Linux Driver Development for Embedded Processors

ST STM32MP1 Practical Labs Setup

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

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

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

Follow the next steps to install the SDK:

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

```
PC:~$ mdir -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

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:
adxl345_stm32mp1.ko                                 100%   12KB  12.3KB/s   00:00
adxl345_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                        keyled_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
```

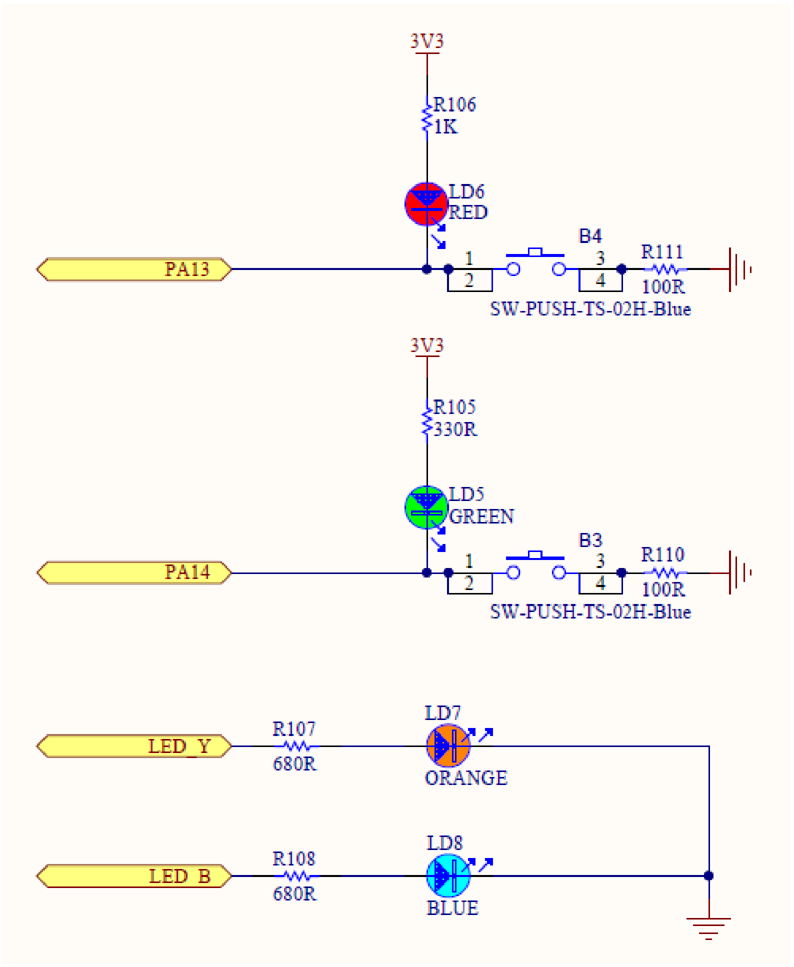
Hardware and device tree descriptions for the ST STM32MP1 labs

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,RGBclassleds";
	reg = <0x50002000 0x400>,
	      <0x50005000 0x400>;

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

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

	red {
		label = "ledred";
	};

	green {
		label = "ledgreen";
	};

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

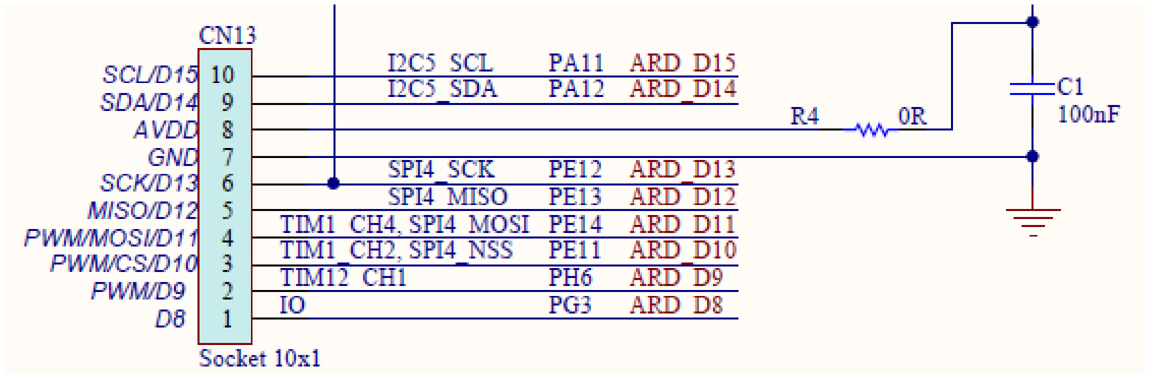
This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the driver for the LAB 5.4:

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

LAB 6.1 hardware and device tree descriptions

In this lab the driver will 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.

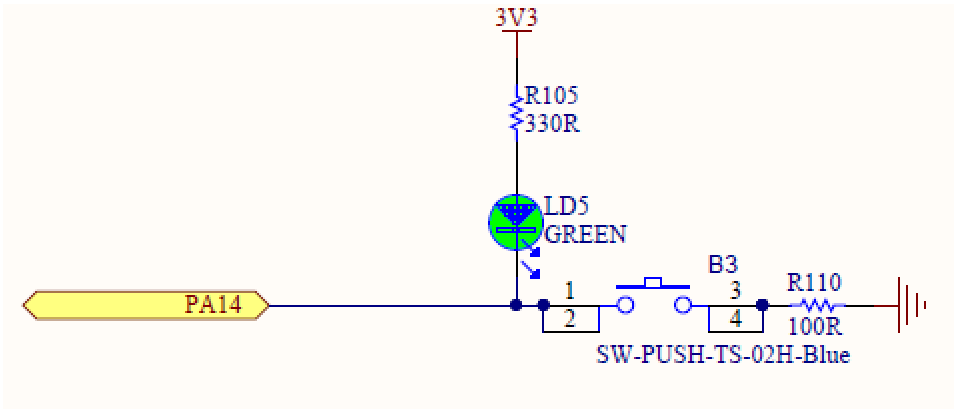
This is the device tree node that should be included in the stm32mp15xx-dkx.dtsi file to run the driver for the LAB 6.2:

```
    &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 = <&gpiog 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.



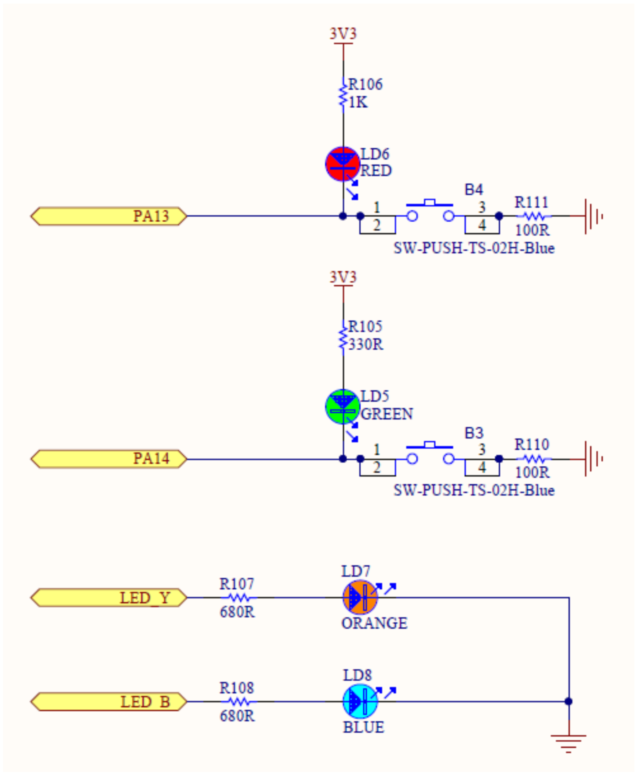
These are the device tree nodes that should be included in the stm32mp15xx-dkx.dtsi file to run the drivers for the LAB 7.1 and the LAB 7.2:

```
    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 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>;
        };

};
```

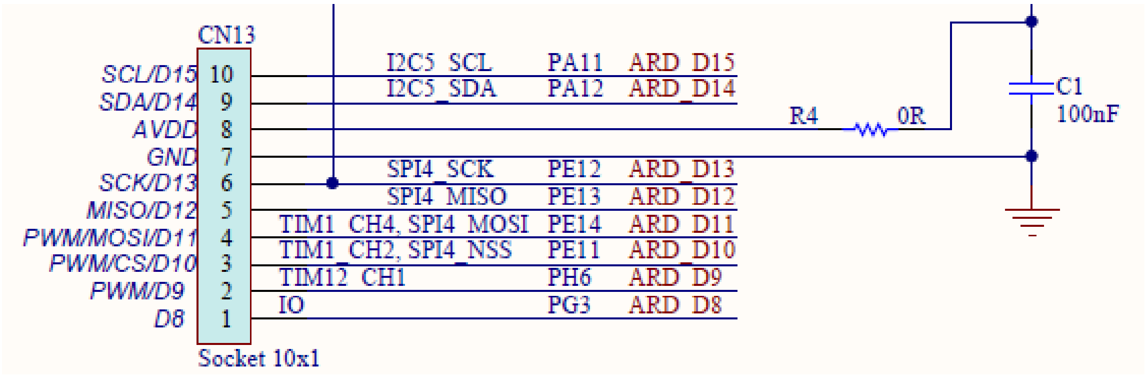
This is the device tree node that should be included in the stm32mp15-pinctrl.dtsi file to run the driver for the LAB 7.3:

```
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

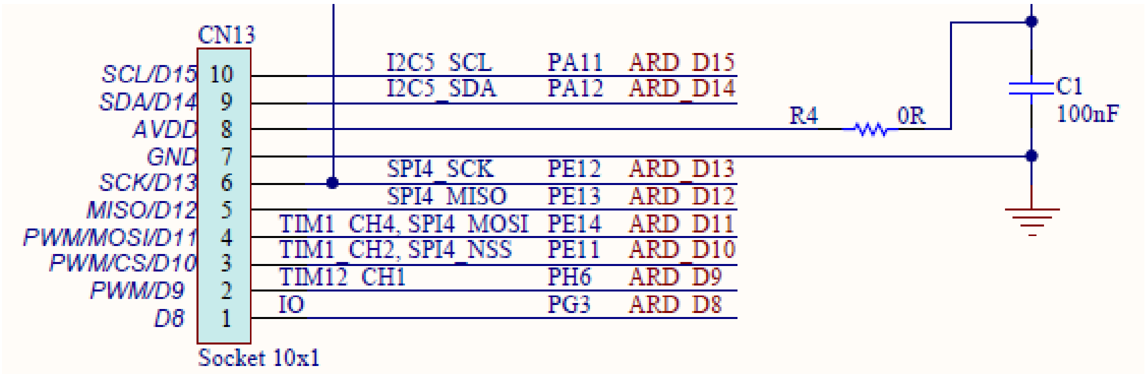
For this LAB 11.1 you will connect the I2C5 pins of the STM32MP157C-DK2 CN13 connector to the SDA and SCL pins of the LTC2607 DC934A evaluation board. You are going to power the LTC2607 with the 3.3V pin of the STM32MP157C-DK2 CN16 connector, connecting it to V+, pin 1 of the DC934A´s connector J1. Also 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

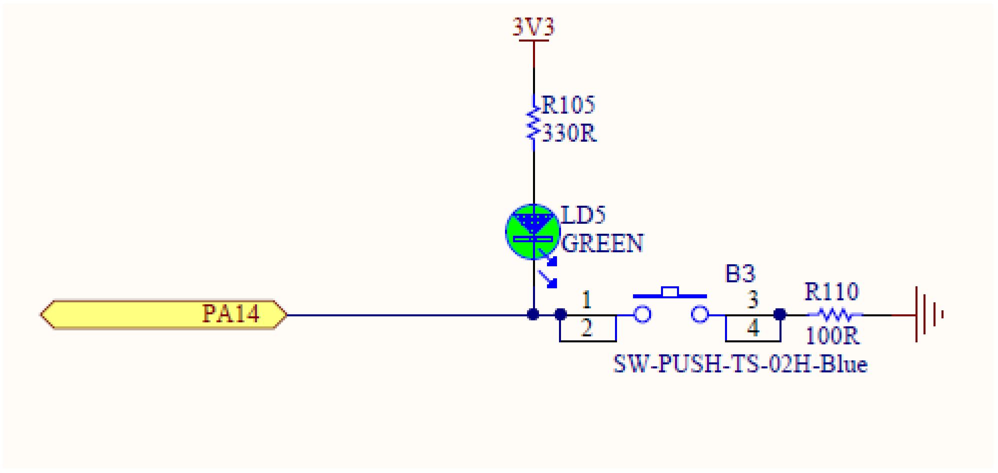
In these three labs you will reuse the hardware description of the LAB 11.1 and will use the SPI4 pins of the STM32MP157C-DK2 CN13 connector to connect to the LTC2422 dual ADC SPI device that is included in the DC934A board.


CN13
SCL/D15 10 I2C5_SCL PA11 ARD_D15
SDA/D14 9 I2C5_SDA PA12 ARD_D14
AVDD 8
R4 0R
C1 100nF
GND 7
SCK/D13 6 SPI4_SCK PE12 ARD_D13
MISO/D12 5 SPI4_MISO PE13 ARD_D12
PWM/MOSI/D11 4 TIM1_CH4, SPI4_MOSI PE14 ARD_D11
PWM/CS/D10 3 TIM1_CH2, SPI4_NSS PE11 ARD_D10
PWM/D9 2 TIM12_CH1 PH6 ARD_D9
D8 1 IO PG3 ARD_D8
Socket 10x1

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

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.

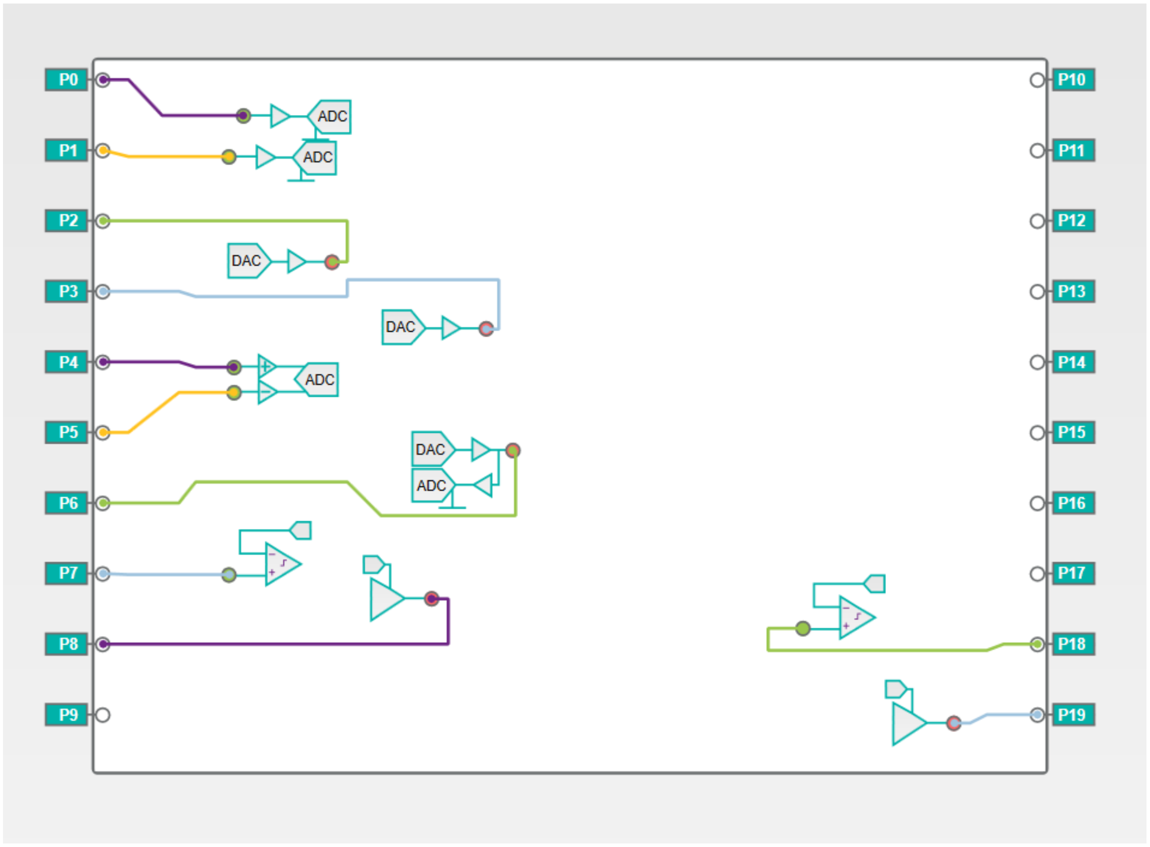
A new low cost evaluation board based on the MAX11300 device has been used, thus expanding the number of evaluation boards that can be tested during Chapter 11 to practice with the theory explained in this chapter.

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 will be in 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:

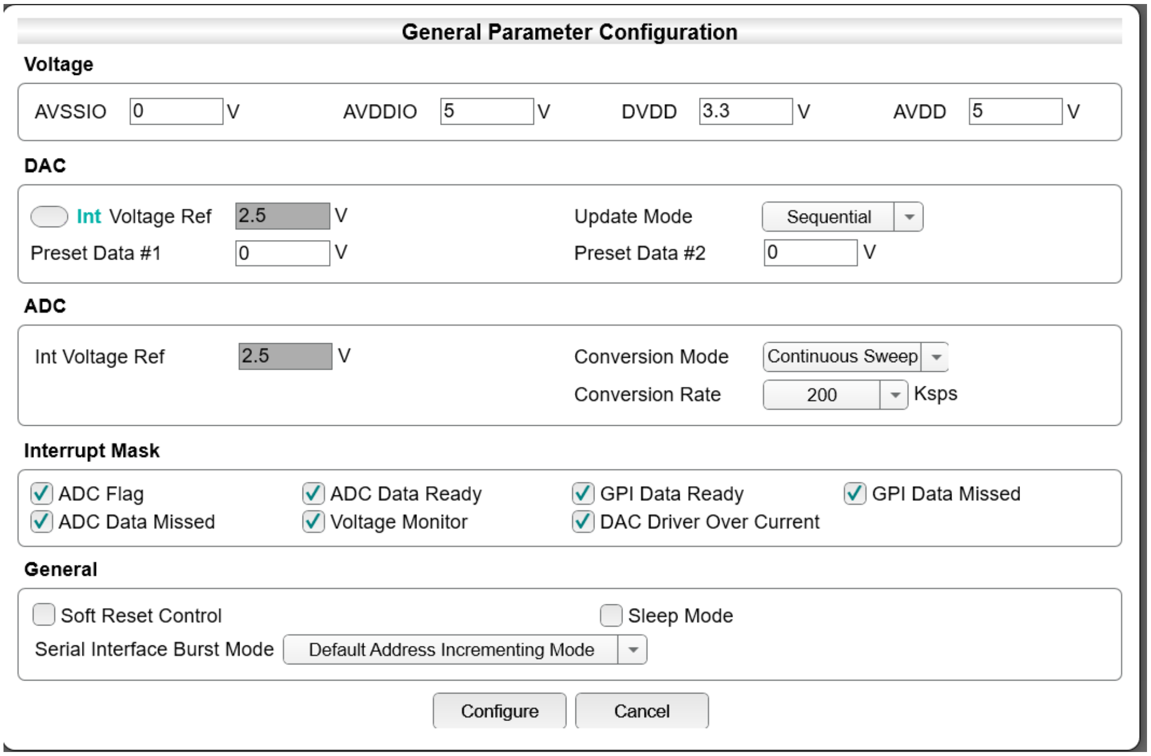


These are the parameters used during the configuration of the used ports of the MAX11300 device:

- **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

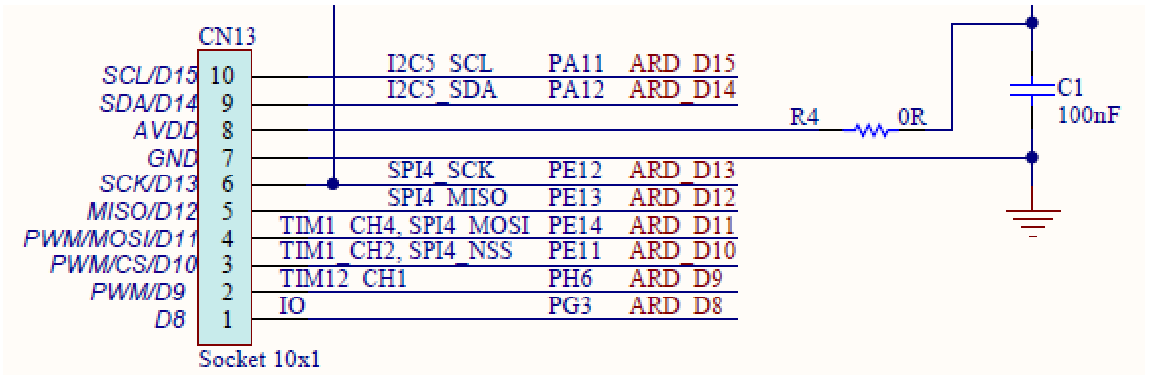
Not all the MAX11300 specifications were included during the development of this driver. These are the main specifications that have been included:

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

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

LAB 11.5 hardware description

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


CN13
SCL/D15 10 I2C5_SCL PA11 ARD_D15
SDA/D14 9 I2C5_SDA PA12 ARD_D14
AVDD 8
GND 7
SCK/D13 6 SPI4_SCK PE12 ARD_D13
MISO/D12 5 SPI4_MISO PE13 ARD_D12
PWM/MOSI/D11 4 TIM1_CH4, SPI4_MOSI PE14 ARD_D11
PWM/CS/D10 3 TIM1_CH2, SPI4_NSS PE11 ARD_D10
PWM/D9 2 TIM12_CH1 PH6 ARD_D9
D8 1 IO PG3 ARD_D8
R4 0R
C1 100nF
Socket 10x1

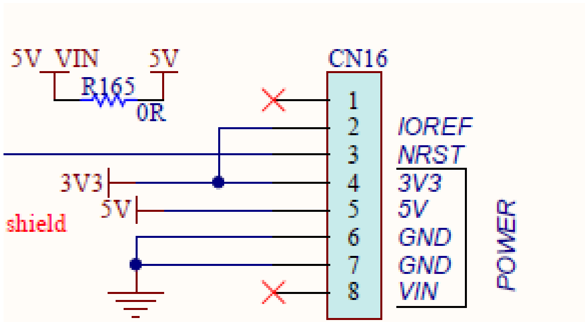
And the PIXI™ CLICK mikroBUS™ socket:

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

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)

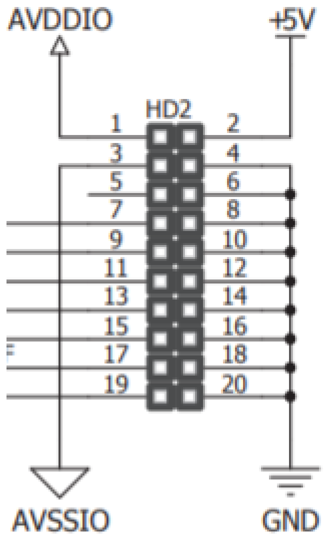
Now in the STM32MP157C-DK2 schematic find the CN16 connector:



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



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 pin muxing, so that the pins are configured as 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 this 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 from previous labs.

Now you will add 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 in the MAX11300 configuration 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 = <&gpiog 3 GPIO_ACTIVE_LOW>;
           interrupt-parent = <&gpiog>;
           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. This file will also be included within the source files of the max11300 driver. 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 categories: Industrial framework as a SPI interaction, Industrial framework as an IIO device and GPIO driver interface. The MAX11300 driver is based on Paul Cercueil´s AD5592R driver (https://elixir.bootlin.com/linux/latest/source/drivers/iio/dac/ad5592r.c)

Industrial framework as a SPI interaction

These are the main code sections:

1. Include the required header files:

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

2. Create a struct spi_driver structure:

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

3. Register to the SPI bus as a driver:

   ```
   module_spi_driver(max11300_spi_driver);
   ```

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

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

5. Define an array of struct spi_device_id structures:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

		*value = be16_to_cpu(st->rx_msg);

		return 0;
	}

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

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

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

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

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



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

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

        return 0;
}
```

Industrial framework as an IIO device

These are the main code sections:

1. Include the required header files:

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

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

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

```
        u8 tx_cmd; // command byte for SPI transactions
        __be16 tx_msg; // transmit value for SPI transactions in BE format
        __be16 rx_msg; // value received in SPI transactions in BE format
};
```

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

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

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

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

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

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

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

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

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

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

5. The max11300_alloc_ports() function will read the properties from the DT channel children nodes of the DT max11300 node 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 later 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 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 is setting 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 in 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 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.

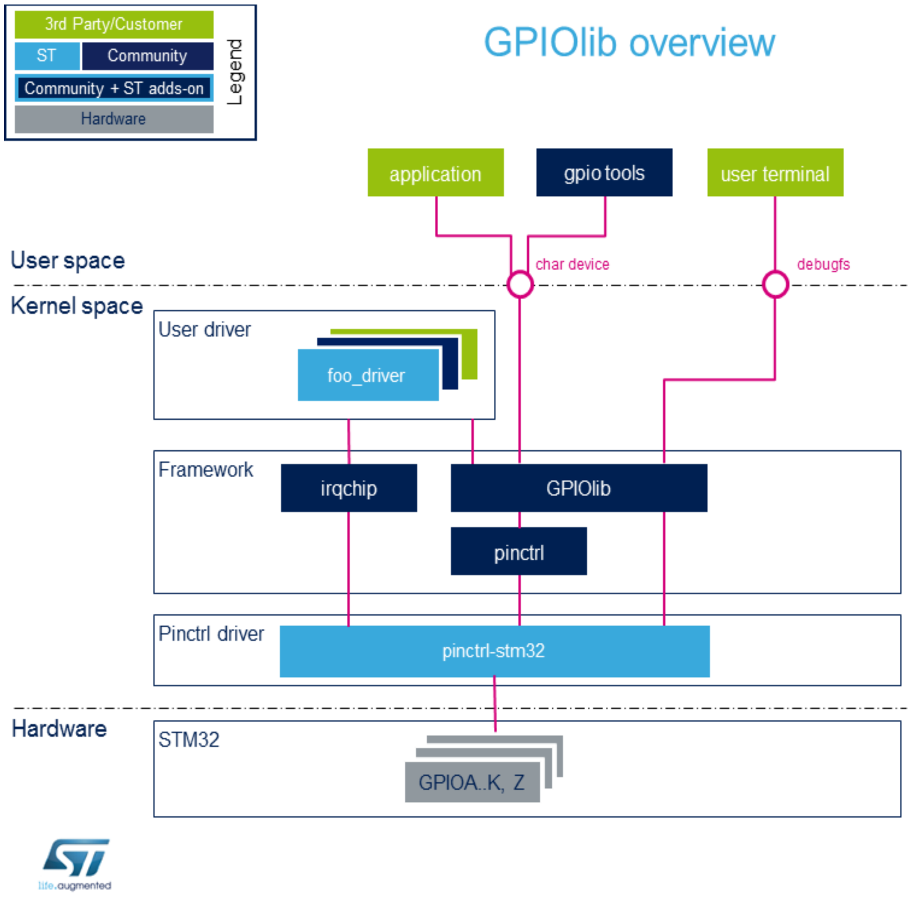
In the Chapter 5 of this book , you saw how to control GPIOs from kernel space using the GPIO descriptor consumer interface of the GPIOLib framework.

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 from the DT sources.

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

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



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

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

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

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

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

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

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

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

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

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

	mutex_lock(&st->gpio_lock);

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

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

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

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

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

		val = (int) read_val;

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

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

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

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

	mutex_lock(&st->gpio_lock);

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

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

	mutex_unlock(&st->gpio_lock);
}

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

        mutex_lock(&st->gpio_lock);


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

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

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

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

        mdelay(1);

err_unlock:
        mutex_unlock(&st->gpio_lock);

        return ret;
}

/*
 * struct gpio_chip direction_output callback function.
 * It configures the GPIO port as an output (GPO) writing to
 * the PORT_CFG register of the max11300 and sets output value of the
 * GPIO line with GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the GPO data registers of the max11300
 */
static int max11300_gpio_direction_output(struct gpio_chip *chip,
                                      unsigned int offset, int value)
{
        struct max11300_state *st = gpiochip_get_data(chip);
        int ret;
        u8 reg;
        u16 port_mode, val;

        mutex_lock(&st->gpio_lock);

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

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

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

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

        mutex_unlock(&st->gpio_lock);

        max11300_gpio_set(chip, offset, value);

        return ret;
}
```

See in the next **Listings** the complete " IIO Mixed-Signal I/O Device" driver source code for the 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

Listing 11-6: max11300-base.h

```
#ifndef __DRIVERS_IIO_DAC_max11300_BASE_H__
#define __DRIVERS_IIO_DAC_max11300_BASE_H__

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

struct max11300_state;

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

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

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

/* declare the global structure that will store the info of the device */
```

```
struct max11300_state {
   struct device *dev;
   const struct max11300_rw_ops *ops;
   struct gpio_chip gpiochip;
   struct mutex gpio_lock;
   u8 num_ports;
   u8 num_gpios;
   u8 gpio_offset[20];
   u8 gpio_offset_mode[20];
   u8 port_modes[20];
   u8 adc_range[20];
   u8 dac_range[20];
   u8 adc_reference[20];
   u8 adc_samples[20];
   u8 adc_negative_port[20];
   u8 tx_cmd;
   __be16 tx_msg;
   __be16 rx_msg;
};

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

#endif /* __DRIVERS_IIO_DAC_max11300_BASE_H__ */
```

Listing 11-7: maxim,max11300.h

```
#ifndef _DT_BINDINGS_MAXIM_MAX11300_H
#define _DT_BINDINGS_MAXIM_MAX11300_H

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

#define    ADC_SAMPLES_1        0
#define    ADC_SAMPLES_2        1
#define    ADC_SAMPLES_4        2
```

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



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

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

#endif /* _DT_BINDINGS_MAXIM_MAX11300_H */
```

Listing 11-8: max11300.c

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

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

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

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

   /* to transmit via SPI the LSB bit of the command byte must be 0 */
```

```
	st->tx_cmd = (reg << 1);

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

	st->tx_msg = cpu_to_be16(val);

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

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

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

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

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

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

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

```
   return 0;
}

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

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

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

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

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

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

   return 0;
}

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

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

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

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

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

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

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

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

Listing 11-9: max11300-base.c

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

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

#include "max11300-base.h"

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

   mutex_lock(&st->gpio_lock);

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

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

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

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

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

```
            val = 0;

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

            val = (int) read_val;

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

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

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

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

    mutex_lock(&st->gpio_lock);

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

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

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

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

	mutex_unlock(&st->gpio_lock);
}

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

	mutex_lock(&st->gpio_lock);

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

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

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

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

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

	mdelay(1);

err_unlock:
	mutex_unlock(&st->gpio_lock);

	return ret;
}

/*
 * struct gpio_chip direction_output callback function.
 * It configures the GPIO port as an output (GPO) writing to
 * the PORT_CFG register of max11300 and sets output value of the
 * GPIO line in GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the GPO data registers of max11300
 */
static int max11300_gpio_direction_output(struct gpio_chip *chip,
					  unsigned int offset, int value)
{
   struct max11300_state *st = gpiochip_get_data(chip);
   int ret;
   u8 reg;
   u16 port_mode, val;

   mutex_lock(&st->gpio_lock);

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

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

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

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

   mutex_unlock(&st->gpio_lock);

   max11300_gpio_set(chip, offset, value);

   return ret;
}

/*
 * Initialize the MAX11300 gpio controller (struct gpio_chip)
 * and register it to the kernel
 */
static int max11300_gpio_init(struct max11300_state *st)
{
   if (!st->num_gpios)
          return 0;
	st->gpiochip.label = "gpio-max11300";
	st->gpiochip.base = -1;
	st->gpiochip.ngpio = st->num_gpios;
	st->gpiochip.parent = st->dev;
	st->gpiochip.can_sleep = true;
	st->gpiochip.direction_input = max11300_gpio_direction_input;
	st->gpiochip.direction_output = max11300_gpio_direction_output;
	st->gpiochip.get = max11300_gpio_get;
	st->gpiochip.set = max11300_gpio_set;
	st->gpiochip.owner = THIS_MODULE;

	mutex_init(&st->gpio_lock);

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

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

	mutex_lock(&iio_dev->mlock);

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

			dev_info(st->dev,
		     "the value of adc cfg addr for channel %d in port mode %d is %x\n",
				i, st->port_modes[i], reg);
		if ((st->port_modes[i]) == PORT_MODE_5)
			val = (port_mode | dac_range);
		else
			val = (port_mode | dac_range | adc_reference);

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

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

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

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

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

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

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

		mdelay(1);

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

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

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

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

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

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

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

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

                        val = (port_mode | adc_range | adc_reference);

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

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

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

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

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

	reg = (PORT_DAC_DATA_BASE_ADDRESS + chan->channel);

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

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

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

```
    }

    return ret;
}

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

    reg = PORT_ADC_DATA_BASE_ADDRESS + chan->channel;

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

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

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

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

```
   return ret;
}

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

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

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

/* ADC in SE mode with positive voltage range and straight binary */
static void max11300_setup_port_7_mode(struct iio_dev *iio_dev,
                                       struct iio_chan_spec *chan, bool output,
                                       unsigned int id, unsigned long port_mode)
{
   chan->type = IIO_VOLTAGE;
   chan->indexed = 1;
   chan->address = port_mode;
   chan->output = output;
   chan->channel = id;
   chan->info_mask_separate = BIT(IIO_CHAN_INFO_RAW);
   chan->scan_type.sign = 'u';
   chan->scan_type.realbits = 12;
   chan->scan_type.storagebits = 16;
   chan->scan_type.endianness = IIO_BE;
   chan->extend_name = "mode_7_ADC";
}

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

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

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

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

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

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

u32 reg, tmp;
int ret;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        }

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

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

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

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

    return 0;
}

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

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

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

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

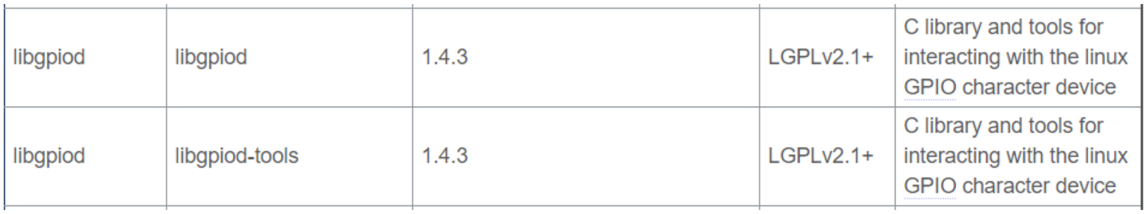
```
write_val = 0;

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

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

    return 0;

error_dev_unregister:
    iio_device_unregister(iio_dev);

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

error:
    return ret;
}
EXPORT_SYMBOL_GPL(max11300_probe);

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

    iio_device_unregister(iio_dev);

    return 0;
}
EXPORT_SYMBOL_GPL(max11300_remove);

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

LAB 11.5 driver demonstration

libgpiod provides a C library and simple tools for interacting with the linux GPIO character device. 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_drivers$ make
```

```
~/STM32MP15-Ecosystem-v2.0.0/linux_5.4_max11300_drivers$ 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:de
vice0# 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:de
vice0# 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:de
vice0# 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:de
vice0# 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:de
vice0# 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:de
vice0# 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:de
vice0# 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 (GPO) to port 18 (GPI)

/* Set port19 (GPO) 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 (GPO) 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:~# gpioset 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:~# gpioset 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:~# gpioset 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 ouput is set
[  549.290375] max11300 spi0.0: The GPIO ouput 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
```