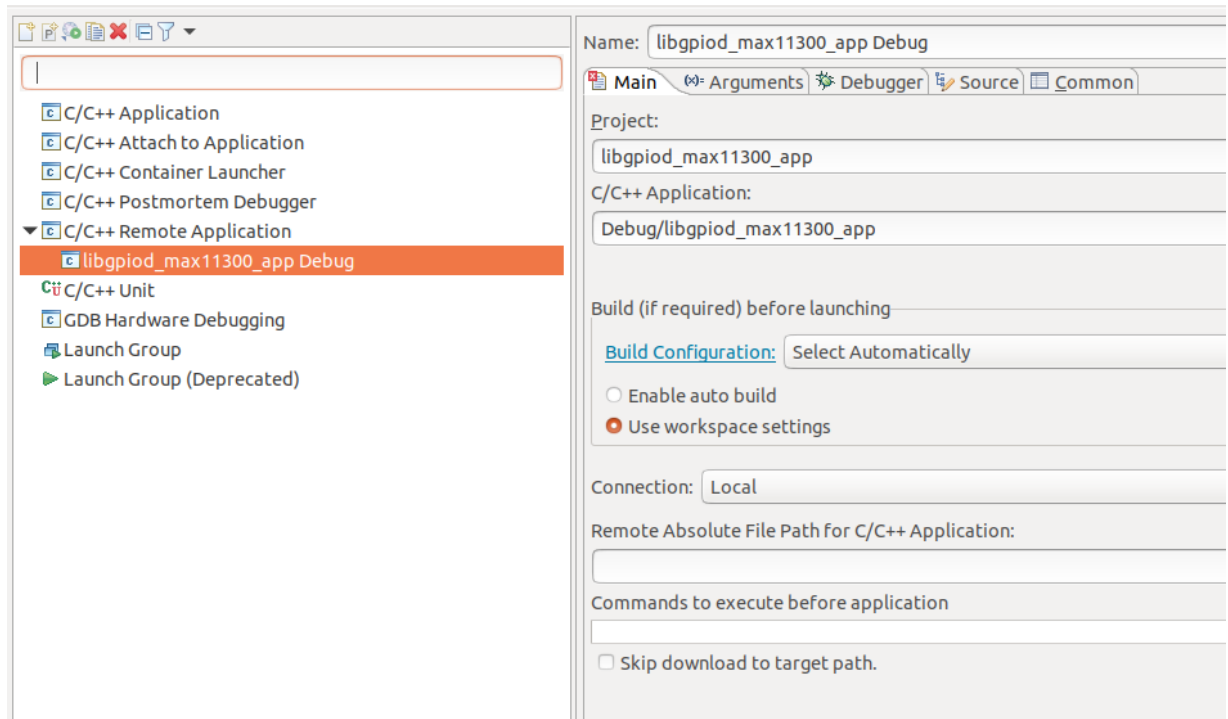


## Debugging with Eclipse

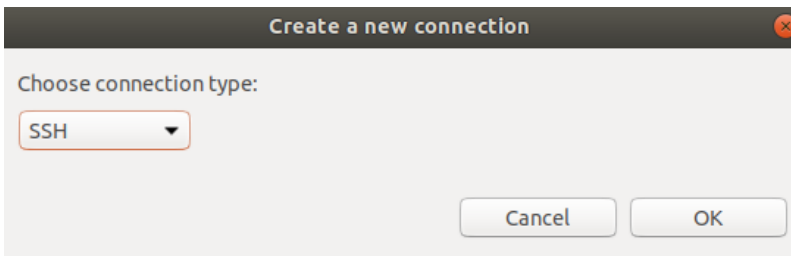
Now, you will see how to create a debug configuration that will connect to the STM32MP1 board to remotely debug your application. This debug configuration will automatically download the compiled binary, start a gdb-server session running and connect to it from within the Eclipse IDE.

Switch back to the C++ perspective in Eclipse. Right click on the previous libgpod\_max11300\_app project and select **Debug As->Debug Configurations...**

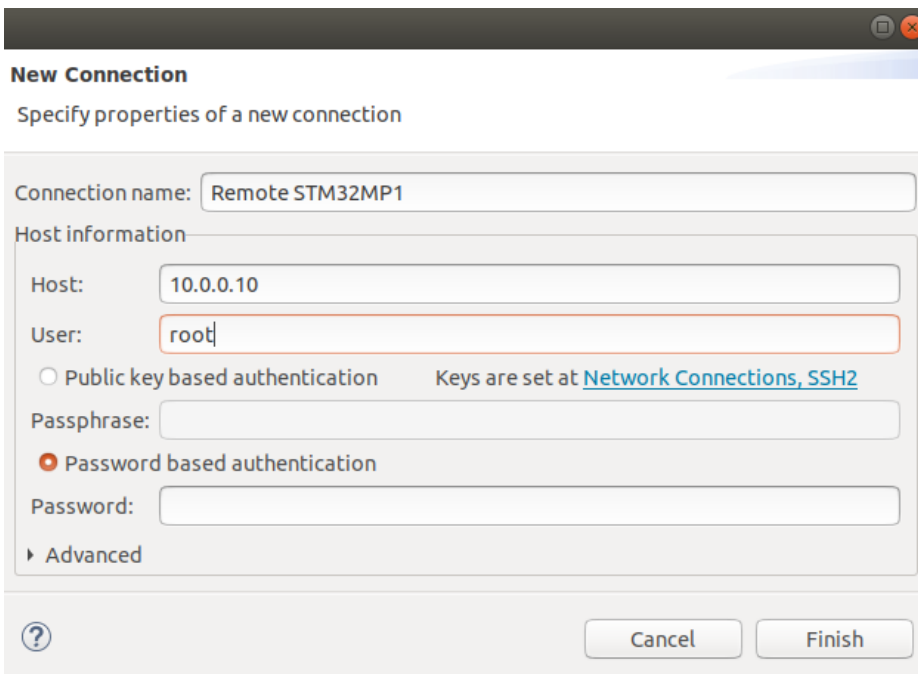
In the Debug Configurations window select **C/C++ Remote Application** and press the **New** Button. The Create configuration window will appear and you can now configure the session.



Create a new **Connection** by clicking **New** and choosing **SSH connection type**; click **OK**:

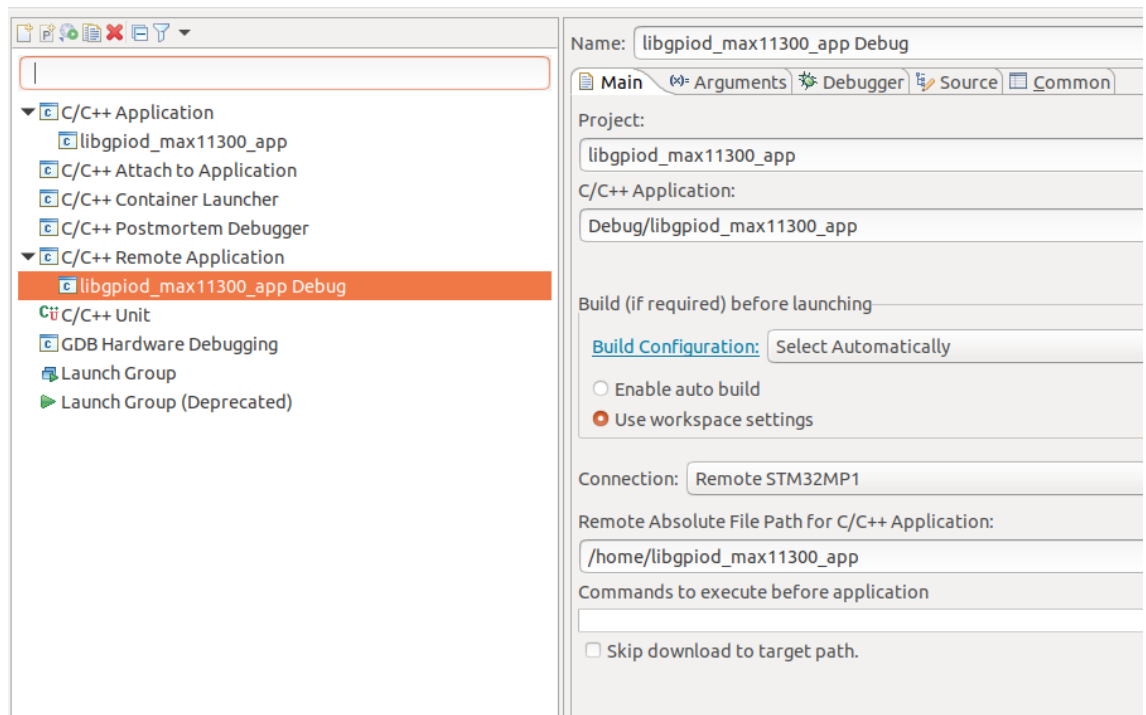


Edit properties of the new connection: specify the STM32MP15 **Host IP address**, the **User** (root) and the **Password based authentication** (leave the **Password** empty); click **Finish**:



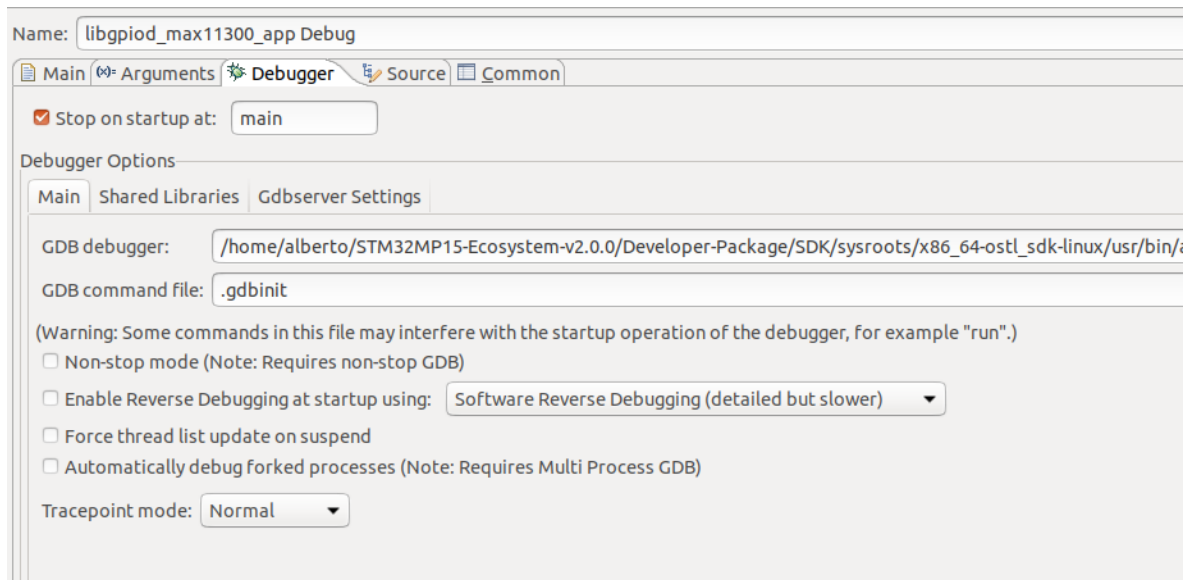
In the Create configuration window change the **Remote Absolute File Path** (this is where the binary will be placed in the STM32MP1 root file system) to be:

`/root/libgpiod_max11300_app`



Now, you need to configure the gdb connection over ssh. Switch from the **Main** tab to the **Debugger** tab. In the GDB debugger enter the Yocto SDK ARM specific gdb:

```
/home/alberto/STM32MP15-Ecosystem-v2.0.0/Developer-Package/SDK/sysroots/x86_64-ostl_sdk-linux/usr/bin/arm-ostl-linux-gnueabi/arm-ostl-linux-gnueabi-gdb
```



Click **Debug**. You will be asked to switch to the **Debug Perspective** window, accept it. Eclipse will automatically deploy your application on the STM32MP1 target and run gdbserver on it using the SSH protocol. The program will halt at the `main()` function.

The image shows a development environment for a C/C++ Remote Application. The Project Explorer on the left lists the application and its threads. The main editor displays the source code for `libgpiod_max11300_app.c`, with line 36 highlighted. The Console at the bottom shows the `gdbserver` process running and listening on port 2345.

```
13
14
15  /* open /dev/gpiochip10 */
16  output_chip = gpiod_chip_open_by_number(10);
17  if (!output_chip)
18      return -1;
19
20  /* get line 3 (port19) of the gpiochip10 device */
21  output_line = gpiod_chip_get_line(output_chip, 3);
22  if (!output_line) {
23      gpiod_chip_close(output_chip);
24      return -1;
25  }
26
27  /* config port19 (GP0) as output and set output to high level */
28  if (gpiod_line_request_output(output_line, "green Led",
29      GPIOD_LINE_ACTIVE_STATE_HIGH) == -1) {
30      gpiod_line_release(output_line);
31      gpiod_chip_close(output_chip);
32      return -1;
33  }
34
35  /* toggle 10 times the port19 (GP0) of the max11300 device */
36  for (int i=0; i < flash; i++) {
37      line_value = !line_value;
38      ret = gpiod_line_set_value(output_line, line_value);
39      if (ret == -1) {
40          ret = -errno;
41          gpiod_line_release(output_line);
42          gpiod_chip_close(output_chip);
43          return ret;
44      }
45      sleep(1);
46  }
47
48  gpiod_line_release(output_line);
49  gpiod_chip_close(output_chip);
50
51  return 0;
52 }
```

libgpiod\_max11300\_app Debug [C/C++ Remote Application]  
libgpiod\_max11300\_app [6237] [cores: 1]  
Thread #1 [libgpiod\_max113] 6237 [core: 1] (Suspended: B)  
main() at libgpiod\_max11300\_app.c:15 0x10578  
Remote Shell  
/home/alberto/STM32MP15-Ecosystem-v2.0.0/Developer-Pa

libgpiod\_max11300\_app Debug [C/C++ Remote Application]  
gdbserver :2345 /home/libgpiod\_max11300\_app;exit  
root@stm32mp1:~# gdbserver :2345 /home/libgpiod\_max11300\_app;exit  
Process /home/libgpiod\_max11300\_app created; pid = 6237  
Listening on port 2345  
Remote debugging from host ::ffff:10.0.0.1, port 40648

Set a breakpoint at line 36. Press Resume (F8), the program will stop at the breakpoint.

```
14  /* open /dev/gpiochip10 */
15  output_chip = gpiod_chip_open_by_number(10);
16  if (!output_chip)
17      return -1;
18
19  /* get line 3 (port19) of the gpiochip10 device */
20  output_line = gpiod_chip_get_line(output_chip, 3);
21  if (!output_line) {
22      gpiod_chip_close(output_chip);
23      return -1;
24  }
25
26  /* config port19 (GP0) as output and set output to high level */
27  if (gpiod_line_request_output(output_line, "Port19_GP0",
28      GPIOD_LINE_ACTIVE_STATE_HIGH) == -1) {
29      gpiod_line_release(output_line);
30      gpiod_chip_close(output_chip);
31      return -1;
32  }
33
34  /* toggle 10 times the port19 (GP0) of the max11300 device */
35  for (int i=0; i < flash; i++) {
36      line_value = !line_value;
37      ret = gpiod_line_set_value(output_line, line_value);
38      if (ret == -1) {
39          ret = -errno;
40          gpiod_line_release(output_line);
41          gpiod_chip_close(output_chip);
42          return ret;
43      }
44      sleep(1);
45  }
46
47  gpiod_line_release(output_line);
48  gpiod_chip_close(output_chip);
49
50  return 0;
51 }
52
```

See below the minicom terminal output:

```
root@stm32mp1:~# [ 6090.669591] max11300 spi0.0: The GPIO is set as an output
[ 6090.676304] max11300 spi0.0: The GPIO output is set
[ 6090.679631] max11300 spi0.0: The GPIO output is set high and port_number is 19.
Pin is > 0x0F
```

Read the port18 of the MAX11300 device. The returned value is 1. See below the minicom output:

```
root@stm32mp1:~# gpioret gpiochip10 2
[ 6193.253441] max11300 spi0.0: The GPIO is set as an input
[ 6193.258859] max11300 spi0.0: The GPIO input is get
[ 6193.262184] max11300 spi0.0: read SE channel
1
```

Press Resume (F8), the program will stop again at the breakpoint. See below the minicom output:

```
root@stm32mp1:~# [ 6291.585690] max11300 spi0.0: The GPIO ouput is set
[ 6291.589025] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
```

Read the port18 of the MAX11300 device. The returned value is 0. See below the minicom output:

```
root@stm32mp1:~# gpioget gpiochip10 2
[ 6328.471317] max11300 spi0.0: The GPIO is set as an input
[ 6328.476896] max11300 spi0.0: The GPIO input is get
[ 6328.480220] max11300 spi0.0: read SE channel
0
```

Remove the breakpoint and Press Resume (F8). See below the minicom output:

```
root@stm32mp1:~# [ 6370.410255] max11300 spi0.0: The GPIO ouput is set
[ 6370.413680] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[ 6371.422515] max11300 spi0.0: The GPIO ouput is set
[ 6371.425977] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[ 6372.434826] max11300 spi0.0: The GPIO ouput is set
[ 6372.438159] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[ 6373.452756] max11300 spi0.0: The GPIO ouput is set
[ 6373.456144] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[ 6374.465011] max11300 spi0.0: The GPIO ouput is set
[ 6374.468374] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[ 6375.482913] max11300 spi0.0: The GPIO ouput is set
[ 6375.486385] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[ 6376.500640] max11300 spi0.0: The GPIO ouput is set
[ 6376.504035] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
[ 6377.512866] max11300 spi0.0: The GPIO ouput is set
[ 6377.516313] max11300 spi0.0: The GPIO ouput is set low and port_number is 19.
Pin is > 0x0F
[ 6378.525251] max11300 spi0.0: The GPIO ouput is set
[ 6378.528586] max11300 spi0.0: The GPIO ouput is set high and port_number is 19.
Pin is > 0x0F
```

Terminate the Debug session.

**Note:** The source code of the applications developed during this lab is included in the linux\_5.4\_max11300\_driver.zip file and can be downloaded from the GitHub repository at [https://github.com/ALIBERA/linux\\_book\\_2nd\\_edition](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>

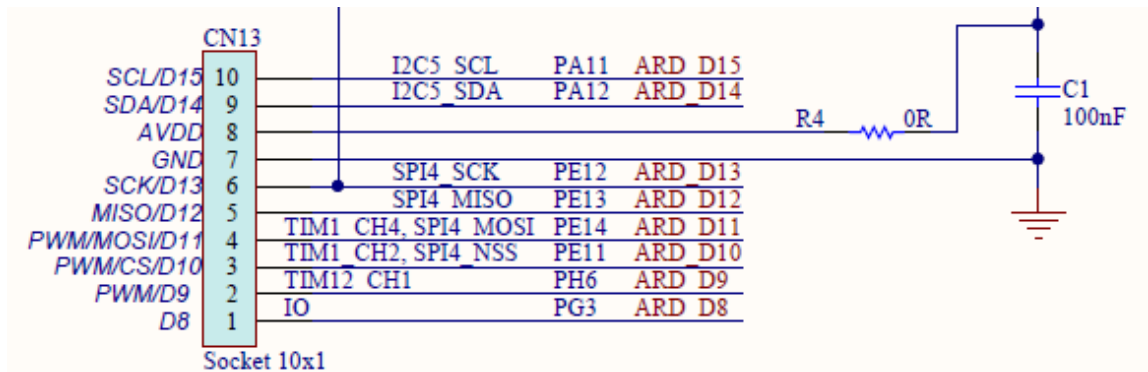
The hardware platforms used in this lab are the STM32MP157C-DK2 board from ST and the EXPAND 6 Click from MIKROE. The documentation of these boards can be found at <https://www.st.com/en/evaluation-tools/stm32mp157c-dk2.html> and <https://www.mikroe.com/expand-6-click>

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 I2C5 pins of the STM32MP157C-DK2 CN13 connector to connect to the EXPAND 6 Click mikroBUS™ socket. See below the STM32MP157C-DK2 CN13 connector:





And the EXPAND 6 Click mikroBUS™ socket:

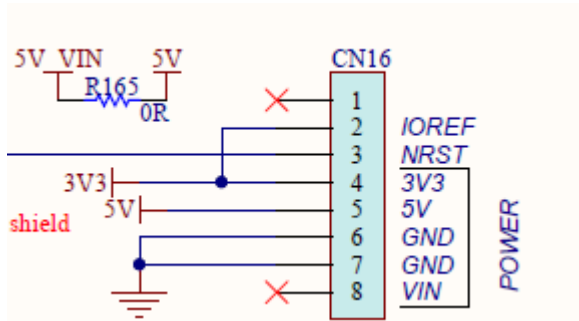
Notes	Pin	mikro™ BUS				Pin	Notes
	NC	1	AN	PWM	16	NC	
Reset	<b>RST</b>	2	RST	INT	15	<b>INT</b>	Interrupt
	NC	3	CS	RX	14	NC	
	NC	4	SCK	TX	13	NC	
	NC	5	MISO	SCL	12	<b>SCL</b>	I2C Clock
	NC	6	MOSI	SDA	11	<b>SDA</b>	I2C Data
Power Supply	<b>3.3V</b>	7	3.3V	5V	10	<b>5V</b>	Power Supply
Ground	<b>GND</b>	8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

Open the STM32MP157C-DK2 schematic to see the CN13 connector and look for the I2C5 pins. Connect the CN13 I2C5 pins to the CY8C9520A I2C ones obtained from the EXPAND 6 Click mikroBUS™ socket:

- Connect the STM32MP157C-DK2 **I2C5\_SCL** (Pin 10 of CN13) to CY8C9520A **SCL** (Pin 12 of Mikrobus)
- Connect the STM32MP157C-DK2 **I2C5\_SDA** (Pin 9 of CN13) to CY8C9520A **SDA** (Pin 11 of Mikrobus)

- Connect the STM32MP157C-DK2 **PG3** pad (Pin 1 of CN13) to CY8C9520A **INT** (Pin 15 of Mikrobus)

Now, find the CN16 connector in the STM32MP157C-DK2 schematic:



And connect the next power pins between the two boards:

- Connect the Pin 4 of CN16 (3.3V) to CY8C9520A 3.3V (Pin 7 of Mikrobus)
- Connect the Pin 6 of CN16 (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 `stm32mp15xx-dkx.dtsi` DT file and find the `i2c5` controller master node. Inside the `i2c5` node, you can see the `pinctrl` properties which configure the pins in I2C mode when the system runs and into a different state (ANALOG) when the system suspends to RAM. Both `i2c5_pins_a` and `i2c5_pins_sleep_a` are already defined in the `stm32mp15-pinctrl.dtsi` file.

The `i2c5` 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 `i2c5` 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 PG3 pad of the STM32MP1 processor, so the interrupt parent of our device is the GPIOG peripheral of the STM32MP1 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 GPIOG peripheral is needed.

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

```
&i2c5 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c5_pins_a>;
    pinctrl-1 = <&i2c5_pins_sleep_a>;
    i2c-scl-rising-time-ns = <185>;
    i2c-scl-falling-time-ns = <20>;
    clock-frequency = <100000>;
    /* spare dmas for other usage */
    /delete-property/dmas;
    /delete-property/dma-names;
    status = "okay";

    cy8c9520a: cy8c9520a@20 {
        compatible = "cy8c9520a";
        reg = <0x20>;
        interrupt-controller;
        #interrupt-cells = <2>;
        gpio-controller;
        #gpio-cells = <2>;
        interrupt-parent = <&gpio>;
        interrupts = <3 IRQ_TYPE_LEVEL_HIGH>;
    };

    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>;
};

adx1345@1d {
    compatible = "arrow,adx1345";
    reg = <0x1d>;
};

ltc3206: ltc3206@1b {
    compatible = "arrow,ltc3206";
    reg = <0x1b>;
    gpios = <&gpio3 GPIO_ACTIVE_LOW>;

    led1r {
        label = "red";
    };

    led1b {
        label = "blue";
    };

    led1g {
        label = "green";
    };

    ledmain {
        label = "main";
    };

    ledsub {
        label = "sub";
    };
};
};

```

## 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 matches with the compatible property of the cy8c9520a DT node:

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

5. Define an array of 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 `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 STM32MP1 processor.

**Note:** The "GPIO expander device" driver source code developed for the STM32MP157C-DK2 board is included in the `linux_5.4_CY8C9520A_driver.zip` file and can be downloaded from the GitHub repository at [https://github.com/ALIBERA/linux\\_book\\_2nd\\_edition](https://github.com/ALIBERA/linux_book_2nd_edition)

## Listing 7-4: CY8C9520A\_stm32mp1.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 output (out == 1)
     */
    pins |= BIT(cypress_get_offs(gpio, port));

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

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

```

```

    return ret;
}

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

/*
 * function to sync and unlock slow bus (i2c) chips
 * REG_INTR_MASK register is accessed via I2C
 * write 0 to the interrupt mask register line to
 * activate the interrupt on the GPIO
 */
static void cy8c9520a_irq_bus_sync_unlock(struct irq_data *d)
{
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    int ret, i;
    unsigned int gpio;
    u8 port;
    dev_info(chip->parent, "cy8c9520a_irq_bus_sync_unlock is called\n");

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

    /* irq_mask_cache stores the last value of irq_mask for each port */
    for (i = 0; i < NPORTS; i++) {
        /*
         * check if some of the bits have changed from the last cached value
         * irq_mask registers were initialized in cy8c9520a_irq_setup()
         */
        if (cygpio->irq_mask_cache[i] ^ cygpio->irq_mask[i]) {
            dev_info(chip->parent, "gpio %u is unmasked\n", gpio);
            cygpio->irq_mask_cache[i] = cygpio->irq_mask[i];
            ret = i2c_smbus_write_byte_data(cygpio->client,
                                           REG_PORT_SELECT, i);

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

            /* enable the interrupt for the GPIO unmasked */
            ret = i2c_smbus_write_byte_data(cygpio->client, REG_INTR_MASK,

```



```

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

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

}

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

/*
 * mask (disable) the GPIO interrupt.
 * In the initial setup all the int lines are masked
 */
static void cy8c9520a_irq_mask(struct irq_data *d)
{
    u8 port;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    unsigned gpio = d->hwirq;
    port = cypress_get_port(gpio);
    dev_info(chip->parent, "cy8c9520a_irq_mask is called\n");

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

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

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

```

```

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

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

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

err:
    return ret;
}

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

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

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

    /*
     * store in stat and clear (to enable ints)
     * the three interrupt status registers by reading them
     */

```

```

ret = i2c_smbus_read_i2c_block_data(cygpio->client,
                                     REG_INTR_STAT_PORT0,
                                     NPORTS, stat);

if (ret < 0) {
    memset(stat, 0, sizeof(stat));
}

ret = IRQ_NONE;

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

    /*
     * In every port check the GPIOs that have their 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_stm32mp1_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_stm32mp1_gpio` device that should be included in the `stm32mp15xx-dkx.dtsi` DT file. Check the differences with the `int_key` node of the LAB 7.1: "button interrupt device" Module that was taken as a reference for the development of this driver.

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

```

int_key {
    compatible = "arrow,intkey";
    pinctrl-names = "default";

```

```

    pinctrl-0 = <&key_pins>;
    label = "PB_USER";
    gpios = <&gpioa 14 GPIO_ACTIVE_LOW>;
    interrupt-parent = <&gpioa>;
    interrupts = <14 IRQ_TYPE_EDGE_FALLING>;
};

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 STM32MP1 processor.

**Note:** The "GPIO child device" driver source code developed for the STM32MP157C-DK2 board is included in the linux\_5.4\_CY8C9520A\_driver.zip file 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](https://github.com/ALIBERA/linux_book_2nd_edition)

## Listing 7-5: int\_stm32mp1\_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@arrows-europe.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/gpiochip10.

**Note:** The "GPIO based IRQ application" source code developed for the STM32MP157C-DK2 board is included in the linux\_5.4\_CY8C9520A\_driver.zip file under the app folder and can be downloaded from the GitHub repository at [https://github.com/ALIBERA/linux\\_book\\_2nd\\_edition](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/gpiochip10"

```

```

#define POLL_TIMEOUT -1 /* No timeout */

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

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

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

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

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

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

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

        /* poll gpio line event */

```

```

    ret = poll(&fdset, 1, POLL_TIMEOUT);
    if (ret <= 0)
        continue;

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

/* close gpio */
close(fd);

return 0;
}

```

## LAB 7.4 driver demonstration

Download the `linux_5.4_CY8C9520A_driver.zip` file from the github of the book and unzip it in the `STM32MP15-Ecosystem-v2.0.0` folder of the Linux host:

```
PC:~$ cd ~/STM32MP15-Ecosystem-v2.0.0/
```

Compile and deploy the drivers and the application to the **STM32MP157C-DK2** Discovery kit:

```

~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_driver$ make
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_driver$ make deploy
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_driver/linux_5.4_gpio_int_driver$
make
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_driver/linux_5.4_gpio_int_driver$
make deploy
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_driver_rev/app$ make
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_driver/app$ make deploy

```

Follow the next instructions to test the drivers:

```

/* load the CY8C9520A module */
root@stm32mp1:~# insmod CY8C9520A_stm32mp1.ko
[ 373.764568] CY8C9520A_stm32mp1: loading out-of-tree module taints kernel.
[ 373.771612] cy8c9520a 1-0020: cy8c9520a_probe() function is called
[ 373.780939] cy8c9520a 1-0020: dev_id=0x20
[ 373.802297] cy8c9520a 1-0020: the cy8c9520a_setup is done
[ 373.806820] cy8c9520a 1-0020: the cy8c9520a_irq_setup function is entered
[ 373.816303] cy8c9520a 1-0020: the interrupt state registers are cleared

```

```
[ 373.831687] cy8c9520a 1-0020: the interrupt mask port registers are set
[ 373.837375] cy8c9520a 1-0020: the interrupt setup is done
```

```
/* Print information of all the lines of the gpiochip10 */
```

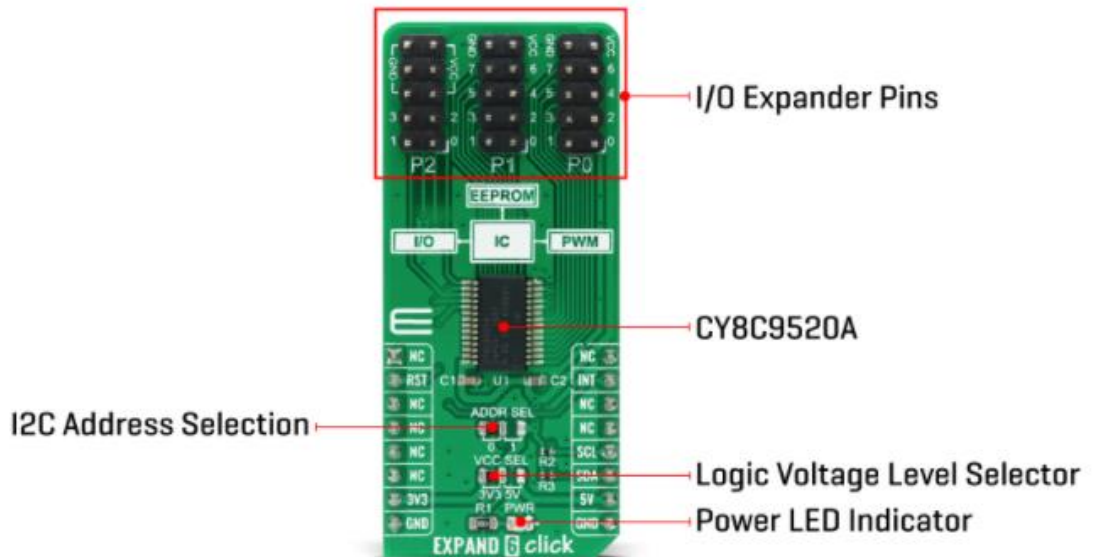
```
root@stm32mp1:~# gpioinfo gpiochip10
```

```
gpiochip10 - 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 gpiochip10 are configured with internal pull-up to Vcc \*/



/\* set to high level the pin 1 of P0 \*/

```
root@stm32mp1:~# gpioset gpiochip10 1=1
```

```
[ 318.304216] cy8c9520a 1-0020: cy8c9520a_gpio_direction output is called
```

```
[ 318.315157] cy8c9520a 1-0020: cy8c9520a_gpio_set_value function with 1 value is called
```

/\* check the value received in the pin 0 of P0 \*/

```
root@stm32mp1:~# gpioget gpiochip10 0
```

```
[ 322.317713] cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
```

```
[ 322.328294] cy8c9520a 1-0020: cy8c9520a_gpio_get function is called
```

```
[ 322.333244] cy8c9520a 1-0020: the in_reg address is 0
```

```
[ 322.340352] cy8c9520a 1-0020: cy8c9520a_gpio_get function with 255 value is returned
```

1

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@stm32mp1:~# insmod int_stm32mp1_gpio.ko
[ 65.355362] int_gpio_expand int_gpio: my_probe() function is called.
[ 65.360469] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 65.377282] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[ 65.382796] cy8c9520a 1-0020: gpio 0 is unmasked
[ 65.396884] cy8c9520a 1-0020: the REG_INTR_MASK value is 254
[ 65.401255] int_gpio_expand int_gpio: IRQ_using_platform_get_irq: 85
[ 65.413756] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 65.418157] cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
[ 65.424004] cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
[ 65.436638] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[ 65.443218] int_gpio_expand int_gpio: mydev: got minor 61
[ 65.451734] int_gpio_expand int_gpio: my_probe() function is exited.

```

```

/* check the gpio interrupt with Linux IRQ number 85 */

```

```

root@stm32mp1:~# cat /proc/interrupts

```

	CPU0	CPU1			
18:	12523	12899	GIC-0	27 Level	arch_timer
20:	0	0	GIC-0	232 Level	arm-pmu
21:	0	0	GIC-0	233 Level	arm-pmu
22:	0	0	GIC-0	68 Level	4000b000.audio-controller
23:	0	0	GIC-0	126 Level	40016000.cec
24:	0	0	GIC-0	116 Level	44005000.spi
25:	0	0	GIC-0	123 Level	4400b004.audio-controller,
4400b024.audio-controller					
26:	0	0	GIC-0	43 Level	dma1chan0
27:	106	0	GIC-0	44 Level	dma1chan1
28:	218	0	GIC-0	45 Level	dma1chan2
29:	215	0	GIC-0	46 Level	dma1chan3
30:	0	0	GIC-0	47 Level	dma1chan4
31:	0	0	GIC-0	48 Level	dma1chan5
32:	0	0	GIC-0	49 Level	dma1chan6
33:	0	0	GIC-0	79 Level	dma1chan7
34:	0	0	GIC-0	88 Level	dma2chan0
35:	0	0	GIC-0	89 Level	dma2chan1
36:	0	0	GIC-0	90 Level	dma2chan2
37:	0	0	GIC-0	91 Level	dma2chan3
38:	0	0	GIC-0	92 Level	dma2chan4
39:	0	0	GIC-0	100 Level	dma2chan5
40:	0	0	GIC-0	101 Level	dma2chan6
41:	0	0	GIC-0	102 Level	dma2chan7
42:	0	0	GIC-0	37 Level	rcc irq
44:	0	0	GIC-0	179 Level	stm_thermal
45:	0	0	GIC-0	112 Level	54002000.hash
46:	0	0	GIC-0	154 Level	58000000.dma
47:	11811	0	GIC-0	81 Level	mmci-pl118x (cmd)
48:	6948	0	GIC-0	156 Level	mmci-pl118x (cmd)
49:	458	0	GIC-0	93 Level	eth0

50:	1214	0	GIC-0 120 Level	5a001000.display-controller
51:	0	0	GIC-0 121 Level	5a001000.display-controller
53:	0	0	GIC-0 111 Level	54001000.cryp
54:	172	0	stm32-exti-h-direct	27 Level 4000e000.serial
55:	1073	0	stm32-exti-h-direct	30 Level 40010000.serial
56:	0	0	stm32-exti-h-direct	70 Level eth0
57:	25	0	stm32-exti-h-direct	43 Level ehci_hcd:usb2
58:	0	0	stm32-exti-h-direct	19 Level 5c004000.rtc
59:	525	0	stm32-exti-h-direct	21 Level 40012000.i2c
60:	0	0	GIC-0 64 Level	40012000.i2c
61:	0	0	stm32gpio 1 Edge	0-0039
62:	0	0	stm32gpio 2 Edge	ft6236
63:	92	0	stm32-exti-h-direct	25 Level 40015000.i2c
64:	0	0	GIC-0 140 Level	40015000.i2c
65:	296	0	stm32-exti-h-direct	24 Level 5c002000.i2c
66:	0	0	GIC-0 128 Level	5c002000.i2c
67:	0	0	stm32gpio 11 Edge	2-0028
68:	0	0	stm32-exti-h 55 Edge	pmic_irq
69:	0	0	pmic_irq 16 Edge	
5c002000.i2c:stpmic@33:regulators				
70:	0	0	pmic_irq 17 Edge	
5c002000.i2c:stpmic@33:regulators				
71:	0	0	pmic_irq 19 Edge	
5c002000.i2c:stpmic@33:regulators				
72:	0	0	pmic_irq 20 Edge	
5c002000.i2c:stpmic@33:regulators				
73:	0	0	pmic_irq 21 Edge	
5c002000.i2c:stpmic@33:regulators				
74:	0	0	pmic_irq 14 Edge	
5c002000.i2c:stpmic@33:regulators				
75:	0	0	pmic_irq 12 Edge	
5c002000.i2c:stpmic@33:regulators				
76:	0	0	pmic_irq 13 Edge	
5c002000.i2c:stpmic@33:regulators				
77:	0	0	pmic_irq 0 Edge	5c002000.i2c:stpmic@33:onkey
78:	0	0	pmic_irq 1 Edge	5c002000.i2c:stpmic@33:onkey
79:	0	0	stm32gpio 7 Edge	58005000.sdmmc cd
80:	0	0	stm32-exti-h-direct	61 Edge 4c001000.mailbox
81:	0	0	GIC-0 133 Level	4c001000.mailbox
82:	0	0	stm32-exti-h 68 Edge	mlahb:m4@10000000
83:	0	0	stm32-exti-h-direct	44 Level 49000000.usb-otg,
49000000.usb-otg, dwc2_hsotg:usb1				
84:	0	0	stm32gpio 3 Level	1-0020
85:	0	0	cy8c9520a-irq 0 Edge	P0_line0_INT
IPI0:	0	0	CPU wakeup interrupts	
IPI1:	0	0	Timer broadcast interrupts	
IPI2:	1978	9850	Rescheduling interrupts	
IPI3:	145	126	Function call interrupts	

```
IPI4:          0          0 CPU stop interrupts
IPI5:        2789        2965 IRQ work interrupts
IPI6:          0          0 completion interrupts
Err:           0
root@stm32mp1:~#
```

```
/* Connect pin 0 of P0 to GND, then disconnect it from GND. Two interrupts are
fired */
```

```
root@stm32mp1:~# [ 109.462672] the interrupt ISR has been entered
[ 109.468622] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 123.566674] the interrupt ISR has been entered
[ 123.572607] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
```

```
/* remove the gpio int module */
```

```
root@stm32mp1:~# rmmod int_stm32mp1_gpio.ko
[ 226.358291] int_gpio_expand int_gpio: my_remove() function is called.
[ 226.366632] int_gpio_expand int_gpio: my_remove() function is exited.
[ 226.371759] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 226.377619] cy8c9520a 1-0020: cy8c9520a_irq_mask is called
[ 226.382987] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[ 226.389667] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 226.395447] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
```

```
/* remove the CY8C9520A module */
```

```
root@stm32mp1:~# rmmod CY8C9520A_stm32mp1.ko
[ 299.364202] cy8c9520a 1-0020: cy8c9520a_remove() function is called
```

**Handle GPIO INT in line 0 of P0 using a GPIO based interrupt application**

```
/* load the CY8C9520A module */
```

```
root@stm32mp1:~# insmod CY8C9520A_stm32mp1.ko
```

```
/* Launch the gpiomon application */
```

```
root@stm32mp1:~# gpiomon --falling-edge gpiochip10 0
[ 75.144142] cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
[ 75.157681] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 75.162219] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[ 75.172701] cy8c9520a 1-0020: gpio 0 is unmasked
[ 75.183485] cy8c9520a 1-0020: the REG_INTR_MASK value is 254
[ 75.190147] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 75.196890] cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
[ 75.201293] cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
[ 75.211559] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
```

```
/* Now connect pin 0 of P0 to GND. An interrupt is fired */
```

```
[ 133.764344] the interrupt ISR has been entered
event: FALLING EDGE offset: 0 timestamp: [1581090779.659029102]
```

```
/* Disconnect pin 0 of P0 from GND. An interrupt is fired */
```

```
[ 134.022438] the interrupt ISR has been entered
event: FALLING EDGE offset: 0 timestamp: [1581090779.917628185]
```

```
/* Exit application with Ctrl+C */
```

```
^C[ 272.381756] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called  
[ 272.386313] cy8c9520a 1-0020: cy8c9520a_irq_mask is called  
[ 272.391818] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called  
[ 272.398294] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called  
[ 272.404159] 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@stm32mp1:~# ./gpio_int
```

```
[ 57.807981] cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called  
[ 57.819284] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called  
[ 57.824075] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called  
[ 57.830155] cy8c9520a 1-0020: gpio 0 is unmasked  
[ 57.849225] cy8c9520a 1-0020: the REG_INTR_MASK value is 254  
[ 57.856408] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called  
[ 57.860806] cy8c9520a 1-0020: cy8c9520a_irq_set_type is called  
[ 57.866652] cy8c9520a 1-0020: cy8c9520a_irq_unmask is called  
[ 57.882486] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called  
[ 69.954568] the interrupt ISR has been entered  
[ 69.960525] cy8c9520a 1-0020: cy8c9520a_gpio_get function is called  
[ 69.965476] cy8c9520a 1-0020: the in_reg address is 0  
[ 69.972588] cy8c9520a 1-0020: cy8c9520a_gpio_get function with 254 value is
```

```
returned
```

```
irq received.
```

```
id: 2, timestamp: 1581090715846627697
```

```
[ 90.880925] the interrupt ISR has been entered  
[ 90.886890] cy8c9520a 1-0020: cy8c9520a_gpio_get function is called  
[ 90.891738] cy8c9520a 1-0020: the in_reg address is 0  
[ 90.898983] cy8c9520a 1-0020: cy8c9520a_gpio_get function with 255 value is
```

```
returned
```

```
irq received.
```

```
id: 1, timestamp: 1581090736772991498
```

```
^C[ 104.312407] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called  
[ 104.317061] cy8c9520a 1-0020: cy8c9520a_irq_mask is called  
[ 104.322305] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called  
[ 104.328875] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called  
[ 104.334756] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
```

# 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\_stm32mp1 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));

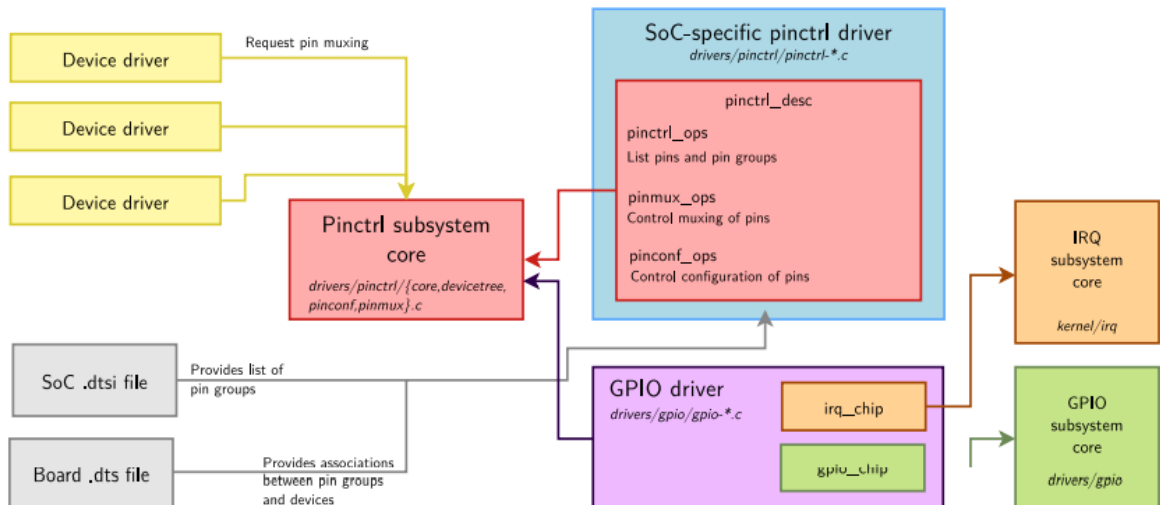
    i2c_smbus_write_byte_data(client, REG_DRIVE_PULLUP + offs, val);

}

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>;
    interrupt-parent = <&gpio>;
    interrupts = <3 IRQ_TYPE_LEVEL_HIGH>;

    pinctrl-names = "default";
    pinctrl-0 = <&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 gpio0 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 sysfs 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>;
    interrupt-parent = <&gpio0>;
    interrupts = <3 IRQ_TYPE_LEVEL_HIGH>;

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

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

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

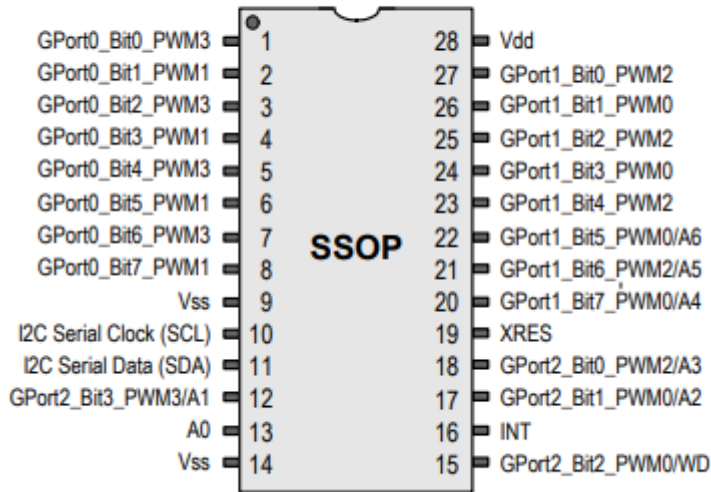
    cy8c9520apulldowns: pinmux2 {
        pins = "gpio2";
        bias-pull-down;
    };

    /* 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 STM32MP1 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 STM32MP157C-DK2 board is included in the linux\_5.4\_CY8C9520A\_pwm\_pinctrl.zip file and can be downloaded from the GitHub repository at [https://github.com/ALIBERA/linux\\_book\\_2nd\\_edition](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;

```

[illegible]

```

    u8 pins, port;

    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

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

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

    mutex_lock(&cygpio->lock);

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

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

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

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

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

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

/* function to lock access to slow bus (i2c) chips */
static void cy8c9520a_irq_bus_lock(struct irq_data *d)
{
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);

```

```

    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    dev_info(chip->parent, "cy8c9520a_irq_bus_lock is called\n");
    mutex_lock(&cygpio->irq_lock);
}

/*
 * function to sync and unlock slow bus (i2c) chips
 * REG_INTR_MASK register is accessed via I2C
 * write 0 to the interrupt mask register line to
 * activate the interrupt on the GPIO
 */
static void cy8c9520a_irq_bus_sync_unlock(struct irq_data *d)
{
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    int ret, i;
    unsigned int gpio;
    u8 port;
    dev_info(chip->parent, "cy8c9520a_irq_bus_sync_unlock is called\n");
    gpio = d->hwirq;
    port = cypress_get_port(gpio);

    /* irq_mask_cache stores the last value of irq_mask for each port */
    for (i = 0; i < NPORTS; i++) {
        /*
         * check if some of the bits have changed from the last cached value
         * irq_mask registers were initialized in cy8c9520a_irq_setup()
         */
        if (cygpio->irq_mask_cache[i] ^ cygpio->irq_mask[i]) {
            dev_info(chip->parent, "gpio %u is unmasked\n", gpio);
            cygpio->irq_mask_cache[i] = cygpio->irq_mask[i];
            ret = i2c_smbus_write_byte_data(cygpio->client,
                                            REG_PORT_SELECT, i);

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

            /* enable the interrupt for the GPIO unmasked */
            ret = i2c_smbus_write_byte_data(cygpio->client, REG_INTR_MASK,
                                            cygpio->irq_mask[i]);

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

            ret = i2c_smbus_read_byte_data(cygpio->client, REG_INTR_MASK);

```

```

        dev_info(chip->parent, "the REG_INTR_MASK value is %d\n", ret);
    }
}

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

/*
 * mask (disable) the GPIO interrupt.
 * In the initial setup all the int lines are masked
 */
static void cy8c9520a_irq_mask(struct irq_data *d)
{
    u8 port;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    unsigned gpio = d->hwirq;
    port = cypress_get_port(gpio);
    dev_info(chip->parent, "cy8c9520a_irq_mask is called\n");

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

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

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

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

```

```

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

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

err:
    return ret;
}

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

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

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

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

    if (ret < 0) {
        memset(stat, 0, sizeof(stat));
    }
}

```



```

ret = IRQ_NONE;

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

    /*
     * In every port check the GPIOs that have their 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 *cygprio)
{
    struct i2c_client *client = cygprio->client;
    struct gpio_chip *chip = &cygprio->gpio_chip;
    u8 dummy[NPORTS];
    int ret, i;

    mutex_init(&cygprio->irq_lock);

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

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

    if (ret < 0) {
        dev_err(&client->dev, "couldn't clear int status\n");
        goto err;
    }

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

    /*
     * 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(cygprio->irq_mask_cache, 0xff, sizeof(cygprio->irq_mask_cache));
    memset(cygprio->irq_mask, 0xff, sizeof(cygprio->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 pinctrl descriptor is done\n");

```

```

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

err:
    mutex_destroy(&cygpio->lock);

    return ret;
}

static int cy8c9520a_remove(struct i2c_client *client)
{
    struct cy8c9520a *cygpio = i2c_get_clientdata(client);
    dev_info(&client->dev, "cy8c9520a_remove() function is called\n");
    return pwmchip_remove(&cygpio->pwm_chip);
}

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

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

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

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

```

## LAB 7.5 driver demonstration

Download the linux\_5.4\_CY8C9520A\_pwm\_pinctrl.zip file from the github of the book and unzip it in the STM32MP15-Ecosystem-v2.0.0 folder of the Linux host:

```
PC:~$ cd ~/STM32MP15-Ecosystem-v2.0.0/
```

Compile and deploy the drivers and the application to the STM32MP157C-DK2 Discovery kit:

```
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_pwm_pinctrl$ make
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_pwm_pinctrl$ make deploy
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_pwm_pinctrl
/linux_5.4_gpio_int_driver$ make
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_pwm_pinctrl
/linux_5.4_gpio_int_driver$ make deploy
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_pwm_pinctrl/app$ make
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_CY8C9520A_pwm_pinctrl/app$ make deploy
```

Follow the next instructions to test the drivers:

```
/* load the CY8C9520A module */
root@stm32mp1:~# insmod CY8C9520A_stm32mp1.ko
[ 61.100977] CY8C9520A_stm32mp1: loading out-of-tree module taints kernel.
[ 61.117227] cy8c9520a 1-0020: cy8c9520a_probe() function is called
[ 61.124602] cy8c9520a 1-0020: dev_id=0x20
[ 61.159082] cy8c9520a 1-0020: the cy8c9520a_setup is done
[ 61.163046] cy8c9520a 1-0020: the initial setup for the cy8c9520a is done
[ 61.176696] cy8c9520a 1-0020: the setup for the cy8c9520a gpio controller done
[ 61.182852] cy8c9520a 1-0020: the cy8c9520a_irq_setup function is entered
[ 61.197225] cy8c9520a 1-0020: the interrupt state registers are cleared
[ 61.214573] cy8c9520a 1-0020: the interrupt mask port registers are set
[ 61.226347] cy8c9520a 1-0020: the interrupt setup is done
[ 61.230303] cy8c9520a 1-0020: the interrupt setup for the cy8c9520a is done
[ 61.242839] cy8c9520a 1-0020: the setup for the cy8c9520a pwm_chip controller
is done
[ 61.252514] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[ 61.259782] cy8c9520a 1-0020: The pin 0 drive mode is PIN_CONFIG_BIAS_PULL_UP
[ 61.274365] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[ 61.279203] cy8c9520a 1-0020: The pin 1 drive mode is PIN_CONFIG_BIAS_PULL_UP
[ 61.297723] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[ 61.303662] cy8c9520a 1-0020: The pin 2 drive mode is PIN_CONFIG_BIAS_PULL_DOWN
[ 61.318751] cy8c9520a 1-0020: cygpio_pinconf_set function is called
[ 61.323780] cy8c9520a 1-0020: The pin 3 drive mode is PIN_CONFIG_DRIVE_STRENGTH
[ 61.341859] 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@stm32mp1:~# insmod int_stm32mp1_gpio.ko
[ 65.355362] int_gpio_expand int_gpio: my_probe() function is called.
[ 65.360469] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 65.377282] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[ 65.382796] cy8c9520a 1-0020: gpio 0 is unmasked
[ 65.396884] cy8c9520a 1-0020: the REG_INTR_MASK value is 254
[ 65.401255] int_gpio_expand int_gpio: IRQ_using_platform_get_irq: 85
[ 65.413756] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 65.418157] cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
[ 65.424004] cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
[ 65.436638] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[ 65.443218] int_gpio_expand int_gpio: mydev: got minor 61
[ 65.451734] 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@stm32mp1:~# [ 109.462672] the interrupt ISR has been entered
[ 109.468622] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 123.566674] the interrupt ISR has been entered
[ 123.572607] 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@stm32mp1:~# cd /sys/class/pwm/

/* Each probed PWM controller will be exported as pwmchipN, where N is the base of
the PWM controller */
root@stm32mp1:/sys/class/pwm# ls
pwmchip0

root@stm32mp1:/sys/class/pwm# cd pwmchip0/

/* npwm is the number of PWM channels which this controller supports (read-only)
*/
root@stm32mp1:/sys/devices/platform/soc/40015000.i2c/i2c-1/1-0020/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@stm32mp1:/sys/devices/platform/soc/40015000.i2c/i2c-1/1-0020/pwm/pwmchip0#
echo 1 > export
[ 120.280266] 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@stm32mp1:/sys/devices/platform/soc/40015000.i2c/i2c-1/1-0020/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@stm32mp1:/sys/devices/platform/soc/40015000.i2c/i2c-1/1-0020/pwm/pwmchip0#
echo 100000 > pwm1/period
[ 190.972833] cy8c9520a 1-0020: cy8c9520a_pwm_config is called

/* Set the active time of the PWM signal (read/write). Value is in nanoseconds */
root@stm32mp1:/sys/devices/platform/soc/40015000.i2c/i2c-1/1-0020/pwm/pwmchip0#
echo 50000 > pwm1/duty_cycle
[ 231.675500] cy8c9520a 1-0020: cy8c9520a_pwm_config is called

/* Enable the PWM signal (read/write) where 0 = disabled and 1 = enabled */
root@stm32mp1:/sys/devices/platform/soc/40015000.i2c/i2c-1/1-0020/pwm/pwmchip0#
echo 1 > pwm1/enable
[ 260.425946] cy8c9520a 1-0020: cy8c9520a_pwm_enable is called
[ 260.430173] cy8c9520a 1-0020: cy8c9520a_gpio_direction output is called
[ 260.448119] 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 */
[ 167.456419] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.462015] the interrupt ISR has been entered
[ 167.469428] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.475114] the interrupt ISR has been entered
[ 167.482396] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.488073] the interrupt ISR has been entered
[ 167.495384] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.500980] the interrupt ISR has been entered
[ 167.511658] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.517733] the interrupt ISR has been entered
[ 167.524639] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.530254] the interrupt ISR has been entered
[ 167.537623] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.543252] the interrupt ISR has been entered
[ 167.550605] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.556309] the interrupt ISR has been entered
[ 167.563568] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.569640] the interrupt ISR has been entered
[ 167.576541] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.582178] the interrupt ISR has been entered
[ 167.589520] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.595193] the interrupt ISR has been entered
[ 167.602523] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
[ 167.608203] the interrupt ISR has been entered
[ 167.615532] int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
.....

/* remove the gpio int module */
root@stm32mp1:~# rmmod int_stm32mp1_gpio.ko
[ 226.358291] int_gpio_expand int_gpio: my_remove() function is called.
[ 226.366632] int_gpio_expand int_gpio: my_remove() function is exited.

```

```
[ 226.371759] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 226.377619] cy8c9520a 1-0020: cy8c9520a_irq_mask is called
[ 226.382987] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
[ 226.389667] cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
[ 226.395447] cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called

/* remove the CY8C9520A module */
root@stm32mp1:~# rmmod CY8C9520A_stm32mp1.ko
[ 299.364202] cy8c9520a 1-0020: cy8c9520a_remove() function is called
```