

Chapter 2: Peripherals

At the end of this chapter you will be able to write firmware for MCU peripherals (e.g. GPIO, PWM, ADC, UART, I2C and SPI) and be able to interface with shield peripherals (e.g. pressure sensor and OLED display). In addition, you will understand the purposes of, and distinctions between the peripheral driver library (PDL) and the hardware abstraction layer (HAL).

Table of contents

2.1	Shield board support libraries	3
2.1.1	CY8CKIT-028-SENSE shield board	3
2.2	PDL vs. HAL.....	4
2.3	Peripherals.....	6
2.3.1	GPIO	7
2.3.2	PWM	10
2.3.3	ADC	14
2.3.4	UART	18
2.3.5	I ² C	24
2.3.6	SPI	27
2.3.7	OLED Display	31
2.4	Interrupts.....	32
2.4.1	Global Interrupt Enable	32
2.4.2	HAL.....	32
2.4.3	PDL.....	34
2.5	Exercises	38
	Exercise 1: (GPIO-HAL) Blink an LED	38
	Exercise 2: (GPIO-PDL) Blink an LED	39
	Exercise 3: (GPIO-HAL) Add debug printing to the LED blink project	40
	Exercise 4: (GPIO-PDL) Add debug printing to the LED blink project	41
	Exercise 5: (GPIO-HAL) Read the state of a mechanical button	42
	Exercise 6: (GPIO-PDL) Read the state of a mechanical button	43
	Exercise 7: (GPIO-HAL) Use an interrupt to toggle the state of an LED.....	44
	Exercise 8: (GPIO-PDL) Use an interrupt to toggle the state of an LED.....	45
	Exercise 9: (PWM-HAL) LED Brightness	46
	Exercise 10: (PWM-PDL) LED Brightness	47
	Exercise 11: (ADC READ-HAL) Read potentiometer sensor value via an ADC	48
	Exercise 12: (ADC READ-PDL) Read potentiometer sensor value via an ADC	49
	Exercise 13: (UART-HAL) Read a value using the standard UART functions	50
	Exercise 14: (UART-PDL) Read a value using the standard UART functions	51
	Exercise 15: (UART-HAL) Write a value using the standard UART functions.....	51
	Exercise 16: (UART-PDL) Write a value using the standard UART functions.....	52
	Exercise 17: Install shield support libraries and use the OLED display	53
	Exercise 18: (I2C READ-HAL) Read sensor values over I2C	54
	Exercise 19: (SPI-HAL) Transfer data using SPI.....	55
	Exercise 20: (SPI-PDL) Transfer data using SPI.....	56

Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	CY_ISR_PROTO(MyISR) ; make build
<i>Italics</i>	Displays file names and paths	<i>sourcefile.hex</i>
[bracketed, bold]	Displays keyboard commands in procedures	[Enter] or [Ctrl] [C]
Menu > Selection	Represents menu paths	File > New Project > Clone
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the Debugger icon, and then click Next .

2.1 Shield board support libraries

This will be used in [Exercise 17:](#) and [Exercise 18:](#)

ModusToolbox™ software includes libraries that make it easier to work with the peripherals on any given kit. In our case, we are using a KIT_XMC72_EVK along with a sense shield. You select the BSP for the baseboard when you select your kit's name in the Project Creator. To make it easier to interface with the shield, a support library has been created. Since this is not installed by default when creating a project, we need to add it.

After creating a new project, open the Library Manager and add the CY8CKIT-028-SENSE, emWin and other required libraries.

2.1.1 CY8CKIT-028-SENSE shield board

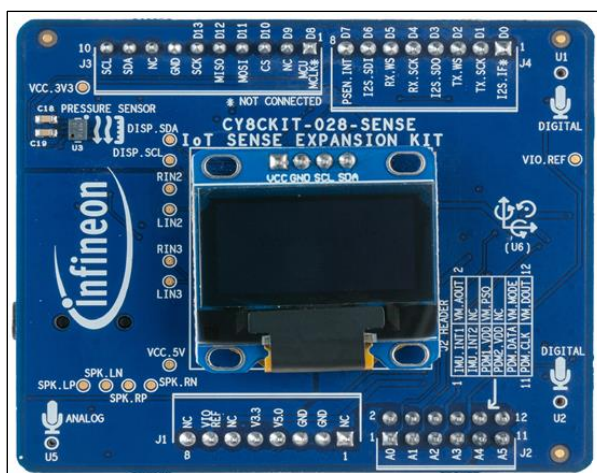
The Sense Shield Board (CY8CKIT-028-SENSE) has been designed as a companion to add common sensors, audio components and user interface to an Arduino based baseboard.

The shield library provides support for:

- Initializing/freeing all of the hardware peripheral resources on the board
- Defining all pin mappings from the Arduino interface to the different peripherals
- Providing access to each of the underlying peripherals on the board

This library makes use of support libraries: [display-oled-ssd1306](#), [sensor-orientation-bmx160](#), [sensor-xensiv-dps3xx](#), and [audio-codec-wm8960](#). **These can be seen in the libs directory and can be used directly, if desired.**

The Sense Shield Board uses the Arduino Uno pin layout plus an additional six pins.



You can see more details about this shield here: <https://www.infineon.com/cms/en/product/evaluation-boards/cy8ckit-028-sense/>

2.2 PDL vs. HAL

As you learned in the Modus Toolbox™ Software Training Level 1 - Getting Started class, Infineon provides two different libraries that allow you to more easily interact with the peripherals on a given device:

- Peripheral driver library (PDL)
- Hardware abstraction layer (HAL)

The PDL is a low-level device specific library that reduces the need to understand register usage and bit structures, thus easing software development for the extensive set of peripherals in the XMC7000 devices. For example, say you wanted to initialize a GPIO pin in a specific way. Rather than having to look up what bits in what registers need to be set, the PDL provides a convenient API to initialize your pin. You can find documentation for the PDL you are using in the Documentation section of the Eclipse IDE for ModusToolbox™ Quick Panel.

The HAL is a high-level, non-device-specific library that provides a generic interface to configure peripherals. The main goals of the HAL are ease-of-use and portability. As such, it abstracts the process of interacting with peripherals even more than the PDL. For example, say you wanted to set up a UART for debugging. When using the PDL, the configuration structure you need to populate to initialize the UART would look something like what is on the left in the following images. When using the HAL to initialize the same UART, the configuration structure you need to populate would look like what is on the right:

PDL UART Initialization Structure

```
const cy_stc_scb_uart_config_t UART_config =
{
    .uartMode = CY_SCB_UART_STANDARD,
    .enableMutliProcessorMode = false,
    .smartCardRetryOnNack = false,
    .irdaInvertRx = false,
    .irdaEnableLowPowerReceiver = false,
    .oversample = 8,
    .enableMsbFirst = false,
    .dataWidth = 8UL,
    .parity = CY_SCB_UART_PARITY_NONE,
    .stopBits = CY_SCB_UART_STOP_BITS_1,
    .enableInputFilter = false,
    .breakWidth = 11UL,
    .dropOnFrameError = false,
    .dropOnParityError = false,
    .breaklevel = false,
    .receiverAddress = 0x0UL,
    .receiverAddressMask = 0x0UL,
    .acceptAddrInFifo = false,
    .enableCts = false,
    .ctsPolarity = CY_SCB_UART_ACTIVE_LOW,
    .rtsRxFifoLevel = 0UL,
    .rtsPolarity = CY_SCB_UART_ACTIVE_LOW,
    .rxFifoTriggerLevel = 63UL,
    .rxFifoIntEnableMask = 0UL,
    .txFifoTriggerLevel = 63UL,
    .txFifoIntEnableMask = 0UL,
};
```

HAL UART Initialization Structure

```
const cyhal_uart_cfg_t uart_config =
{
    .data_bits = 8,
    .stop_bits = 1,
    .parity = CYHAL_UART_PARITY_NONE,
    .rx_buffer = NULL,
    .rx_buffer_size = 0
};
```

The HAL will also automatically set up other related items for a given peripheral. For example, when initializing a UART, instead of selecting and configuring a clock, the HAL allows you to specify NULL for the clock, which results in the HAL setting up an appropriate clock based on the chosen baud rate.

Likewise, the GPIO pins being used for the UART are configured automatically by the HAL, but they must be configured manually when using the PDL.

The HAL's focus on ease-of-use and portability means that it may not expose all of the low-level peripheral functionality. The HAL and PDL API's can however be used together within a single application.

You can leverage the HAL's simpler and more generic interface for most peripherals even if interactions with some peripherals require finer-grained control. You can find documentation for the HAL in the Documentation section of the Eclipse IDE for Modus Toolbox™ Quick Panel.

The HAL is built on top of the PDL. Therefore, you cannot include the HAL in an application without also including the PDL. For this reason, flash memory limited applications often choose to only use the PDL and exclude the HAL entirely. In this chapter we will look at how to initialize and use GPIOs, PWMs, ADCs, and UART using both the PDL and the HAL.

The abbreviation "Cat" stands for "Category." Since the PDL is architecture specific, different categories are created whenever a new architecture requires different PDL functions. CAT1 PDL and HAL Libraries are being used by the XMC7000 devices. For the specifics of each library it is best to refer to their respective documentation:

- [CAT1 PDL Reference Manual](#)
- [CAT1 HAL Reference Manual](#)

When searching for documentation on a peripheral, the HAL will use the high-level function name such as UART or ADC because the HAL is independent from the lower-level implementation. On the other hand, the PDL more closely represents the hardware implementation, which the names will reflect. Each of the device architectures may implement things differently, resulting in distinct PDL names from one device to the next.

For example, the HAL API documentation has sections for UART, I2C, and SPI. However, in XMC7000 devices, those functions are all implemented in a programmable serial communication block (SCB). For that reason, the PDL API documentation for all three functions will be found under the heading "SCB."

As a second example, the HAL has an API for analog to digital conversion (ADC). In XMC7000 devices, the ADC is implemented using a successive approximation register ADC. Consequently, the PDL API documentation will be found under the heading SAR2 ADC.

As a final example, the HAL has separate APIs for Timer/Counter and PWM. In the XMC7000 devices, those functions are implemented by the TCPWM hardware block and you will find the documentation all under the TCPWM heading.

Note: The HAL objects that the user provides to the HAL drivers contain the peripheral's base hardware address, which is what the PDL functions need to operate. Therefore, it is possible to call PDL functions on peripherals that were set up using the HAL by using the base address from the HAL object. However, if you choose this method, be careful that the PDL functions do not interfere with the HAL operation.

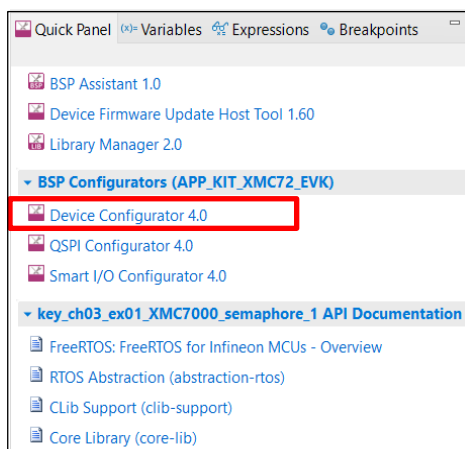
2.3 Peripherals

The PDL and HAL documentation are the best places to read about the APIs provided by these libraries. They include complete descriptions of the APIs, as well as a plethora of code snippets and use case examples. Rather than repeat all of that information in this section, we will only discuss the basic flow for setting up and using the kit peripherals and provide some basic examples. For specifics you should refer to the documentation.

Note: *A quick way to look up documentation for a particular function or structure within the Eclipse IDE for ModusToolbox™ is to highlight the element in question, right-click it, and then select **Open Declaration**. This will take you to where the element is declared in the library source code, where there will usually be a brief description of the element in a comment block.*

Note: *Almost all of Infineon's libraries include an html document that contains all of the information you need to effectively use that library. For these libraries, this document can be found in the following path: `mtb_shared/<library_name>/<version>/docs/reference_manual.html`. You can either go to that location and open the file with the web browser of your choice, or if you are using the Eclipse IDE for ModusToolbox™, there is a link to each file in the quick panel under the **Documentation** section.*

If you're using the PDL rather than the HAL in your application, it can be quite cumbersome to set up all the peripherals using the PDL API. Instead you should use the **Device Configurator**, which provides you with a GUI to configure all the peripherals in your device. The Device Configurator then automatically generates PDL code based on your selections. The code that the Device Configurator generates is run when you call the function `cybsp_init`.



2.3.1 GPIO

You will use this in [Exercise 1](#), [Exercise 2](#), [Exercise 3](#), [Exercise 4](#), [Exercise 5](#) and [Exercise 6](#):

2.3.1.1 Drive Mode

A drive mode is essentially a specific electrical configuration that a GPIO can take on. The XMC7000 GPIOs support seven primary drive modes:

- Strong – Used for digital outputs, able to pull the pin high or low. These are often used for LEDs.
- High Impedance (High-Z) – Used for digital input pins and analog pins.
- Resistive Pull Up – Able to drive the pin low, but only pulls the pin high through a resistor so that an external source can force the pin low. These are often used for active low buttons.
- Resistive Pull Down – Able to drive the high, but only pulls the pin low through a resistor so that an external source can force the pin high.
- Open Drain Drives Low – Able to drive the pin low, can be pulled high externally. These are often used for wired-or communication standards such as I2C.
- Open Drain Drives High – Able to drive the pin high, can be pulled low externally.
- Resistive Pull Up and Down – DC biases the pin, useful for some analog pins. Also allows external sources to force the pin to the opposite state.

In addition to these primary drive modes, an input buffer can also be enabled/disabled on each GPIO. In total there are fourteen drive modes a GPIO can take on, each of the seven primary drive modes with or without an input buffer.

More information about the supported GPIO drive modes can be found in the PDL documentation under **CAT1 Peripheral Driver Library > PDL API Reference > GPIO > Pin drive mode**

2.3.1.2 HAL

To initialize a GPIO using the HAL, call the function `cyhal_gpio_init`. While there are no `init_adv` or `init_cfg` functions for the GPIO, you can choose to use the Device Configurator for GPIO configuration instead of calling the standard `init` function.

Once initialized, input pins can be read using the function `cyhal_gpio_read` and outputs can be driven using the function `cyhal_gpio_write` or `cyhal_gpio_toggle`.

For example, the following snippet will initialize a pin as a strong drive output with the output driven high (1); in the main loop the output will toggle to the opposite state every 100ms.

```
cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT, CYHAL_GPIO_DRIVE_STRONG, 1);

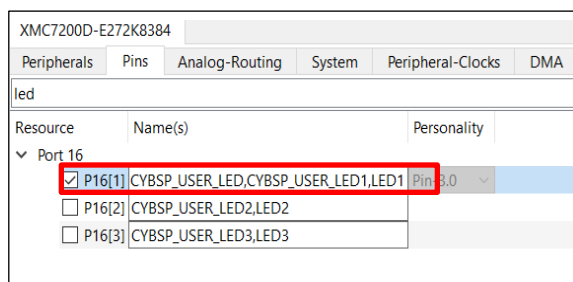
for(;;)
{
    cyhal_system_delay_ms(100);
    cyhal_gpio_toggle(CYBSP_USER_LED);
}
```

If you want to change what a GPIO is used for, for example if you were using it to read input from a button, but now you want to drive it with a PWM, it is important to properly reconfigure the pin to do so. In order to reconfigure the pin using the HAL, you must first call the `cyhal_gpio_free` function to un-initialize the pin and then call the initialization function to reinitialize the pin with your new configuration parameters.

The documentation for the HAL GPIO functions can be found under **Hardware Abstraction Layer > HAL API Reference > HAL Drivers > GPIO**.

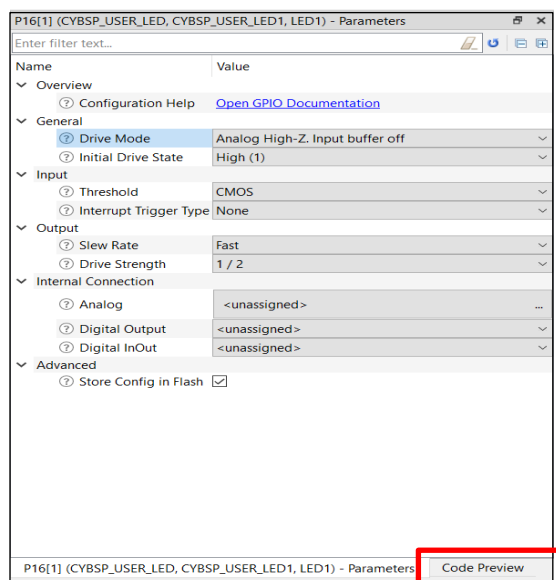
2.3.1.3 PDL

To initialize a GPIO using the PDL you should open the Device Configurator and go to the **Pins** tab. From here you can enable a pin simply by checking its box:



Many pins have pre-defined names. You can choose to enter your own application-specific name, or add to the list of existing names for a given pin. Names in the list are separated by commas. Do not use any spaces as they will be converted to underscores. Adding a sensible application-specific name can make the code easier to follow since that name can be used in the code.

Next, on the right side of the screen, you will be presented with a list of pin parameters to configure for the selected pin:



If you click on the **Code Preview** tab at the bottom of this window you will see the PDL code that the Device Configurator is generating for you.

Note: The Code Preview window may also be located below the parameters window instead of a separate tab. The organization and size of the windows can be adjusted by dragging the window banners to the desired location.

Configure the pin to your liking, then do **File > Save**, and exit the Device Configurator.

The final step is to read from or write to the pin. Any pins you set up in the Device Configurator will automatically be initialized when you call `cybsp_init`. The PDL provides several functions for reading from and writing to pins, some commonly used ones are:

- `Cy_GPIO_Read`
- `Cy_GPIO_Write`
- `Cy_GPIO_Set`
- `Cy_GPIO_Clr`
- `Cy_GPIO_Inv`

For example, the following function will toggle the state of an output pin:

```
Cy_GPIO_Inv(CYBSP_USER_LED_PORT, CYBSP_USER_LED_NUM);
```

If you want to change what a GPIO is used for, for instance say you were using it to read input from a button, but now you want to drive it with a PWM, it is important to properly reconfigure the pin to do so. If you are using the PDL, you should use the function `Cy_GPIO_Pin_Init` for this.

The documentation for the PDL GPIO functions can be found under **Peripheral Driver Library > PDL API Reference > GPIO > Functions**.

2.3.1.4 PDL vs. HAL

The HAL API has no way to directly configure the following GPIO parameters:

- AMux bus splitter
- Vtrip
- SlewRate

If you need to configure any of these parameters in a way that is different than the HAL provides by default, you will need to use the PDL.

2.3.1.5 Interrupt Events

GPIOs are able to trigger interrupts in the following scenarios:

- Rising Edge – When the pin goes from low to high
- Falling Edge – When the pin goes from high to low
- Rising/Falling Edge – When the pin goes from low to high or from high to low

2.3.2 PWM

You will use this in [Exercise 9](#): and [Exercise 10](#):

PWMs are implemented using a Timer, Counter, PWM hardware block called a TCPWM for short. For XMC7000 devices, each TCPWM connects to a specific set of GPIO pins, so it is important to consider which pins will be used for TCPWM functions to make sure the required resources are available. Some TCPWM blocks have 16-bit counters while others have 32-bit counters. The number and type of TCPWM blocks is device specific.

2.3.2.1 HAL

If you are using the HAL, you first need to call the PWM initialization function `cyhal_pwm_init`. Alternately, you can call `cyhal_pwm_init_adv` if you need advanced functionality or `cyhal_pwm_init_cfg` if you want to set up the PWM using the Device Configurator.

After initializing your PWM, you need to call the `cyhal_pwm_set_duty_cycle` function to specify the frequency and duty cycle of your PWM. You also have the option to call `cyhal_pwm_set_period` to specify the period and pulse width.

To start the PWM, call the function `cyhal_pwm_start`.

To stop the PWM, call the function `cyhal_pwm_stop`.

The following code snippet will setup and start a PWM with a frequency of 1 Hz and a duty cycle of 50%:

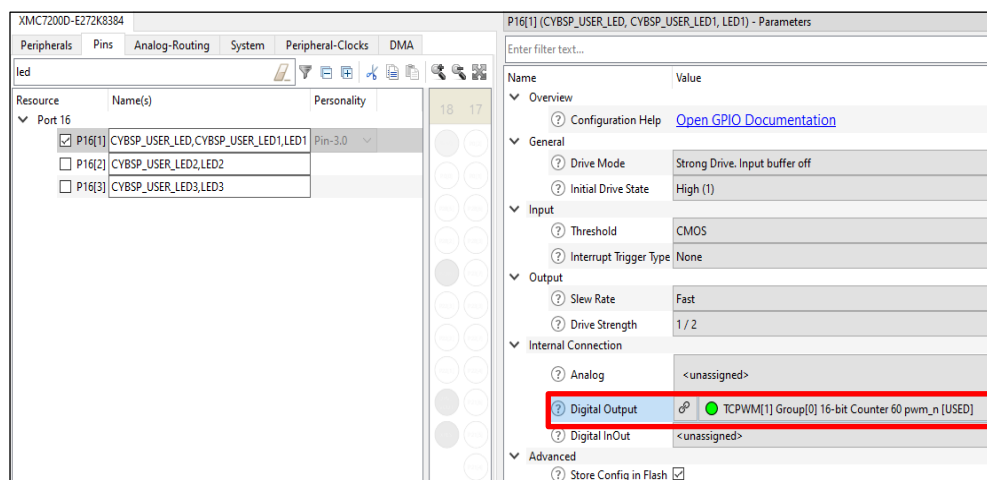
```
cyhal_pwm_t pwm_obj;  
cyhal_pwm_init(&pwm_obj, CYBSP_USER_LED, NULL);  
cyhal_pwm_start(&pwm_obj);  
cyhal_pwm_set_duty_cycle(&pwm_obj, 50.0, 1);
```

The documentation for the HAL PWM functions can be found under **Hardware Abstraction Layer > HAL API Reference > HAL Drivers > PWM**.


2.3.2.2 PDL

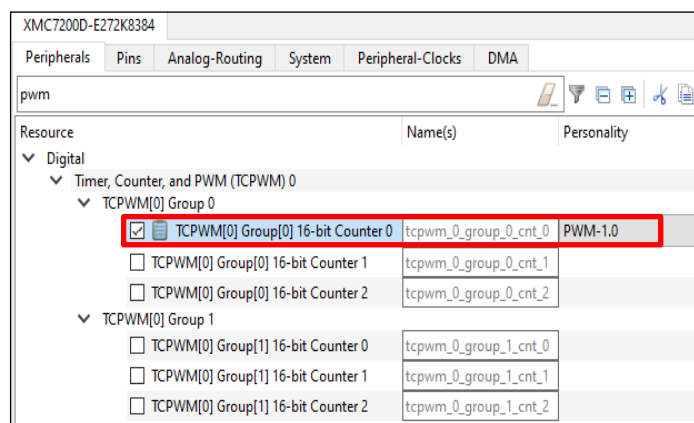
If you are using the PDL, first decide which pin you want the PWM to output on. Once chosen, you should open the Device Configurator and navigate to that pin. In the pin configuration settings, under **Digital Output**, choose the option ending in **pwm** or **pwm_n**. You may select either a 16-bit or 32-bit TCPWM based on your application's requirements. The drive mode should be set appropriately.

Note: If you don't select a valid drive mode for a PWM, you will see messages in the Notice List when you save the configuration.



*Note: The **pwm_n** output is just the compliment of the **pwm** output.*

Then click on the "chain" button  that appears in the **Digital Output** parameter. This will take you to **Peripherals** tab for the TCPWM you selected. Enable the counter you need under the respective group by checking its box (it will appear with a clip-board next to its name), select **PWM-<version>** from the popup and click **OK**.



While you can choose to use the default name, it is recommended to enter a more descriptive name for your application such as "PWM." Either way, make a note of the name as you will need to know it when you write the code. If you look in the Code Preview area, you will see the definitions that are created using the name that you choose:

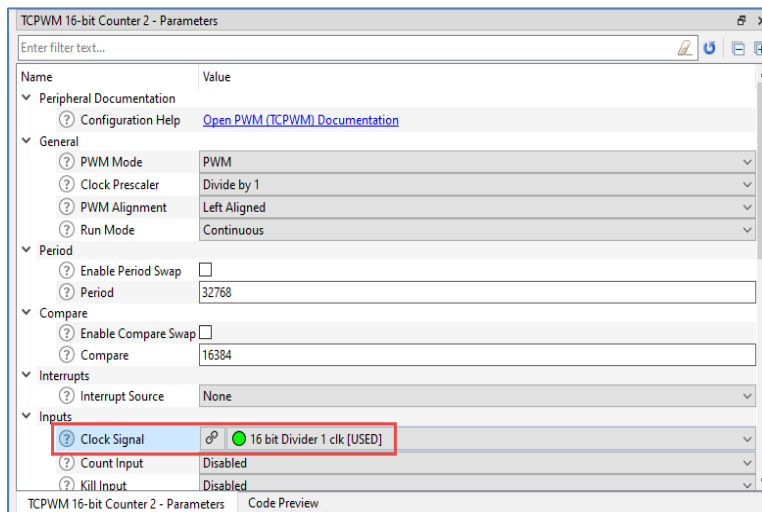
```
Code Preview
Enter search text...

#define PWM_HW TCPWM
#define PWM_NUM 2UL
#define PWM_MASK (1UL << 2)
#define PWM_INFUT_DISABLED 0x7U

const cy_stc_tcpwm_pwm_config_t PWM_config =
{
    .pwmMode = CY_TCPWM_PWM_MODE_PWM,
    .clockPrescaler = CY_TCPWM_PWM_PRESCALER_DIVBY_1,
    .pwmAlignment = CY_TCPWM_PWM_LEFT_ALIGN,
    .deadTimeClocks = 0,
    .runMode = CY_TCPWM_PWM_CONTINUOUS,
    .period0 = 99,
    .period1 = 32768,
    .enablePeriodSwap = false,
    .compare0 = 1,
    .compare1 = 16384,
    .enableCompareSwap = false,
    .interruptSources = CY_TCPWM_INTI_NONE,
    .invertPWMout = CY_TCPWM_PWM_INVERT_DISABLE,
    .invertPWMoutN = CY_TCPWM_PWM_INVERT_DISABLE,
    .killMode = CY_TCPWM_PWM_STOP_ON_KILL,
}

TCPWM 16-bit Counter 2 (PWM) - Parameters Code Preview
```

Then, on the right side of the screen, you can configure the PWM how you want. One parameter you need to change is **Clock Signal**:



Then do **File > Save**, and exit the Device Configurator.

In your application code you need to call the function `Cy_TCPWM_PWM_Init` to initialize your PWM. The Device Configurator generates macros that can be used for the first two arguments to this function. By default, these are called `<PWM_Name>_HW` and `<PWM_Name>_NUM`, where `<PWM_Name>` is the name of your PWM from earlier. The third argument to this function is a pointer to the configuration structure that the Device Configurator generated. By default, this structure is called `<PWM_Name>_config`.

Then you need to call the function `Cy_TCPWM_PWM_Enable` to enable your PWM.

To start the PWM, call the function `Cy_TCPWM_TriggerStart_Single`.

To stop the PWM, call the `Cy_TCPWM_TriggerStopOrKill_Single` function.

The following example will initialize and start a PWM that was setup using the Device Configurator and was named "PWM."

```
Cy_TCPWM_PWM_Init(PWM_HW, PWM_NUM, &PWM_config);
Cy_TCPWM_PWM_Enable(PWM_HW, PWM_NUM);
Cy_TCPWM_TriggerStart_Single(PWM_HW, PWM_NUM);
```

Many other functions exist to change the period, compare, etc. You can find complete documentation for the XMC7000 PDL PWM functions under **Peripheral Driver Library > PDL API Reference > TCPWM > PWM > Functions**.

2.3.2.3 PDL vs. HAL

The HAL API has no way to directly configure the following PWM parameters:

- Compare1
- Period1
- Enable Compare swap or Period swap

The HAL only allows you to set one period and one duty cycle (the compare value is calculated for you from the duty cycle). There is no way to set a second period or duty cycle, you will have to use the PDL instead.

When using the HAL, you are required to connect the output of your PWM to a GPIO. The PDL allows for more flexibility, the PWM output can be connected to a GPIO or directly to other hardware blocks.

2.3.2.4 Interrupt Events

PWMs are able to trigger interrupts in the following scenarios:

- Terminal Count – When the counter rolls from its max value (period) back to 0
- Compare – When the counter matches the compare value
- Both – Both Terminal Count and Compare

2.3.3 ADC

You will use this in [Exercise 11](#): and [Exercise 12](#):

This section describes the SAR ADC that is available on XMC7000 devices.

2.3.3.1 HAL

To initialize an ADC, you must initialize an ADC block and an ADC channel. If you are using the HAL, you can call the function `cyhal_adc_init` to initialize an ADC block. If you want to use a different ADC configuration than the default, you can call the function `cyhal_adc_configure`.

You can then initialize and configure an ADC channel by calling the function `cyhal_adc_channel_init_diff`. If you want to change the channel configuration at run time, you can call the function `cyhal_adc_channel_configure`. Alternately, you can use `cyhal_adc_init_cfg` if you want to use the Device Configurator to set up the ADC and its channels. To read from the ADC, simply call the function `cyhal_adc_read`.

The following snippet will initialize an ADC with one single ended channel and will read the value once every second.

```
/* ADC block and channel objects */
cyhal_adc_t adc_obj;
cyhal_adc_channel_t adc_chan_0_obj;

/* ADC conversion result */
int adc_out;

/* Initialize ADC */
cyhal_adc_init(&adc_obj, P10_6, NULL);

/* ADC configuration structure */
const cyhal_adc_config_t ADCconfig = {
    .continuous_scanning = false,
    .resolution = 12,
    .average_count = 1,
    .average_mode_flags = 0,
    .ext_vref_mv = 0,
    .vneg = CYHAL_ADC_VNEG_VREF,
    .vref = CYHAL_ADC_REF_VDDA,
    .ext_vref = NC,
    .is_bypassed = false,
    .bypass_pin = NC
};

/* Configure the ADC */
cyhal_adc_configure(&adc_obj, &ADCconfig);

/* ADC channel configuration structure */
const cyhal_adc_channel_config_t channel_config = {
    .enable_averaging = false,
    .min_acquisition_ns = 220,
    .enabled = true };

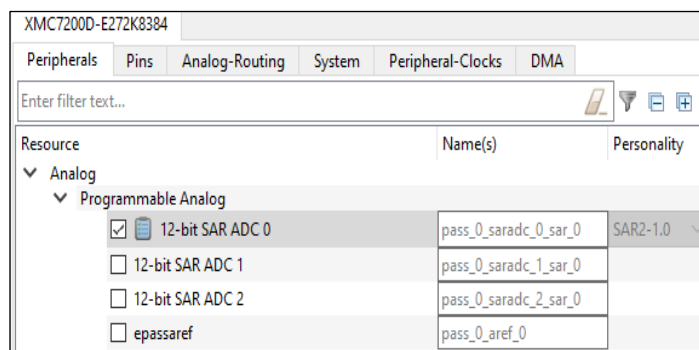
/* Initialize ADC channel 0 */
cyhal_adc_channel_init_diff(&adc_chan_0_obj, &adc_obj, P10_6, CYHAL_ADC_VNEG,
&channel_config);

/* Read the ADC conversion result for corresponding ADC channel. */
adc_out = cyhal_adc_read_uv(&adc_chan_0_obj);
```

The documentation for the HAL ADC functions can be found under **Hardware Abstraction Layer > HAL API Reference > HAL Drivers > SAR**.

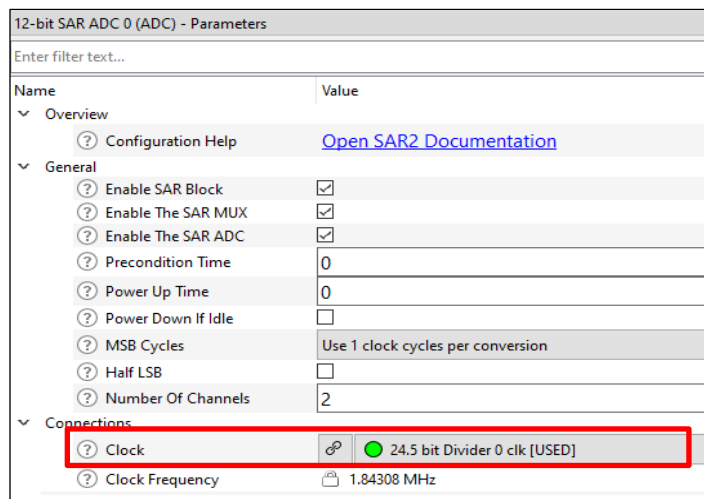
2.3.3.2 PDL

To initialize an ADC using the PDL, open the Device Configurator and select the **Peripherals** tab. Then, in the dropdown menu, select **Analog > Programmable Analog**. Then check the box on the **12-bit SAR ADC** to enable it:



Give the ADC a practical name like "ADC" and make a note of it for the code.

Then, on the right side of the screen you can configure the ADC how you want. One parameter you need to change is **Clock**:



You also need to select how many channels you want and specify which pin(s) each channel connects to. The pins themselves should be configured with the drive mode **Analog High-Z, Input buffer off**.

Note: If you don't select a valid drive mode for an ADC, you will see messages in the Notice List when you save the configuration.

If you look in the Code Preview area, you will see the definitions that are created using the name that you chose:

```
Code Preview
Enter search text...

/* NOTE: This is a preview only. It combines elements of the
 * cycfg_peripherals.c and cycfg_peripherals.h files located in
 * C:/WORK/mtw-XMC7200_ch02/12/ch02_PDL_adcread/bsps/TARGET_APP_
 */

#include "cy_sar2.h"
#include "cy_sysclk.h"
#include "cycfg_routing.h"

#define ADC_HW PASS0_SAR0
#define ADC_CH0_HW PASS0_SAR0_CH0
#define ADC_CH1_HW PASS0_SAR0_CH1
#define ADC_CH0_IRQ pass_0_interrupts_sar_0_IRQn
#define ADC_CH1_IRQ pass_0_interrupts_sar_1_IRQn
#define ADC_VDDA_RANGE CY_SAR2_VDDA_2_7V_TO_4_5V
#define SARMUX0_CH0_PORT_ADDR
    #define SARMUX0_CH0_PORT_ADDR 0
#endif
#define SARMUX0_CH1_PORT_ADDR
    #define SARMUX0_CH1_PORT_ADDR 0
#endif

const cy_stc_sar2_channel_config_t ADC_channel_0_config =
{
    .channelHwEnable = true,
    .triggerSelection = CY_SAR2_TRIGGER_OFF,
    .channelPriority = 0U,
    .preemptionType = CY_SAR2_PREEMPTION_FINISH_RESUME,
    .doneLevel = CY_SAR2_DONE_LEVEL_PULSE,
}
```

Next, do **File > Save**, and exit the Device Configurator.

In the example shown above, the reference is Vdda by default. If you instead chose the internal reference, you will need to enable that reference in the configurator and start it in the code using `Cy_SysAnalog_Init` and `Cy_SysAnalogEnable`.

In your application code you need to call the function `Cy_SAR2_Init` to initialize your ADC. The Device Configurator generated a macro for the first argument to this function, by default this is called `<ADC_Name>_HW`, where `<ADC_Name>` is the name of your ADC from earlier. The second argument to this function is a pointer to the configuration structure that the Device Configurator generated. By default, this structure is called `<ADC_Name>_config`.

Then you need to call the function `Cy_SAR2_Enable` to enable your ADC.

You can call the function `Cy_SAR2_Channel_SoftwareTrigger` to trigger a software interrupt to start conversion. The function `Cy_SAR2_Channel_GetInterruptStatus` is used to check whether the conversion is done. Then the function `Cy_SAR2_Channel_getResult` can be used to obtain the results it counts. You can also configure your ADC to produce an interrupt when a conversion has finished

The following snippet will initialize and start an ADC that was configured in the Device Configurator as shown above with the name ADC and using Vdda as the reference. It will then start a conversion and will wait until the result is ready. The below code also contains calculations for counts to voltage for 3.3v and 5v internal reference voltages.

```
int32_t ADCresult = 0; /* ADC conversion result in counts */
int32_t microVolts = 0; /* ADC conversion result in microVolts */

/* Initialize and enable the ADC */
Cy_SAR2_Init(ADC_HW, &ADC_config);
Cy_SAR2_Enable(ADC_HW);
/* Initiate a software trigger to start conversion*/
Cy_SAR2_Channel_SoftwareTrigger(ADC_HW, 0);

/* Wait for conversion to complete*/
while (CY_SAR2_INT_GRP_DONE != Cy_SAR2_Channel_GetInterruptStatus(ADC_HW, 0));
```



```
/* Get conversion results in counts, do not obtain or analyze status here */
ADCresult = Cy_SAR2_Channel_GetResult(ADC_HW, 0, NULL);

/* Clear interrupt source */
Cy_SAR2_Channel_ClearInterrupt(ADC_HW, 0, CY_SAR2_INT_GRP_DONE);

/*counts to voltage calculation for 3.3v internal reference*/
microVolts = (ADCresult*3.3*1000000)/4095;
/*counts to voltage calculation for 5v internal reference*/
microVolts = (ADCresult*5*1000000)/4095;
```

Waiting for the conversion result is very inefficient. Typically, an interrupt would be used to read the result so that the CPU could complete other useful work while the conversion is taking place.

The documentation for the PDL ADC functions can be found under **Peripheral Driver Library > PDL API Reference > SAR2 ADC > Functions**.

Note: If you need to configure any of these parameters in a way that is different than the default HAL provides, you will need to use the PDL. When using the HAL, the input of your ADC is required to be a GPIO. When using the PDL this is not a requirement as the ADC input can be connected to a GPIO or directly to other hardware blocks.

2.3.3.3 PDL vs. HAL

The HAL API has no way to directly configure the following ADC parameters:

- Injection Channel
- Start of Conversion Input Trigger
- Differential Result Format
- Single Ended Result Format
- Range Interrupt
- Saturation Interrupt

If you need to configure any of these parameters in a way that is different than the HAL provides by default, you will need to use the PDL.

When using the HAL, the input of your ADC is required to be a GPIO. When using the PDL this is not a requirement, the ADC input can be connected to a GPIO or directly to other hardware blocks.

2.3.3.4 Interrupt Events

ADCs are able to trigger interrupts in the following scenarios:

- Overflow – When an overflow occurs
- FW Collision – When a firmware collision occurs
- End of Scan – When a scan of all channels has completed (for continuous scanning only)
- Async Read Complete – When an asynchronous read operation has completed
- Range – When the value measured by a channel is not within a specified range
- Saturation – When a channel becomes saturated

2.3.4 UART

You will use this in [Exercise 3](#) , [Exercise 4](#) , [Exercise 13](#) , [Exercise 14](#) , [Exercise 15](#) and [Exercise 16](#) .

UARTs are implemented using a Serial Communication Block called an SCB for short. Each SCB can be configured to implement UART, SPI, I²C or E²I²C. In XMC7000 devices, each SCB connects to a specific set of GPIO pins, so it is important to consider which pins will be used for SCB functions and make sure the required resources are available.

2.3.4.1 Printing with *retarget-io*

Printing messages to a serial terminal emulator window on a computer is so common (e.g. for printing debug messages) that a library called *retarget-io* is provided in ModusToolbox™ to simplify the process. It allows you to use standard C functions such as `printf` and redirects them to the UART so that they can be displayed on a serial terminal window.

To use the *retarget-io* library to print messages, the steps are:

1. Use the library manager to add the *retarget-io* library in the Peripherals category.
2. Include the header file `cy_retarget_io.h` in `main.c`.
3. In the initialization section of the firmware, call the following function to initialize the interface using the debug UART pins (these are the pins that connect to the KitProg3) with the default baud rate of 115200.

```
cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,  
CY_RETARGET_IO_BAUDRATE);
```

4. Use `printf` in your code as normal. For example, to print a variable "myVar" you could use:

```
printf("The value of myVar is: %d\n", myVar);
```

The *retarget-io* library also has the ability to automatically convert new line characters (`\n`) into a new line plus carriage return (`\n\r`) by adding `CY_RETARGET_IO_CONVERT_LF_TO_CRLF` to the `DEFINES` in the application's *Makefile*. This feature is helpful if the serial terminal emulator you are using doesn't support that option and you don't want to use `\n\r` in all of the `printf` statements. This can be easily done in the application's *Makefile*:

```
DEFINES= CY_RETARGET_IO_CONVERT_LF_TO_CRLF
```

The UART object that is created by the *retarget-io* library is externally accessible so you can even use standard HAL UART functions to do other things with it. For example, you can use HAL UART functions to enable and configure an RX channel if you want to receive data from the UART while still using `printf` to send messages. The UART object can be found in the `cy_retarget_io.h` file:

```
extern cyhal_uart_t cy_retarget_io_uart_obj;
```

As with most Infineon libraries, the documentation for the *retarget-io* library can be found in the *api_reference_manual.html* file in the library's *docs* directory. This file can also be accessed from the Quick Panel in the Eclipse IDE for ModusToolbox™.

2.3.4.2 HAL

If you need UART functionality beyond what *retarget-io* provides, you can use the HAL UART API.

First, the function `cyhal_uart_init` is used to initialize a UART (assuming you don't use *retarget-io* to initialize it). Alternately, you can use `cyhal_uart_init_cfg` if you want to use the Device Configurator to set up the UART.

There are functions to get/put single characters, read/write a buffer of data, and even asynchronous read/write functions to allow background transfer/receive operations. See the API documentation for further details and usage examples.

The following snippet shows how you can receive characters from the UART using the HAL.

```
/* Variable to hold read value */
uint8_t read_data;

/* UART object and configuration structure */
cyhal_uart_t uart_obj;
const cyhal_uart_cfg_t uart_config =
{
    .data_bits = 8,
    .stop_bits = 1,
    .parity = CYHAL_UART_PARITY_NONE,
    .rx_buffer = NULL, /* Software FIFO not used since we */
                      /* will read characters as they arrive */
    .rx_buffer_size = 0
};

/* Initialize UART */
cyhal_uart_init(&uart_obj, CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
               NC, NC, NULL, &uart_config);

/* Read one character */
cyhal_uart_getc(&uart_obj, &read_data, 0);

/* Add code here to operate on the value of read_data */
```

Note: *The function `cyhal_uart_getc` is blocking, meaning it will wait until a character is received. In a real application, it would be more common to setup an interrupt that is called whenever a new character is received so that the CPU could perform other tasks instead of waiting.*

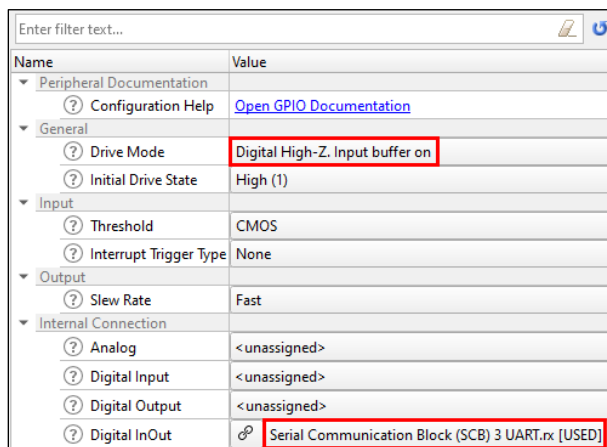
The documentation for the HAL UART functions can be found under **Hardware Abstraction Layer > HAL API Reference > HAL Drivers > ADC**.

2.3.4.3 PDL

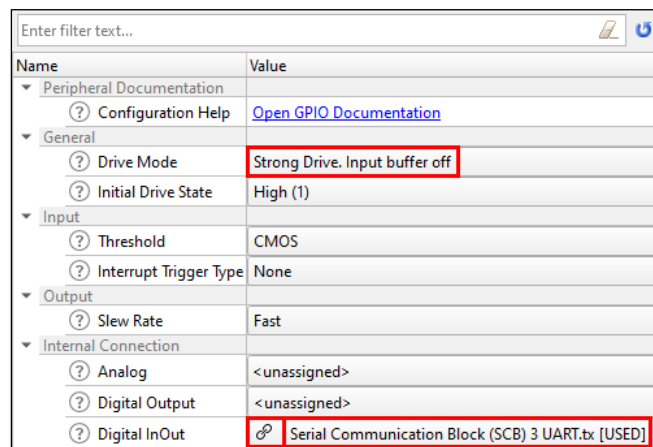
To initialize a UART using the PDL, you first need to figure out what pins you want the UART to use. Once you've done that, you should open the Device Configurator and setup the pins.

In the case of the KIT_XMC72_EVK, the debug UART TX is connected to pin 13.1 and the RX is connected to pin 13.0.


Open the Device Configurator and navigate to the RX pin. Enable the pin by checking its box and set the **Digital InOut** parameter to the option ending in **UART.rx**. The drive mode for RX should be set to **Digital High-Z Input buffer on** as shown below. Set the same for the TX pin except choose the option ending in **UART.tx**. The drive mode for TX should be **Strong Drive, Input buffer off**. Make sure both pins are using the same SCB.

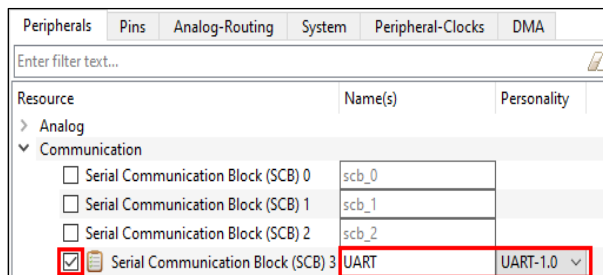


Name	Value
Peripheral Documentation	
Configuration Help	Open GPIO Documentation
General	
Drive Mode	Digital High-Z. Input buffer on
Initial Drive State	High (1)
Input	
Threshold	CMOS
Interrupt Trigger Type	None
Output	
Slew Rate	Fast
Internal Connection	
Analog	<unassigned>
Digital Input	<unassigned>
Digital Output	<unassigned>
Digital InOut	Serial Communication Block (SCB) 3 UART.rx [USED]



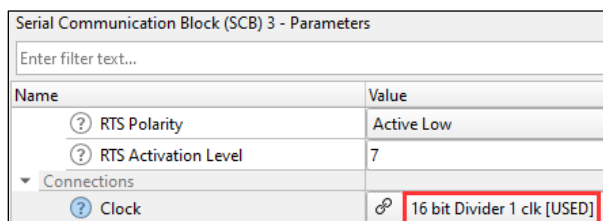
Name	Value
Peripheral Documentation	
Configuration Help	Open GPIO Documentation
General	
Drive Mode	Strong Drive. Input buffer off
Initial Drive State	High (1)
Input	
Threshold	CMOS
Interrupt Trigger Type	None
Output	
Slew Rate	Fast
Internal Connection	
Analog	<unassigned>
Digital Output	<unassigned>
Digital InOut	Serial Communication Block (SCB) 3 UART.tx [USED]

Click the "chain" button  next to the **Digital InOut** connection on either pin. This will take you to **Peripherals** tab for the SCB you selected. Enable the SCB by checking its box and select **UART-<version>** from the popup menu. Then give it a name:



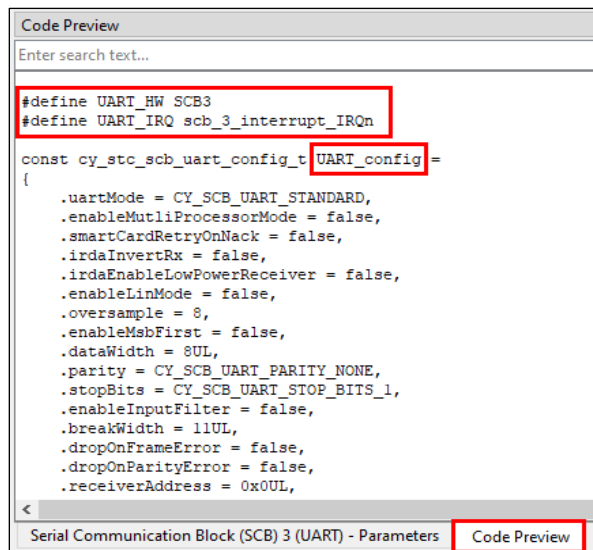
Resource	Name(s)	Personality
Communication		
<input type="checkbox"/> Serial Communication Block (SCB) 0	scb_0	
<input type="checkbox"/> Serial Communication Block (SCB) 1	scb_1	
<input type="checkbox"/> Serial Communication Block (SCB) 2	scb_2	
<input checked="" type="checkbox"/> Serial Communication Block (SCB) 3	UART	UART-1.0

On the right side of the screen you can configure the UART how you want. One parameter you need to change is **Clock**:



Name	Value
RTS Polarity	Active Low
RTS Activation Level	7
Connections	
Clock	16 bit Divider 1 clk [USED]

If you look in the Code Preview area, you will see the definitions that are created using the name that you chose:



```

Code Preview
Enter search text...

#define UART_HW SCB3
#define UART_IRQ scb_3_interrupt_IRQn

const cy_stc_scb_uart_config_t UART_config =
{
    .uartMode = CY_SCB_UART_STANDARD,
    .enableMutliProcessorMode = false,
    .smartCardRetryOnNack = false,
    .irdaInvertRx = false,
    .irdaEnableLowPowerReceiver = false,
    .enableLinMode = false,
    .oversample = 8,
    .enableMsbFirst = false,
    .dataWidth = 8UL,
    .parity = CY_SCB_UART_PARITY_NONE,
    .stopBits = CY_SCB_UART_STOP_BITS_1,
    .enableInputFilter = false,
    .breakWidth = 11UL,
    .dropOnFrameError = false,
    .dropOnParityError = false,
    .receiverAddress = 0x0UL,
}

```

Serial Communication Block (SCB) 3 (UART) - Parameters Code Preview

Then do **File > Save**, and exit the Device Configurator.

In your application code you need to call the function `Cy_SCB_UART_Init` to initialize the UART. The Device Configurator generates a macro for the first argument to this function, by default this is called `<SCB_Name>_HW`, where `<SCB_Name>` is the name you gave your SCB in the Configurator. The second argument to this function is a pointer to the configuration structure that the Device Configurator generated. By default, this structure is called `<SCB_Name>_config`. The final argument is a context pointer that you must provide for the function to fill in.

Then you need to call the function `Cy_SCB_UART_Enable` to enable your UART.

There are several UART functions to send and receive data that you can read about in the documentation. The send and receive functions are broken into high-level operations such as `Cy_SCB_UART_Receive`, and low-level operations such as `Cy_SCB_UART_Get`. One low-level function that's useful for debugging is `Cy_SCB_UART_PutString`.

Note: High-level APIs are fairly abstracted, meaning that they are more generic and therefore limited in functionality. Low-level APIs are much more detailed and specific due to a low level of abstraction. Low-level APIs allow for finer control over application functi

The following snippet shows how to read a single character from the UART that was configured using the Device Configurator settings as shown above

```

char charReceived;

/* UART context variable */
cy_stc_scb_uart_context_t UART_context;

/* Configure and enable the UART */
Cy_SCB_UART_Init(UART_HW, &UART_config, &UART_context);
Cy_SCB_UART_Enable(UART_HW);

```

```
/* Wait for data in the Rx FIFO */
while ((Cy_SCB_UART_GetRxFifoStatus(UART_HW) & CY_SCB_UART_RX_NOT_EMPTY)
      != CY_SCB_UART_RX_NOT_EMPTY)
{
    /* Do nothing until there is data in the Rx FIFO */
}

/* Read one character */
charReceived = Cy_SCB_UART_Get(UART_HW);

/* Add code here to operate on the value of charReceived */
```

Note: The `Cy_SCB_UART_Get` function is non-blocking, so in this case we need to wait until the receive FIFO is not empty to read a character.

The documentation for the PDL UART functions can be found under **Peripheral Driver Library > PDL API Reference > SCB > UART**.

2.3.4.4 PDL vs. HAL

The HAL API has no way to directly configure the following UART parameters:

- Com Mode
- Oversample
- Bit Order
- Digital Filter
- TX-Enable (RS-485 Support)
- Flow Control
- Multi-Processor Mode
- Drop on Frame Error
- Drop on Parity Error
- Break Signal Bits

If you need to configure any of these parameters in a way that is different than the HAL default, you will need to use the PDL.

2.3.4.5 Interrupt Events

UARTs are able to trigger interrupts in the following scenarios:

- RX FIFO not Empty – When the HW RX FIFO buffer is not empty
- RX FIFO Full – When the HW RX FIFO buffer is full
- RX FIFO Overflow – When an attempt to write to a full HW RX FIFO buffer occurs
- RX FIFO Underflow – When an attempt to read from an empty HW RX FIFO buffer occurs
- RX Frame Error – When an RX frame error is detected
- Break Detected – When a break is detected
- RX FIFO Above Level – When the number of data elements in the HW RX FIFO is above the specified level

-
- UART Done – When a UART transfer is complete
 - TX FIFO Empty – When the HW TX FIFO buffer is empty
 - TX FIFO Not Full – When the HW TX FIFO buffer is not full
 - TX FIFO Overflow – When an attempt to write to a full HW TX FIFO buffer occurs
 - TX FIFO Underflow – When an attempt to read from an empty HW TX FIFO buffer occurs
 - TX FIFO Below Level – When the number of data elements in the HW TX FIFO buffer is below the specified level

2.3.5 I²C

You will use this in [Exercise 18](#):

I²C uses the same Serial Communication Hardware block as the UART. Again, in XMC7000 devices, each SCB connects to a specific set of GPIO pins, so it is important to consider which pins will be used for SCB functions to make sure the required resources are available. The I²C block supports both master and slave operations.

I²C is commonly used to read data from sensors. In this case, KIT_XMC72_EVK will be an I²C master while the sensor will be an I²C slave.

2.3.5.1 HAL

If you are using the HAL, the function to initialize an SCB block for I2C is `cyhal_i2c_init`. If you want to use a different I²C configuration than the default, you can call the function `cyhal_i2c_configure`. Alternately, you can use `cyhal_i2c_init_cfg` if you want to use the Device Configurator to set up the I2C block.

The following are the ways for a master to read/write data from/to the slave:

- There is a dedicated read function called `cyhal_i2c_master_read` and a dedicated write function called `cyhal_i2c_master_write`.
- There is a dedicated read function `cyhal_i2c_master_mem_read` and a dedicated write function `cyhal_i2c_master_mem_write` that performs I2C reads and writes using a block of data at a specified memory address.

There is also a function called `cyhal_i2c_master_transfer_async` which can do a read, a write, or both. On the slave side, the functions to configure the slave's read and write buffers are:

`cyhal_i2c_slave_config_read_buffer` and `cyhal_i2c_slave_config_write_buffer`.


The documentation for the HAL I²C functions can be found under **Hardware Abstraction Layer > HAL API Reference > HAL Drivers > I2C**. It includes several usage examples along with the full API description.

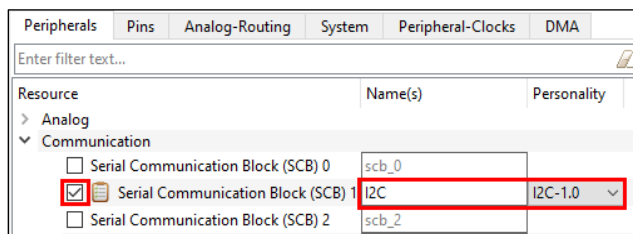
2.3.5.2 PDL

To initialize I²C using the PDL, the first thing you need to do is figure out what pins you want to use. Once you've got that, you should open the Device Configurator and select the pins for SCL and SDA from the **Digital InOut** pin setting. Be sure to choose the same SCB for both pins. The **Drive Mode** should be configured as **Open Drain Drives Low. Input buffer on** or **Resistive Pull-Up. Input buffer on** depends on whether external pull-up resistors are present on the board.

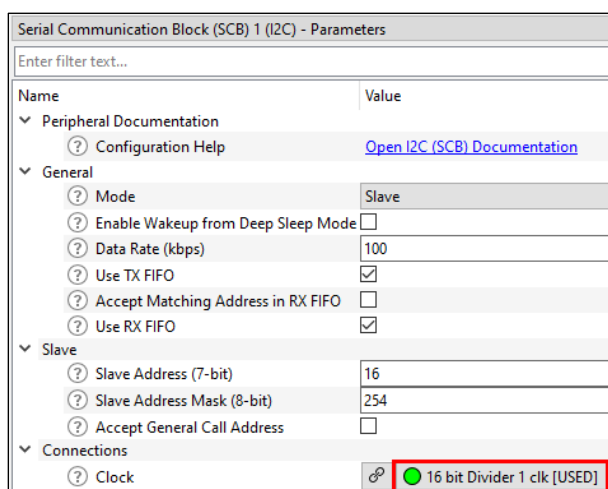
Enter filter text...	
Name	Value
Peripheral Documentation	
Configuration Help	Open GPIO Documentation
General	
Drive Mode	Open Drain, Drives Low. Input buffer on
Initial Drive State	High (1)
Input	
Threshold	CMOS
Interrupt Trigger Type	None
Output	
Slew Rate	Fast
Internal Connection	
Analog	<unassigned>
Digital Input	<unassigned>
Digital Output	<unassigned>
Digital InOut	Serial Communication Block (SCB) 1 I2C.scl [USED]

Enter filter text...	
Name	Value
Peripheral Documentation	
Configuration Help	Open GPIO Documentation
General	
Drive Mode	Open Drain, Drives Low. Input buffer on
Initial Drive State	High (1)
Input	
Threshold	CMOS
Interrupt Trigger Type	None
Output	
Slew Rate	Fast
Internal Connection	
Analog	<unassigned>
Digital Input	<unassigned>
Digital Output	<unassigned>
Digital InOut	Serial Communication Block (SCB) 1 I2C.sda [USED]

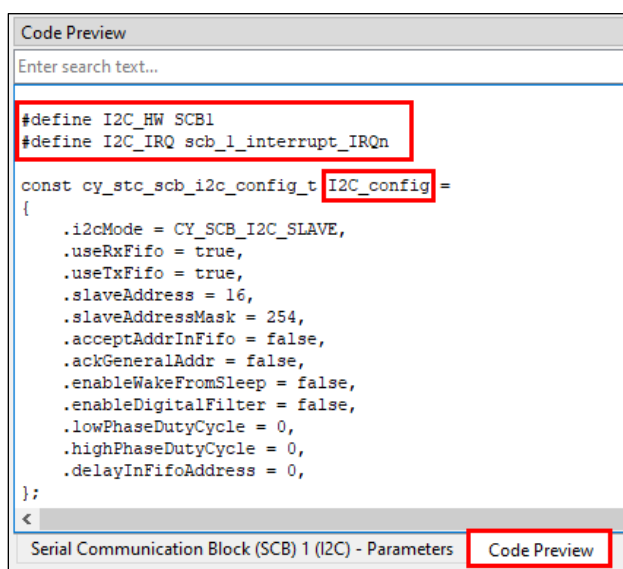
Once the pins are configured, click the "chain" button  next to the **Digital InOut** connection on one of the pins. This will take you to **Peripherals** tab for the SCB you selected. Enable the SCB by checking its box, selecting **I2C-<version>** from the popup menu and giving it a name:



On the right side of the screen you can configure your I²C how you want. One parameter you need to change is **Clock**:



If you look in the Code Preview area, you will see the definitions that are created using the name that you choose:



Once you've configured your I²C how you want it, do **File > Save**, and exit the Device Configurator.

In your application code you need to call the function `Cy_SCB_I2C_Init` to initialize your I2C. The Device Configurator generated a macro for the first argument to this function, by default this is called `<SCB_Name>_HW`, where `<SCB_Name>` is the name of your SCB from earlier. The second argument to this function is a pointer to the configuration structure that the Device Configurator generated. By default, this structure is called `<SCB_Name>_config`.

Then you need to call the function `Cy_SCB_I2C_Enable` to enable your I²C.

There are several I²C functions to send and receive data that you can read about in the documentation.

The documentation for the PDL I²C functions can be found under **Peripheral Driver Library > PDL API Reference > SCB > I2C**. It includes several usage examples along with the full API description.

2.3.5.3 PDL vs. HAL

The HAL API has no way to directly configure the following I²C parameters:

- Digital Filter
- SCL Low Phase
- SCL High Phase

If you have to configure any of these parameters in a way that is different than the HAL provides by default, you will need to use the PDL.

2.3.5.4 Interrupt Events

I2Cs are able to trigger interrupts in the following scenarios:

- Slave Read – When the slave was addressed and the master wants to read data
- Slave Write – When the slave was addressed and the master wants to write data
- Slave Read in FIFO – When all slave data from the SW read buffer has been loaded in to the HW TX FIFO buffer
- Slave Read Buffer Empty – When the master has read all data out of the read buffer
- Slave Read Complete – When the master completes reading from the slave
- Slave Write Complete – When the master completes writing to the slave
- Slave Error – When a slave I²C error is detected
- Master Write in FIFO – When all master write data from the SW write buffer has been loaded into the HW TX FIFO buffer (asynchronous transfers only)
- Master Write Complete – When the master completes writing to the slave
- Master Read Complete – When the master completes reading from the slave
- Master Error – When a master I²C error is detected

2.3.6 SPI

SPI uses the same Serial Communication Hardware block as the UART and I2C. Again, in XMC7000 devices, each SCB connects to a specific set of GPIO pins, so it is important to consider which pins will be used for SCB functions to make sure the required resources are available.

There are several Motorola modes available:

CYHAL_SPI_MODE_00_MSB	Standard motorola SPI CPOL=0, CPHA=0 with MSB first operation.
CYHAL_SPI_MODE_00_LSB	Standard motorola SPI CPOL=0, CPHA=0 with LSB first operation.
CYHAL_SPI_MODE_01_MSB	Standard motorola SPI CPOL=0, CPHA=1 with MSB first operation.
CYHAL_SPI_MODE_01_LSB	Standard motorola SPI CPOL=0, CPHA=1 with LSB first operation.
CYHAL_SPI_MODE_10_MSB	Standard motorola SPI CPOL=1, CPHA=0 with MSB first operation.
CYHAL_SPI_MODE_10_LSB	Standard motorola SPI CPOL=1, CPHA=0 with LSB first operation.
CYHAL_SPI_MODE_11_MSB	Standard motorola SPI CPOL=1, CPHA=1 with MSB first operation.
CYHAL_SPI_MODE_11_LSB	Standard motorola SPI CPOL=1, CPHA=1 with LSB first operation.

You can refer the [XMC7000 TRM documentation](#) to learn more about the operating modes of SPI protocol.

2.3.6.1 HAL

If you are using HAL you need to initialize the SPI master or slave interface using `cyhal_spi_init()` and provide the SPI pins (mosi, miso, sclk, ssel), number of bits per frame (data_bits) and SPI Motorola mode. The data rate can be set using `cyhal_spi_set_frequency()`. Use the function `cyhal_spi_send()` to synchronously send a byte of data and `cyhal_spi_recv()` to synchronously receive a byte of data from the read buffer. You can also use `cyhal_spi_transfer()` to synchronously write a block out and receive a value. This function will block for the duration of the data transfer. Also `cyhal_spi_transfer_async()` can be used for non-blocking asynchronous transfers.

The following code snippet initializes an SPI Master interface and the data rate of transfer is set using `cyhal_spi_set_frequency()`. The code snippet shows how to transfer a single byte of data from SPI master.

```
cy_rslt_t rslt;
cyhal_spi_t mSPI;
uint32_t spi_master_frequency = 1000000;
uint32_t transmit_data = 0x01;
uint32_t receive_data;

// Configuring the SPI master: Specify the SPI interface pins, frame size, SPI
// Motorola mode and master mode
rslt = cyhal_spi_init(&mSPI, CYBSP_SPI_MOSI, CYBSP_SPI_MISO, CYBSP_SPI_CLK,
CYBSP_SPI_CS, NULL, 8, CYHAL_SPI_MODE_00_MSB, false);
// Set the data rate to 1 Mbps
rslt = cyhal_spi_set_frequency(&mSPI, spi_master_frequency);
// Transfer one byte of data to the slave.
if (CY_RSLT_SUCCESS == cyhal_spi_send(&mSPI, transmit_data))
{
    // Receives one byte of data from the slave by transmitting a dummy byte 0xFF.
    if (CY_RSLT_SUCCESS == cyhal_spi_recv(&mSPI, &receive_data))
    {
        // Transfer Complete and successful
    }
}
```

}

Note: Remember to use the correct **MOSI**, **MISO**, **SCLK** and **SSEL** pins for master and slave configuration. You can find the pins in each SCB block under connections by configuring it as SPI in the Device Configurator.

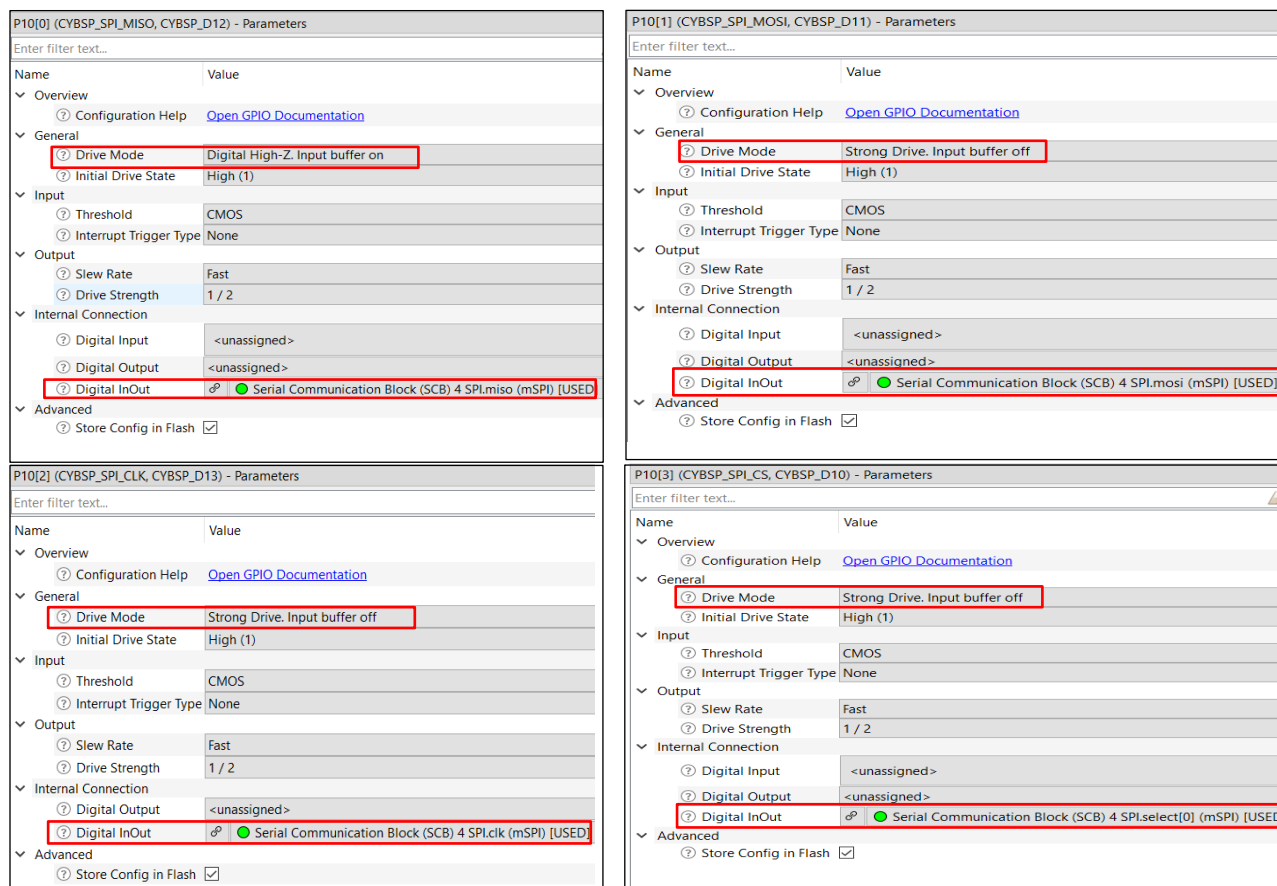
Follow the above code snippet to use SPI slave for single byte data transfer by making the `is_slave` parameter `true` in the `cyhal_spi_init()` function as shown below.

```
cyhal_spi_init(&sSPI, CYBSP_SPI_MOSI, CYBSP_SPI_MISO, CYBSP_SPI_CLK, CYBSP_SPI_CS, NULL,
8, CYHAL_SPI_MODE_00_MSB, true);
```

The documentation for the HAL SPI functions can be found under > **Hardware Abstraction Layer** > **HAL API Reference** > **HAL Drivers** > **SPI**.


2.3.6.2 PDL

To initialize SPI using the PDL, the first thing you need to do is figure out what pins you want to use. Once you've decided that, you should open the Device Configurator and select the pins for **miso**, **mosi**, **clk** and **cs** from the **Digital InOut** pin setting. Make sure to choose the same SCB for all 4 pins. The **Drive Mode** should be configured as **Digital High-z. Input buffer on** for **MISO** and **Strong Drive. Input buffer off** for **MOSI**, **CLK** and **CS** pins.



The following table summarizes the configurations shown in the screenshots:

Pin	Drive Mode	Digital InOut
P10[0] (CYBSP_SPI_MISO, CYBSP_D12)	Digital High-Z. Input buffer on	Serial Communication Block (SCB) 4 SPI.miso (mSPI) [USED]
P10[1] (CYBSP_SPI_MOSI, CYBSP_D11)	Strong Drive. Input buffer off	Serial Communication Block (SCB) 4 SPI.mosi (mSPI) [USED]
P10[2] (CYBSP_SPI_CLK, CYBSP_D13)	Strong Drive. Input buffer off	Serial Communication Block (SCB) 4 SPI.clk (mSPI) [USED]
P10[3] (CYBSP_SPI_CS, CYBSP_D10)	Strong Drive. Input buffer off	Serial Communication Block (SCB) 4 SPI.select[0] (mSPI) [USED]






Once the pins are configured, click the "chain" button  next to the **Digital InOut** connection on one of the pins. This will take you to **Peripherals** tab for the SCB you selected. Check the box to enable the SCB, then select the **SPI-<version>** from the popup menu and give it a name:

<input checked="" type="checkbox"/> Serial Communication Block (SCB) 4	mSPI	SPI-3.0
<input type="checkbox"/> Serial Communication Block (SCB) 5	scb_5	

On the right side of the screen you can configure your SPI how you want. One parameter you need to change is **Clock**. If you look in the Code Preview area, you will see the definitions that are created using the name that you choose.

Serial Communication Block (SCB) 4 (mSPI) - Parameters

Enter filter text...

Name	Value
Overview	
Configuration Help	Open SPI SCB Documentation
General	
Mode	Master
Sub Mode	Motorola
SCLK Mode	CPHA = 0, CPOL = 0
Data Rate (kbps)	1000
Oversample	16
Enable Input Glitch Filter	<input type="checkbox"/>
Enable MISO Late Sampling	<input checked="" type="checkbox"/>
SCLK Free Running	<input type="checkbox"/>
Parity	No Parity
Data Configuration	
Bit Order	MSB First
RX Data Width	8
TX Data Width	8
Slave Select	
Deassert SS Between Data Element	<input type="checkbox"/>
Setup Delay	0.75 Clock Cycles
Hold Delay	0.75 Clock Cycles
Inter-dataframe Delay	1.5 Clock Cycles
SS0 Polarity	Active Low
SS1 Polarity	Active Low
SS2 Polarity	Active Low
SS3 Polarity	Active Low
Connections	
Clock	 8 bit Divider 0 clk [USED]
SCLK	 P10[2] digital_inout (CYBSP_SPI_CLK, CYBSP_D13) [U]
MOSI	 P10[1] digital_inout (CYBSP_SPI_MOSI, CYBSP_D11) [U]
MISO	 P10[0] digital_inout (CYBSP_SPI_MISO, CYBSP_D12) [U]
SS0	 P10[3] digital_inout (CYBSP_SPLCS, CYBSP_D10) [U]

Code Preview

Enter search text...

```

/* NOTE: This is a preview only. It combines elements of the
 * cycfg_peripherals.c and cycfg_peripherals.h files located in the folder
 * C:/WORK/mtw-XMC7200_ch02/PDL_SPI_MASTER/ch02_PDL_SPI_MASTER/bps/TARGET_APP_F
 */

#include "cy_scb_spi.h"
#include "cy_sysclk.h"
#if defined(CY_USING_HAL)
#include "cyhal_hwmgr.h"
#endif //defined (CY_USING_HAL)

#define mSPI_HW SCB4
#define mSPI_IRQ scb_4_interrupt_IRQn

const cy_stc_scb_spi_config_t mSPI_config =
{
    .spiMode = CY_SCB_SPI_MASTER,
    .subMode = CY_SCB_SPI_MOTOROLA,
    .sclkMode = CY_SCB_SPI_CPHA0_CPOL0,
    .parity = CY_SCB_SPI_PARITY_NONE,
    .dropOnParityError = false,
    .oversample = 16,
    .rxDataWidth = 8UL,
    .txDataWidth = 8UL,
    .enableMsbFirst = true,
    .enableInputFilter = false,
    .enableFreeRunSclk = false,
    .enableMisoLateSample = true,
    .enableTransferSeparation = false,
    .ssPolarity = ((CY_SCB_SPI_ACTIVE_LOW << CY_SCB_SPI_SLAVE_SELECT0) | \
                  (CY_SCB_SPI_ACTIVE_LOW << CY_SCB_SPI_SLAVE_SELECT1) | \
                  (CY_SCB_SPI_ACTIVE_LOW << CY_SCB_SPI_SLAVE_SELECT2) | \
                  (CY_SCB_SPI_ACTIVE_LOW << CY_SCB_SPI_SLAVE_SELECT3))

    .ssSetupDelay = false,
    .ssHoldDelay = false,
    .ssInterFrameDelay = false,
    .enableWakeFromSleep = false,
    .rxFifoTriggerLevel = 63UL,
    .rxFifoIntEnableMask = 0UL,
    .txFifoTriggerLevel = 63UL,
    .txFifoIntEnableMask = 0UL,
    .masterSlaveIntEnableMask = 0UL,
};

#if defined(CY_USING_HAL)
const cyhal_resource_inst_t mSPI_obj =
{
    .type = CYHAL_RSC_SCB,
    .block_num = 4U,
    .channel_num = 0U,
};

```

Once you've configured your SPI, do **File > Save**, and exit the Device Configurator.

To initialize the driver, call `Cy_SCB_SPI_Init` function, providing a pointer to the `cy_stc_scb_spi_config_t` structure that is created by the configurator and a context pointer of type `cy_stc_scb_spi_context_t` that you must provide for the function to fill in.

Use the function `Cy_SCB_SPI_Read` to read a single data element from the SPI RX FIFO and `Cy_SCB_SPI_ReadArray` to read an array of data out of the SPI RX FIFO.

Similarly, use the function `Cy_SCB_SPI_Write` to write a single data element in the SPI TX FIFO and `Cy_SCB_SPI_WriteArray` to write an array of data into the SPI TX FIFO. These are some of the low-level

functions to read and write data. Similarly, there are other high-level functions. To know more about these functions and several other functions of SPI, refer to the [documentation](#).

The documentation for the PDL SPI functions can be found under **Peripheral Driver Library > PDL API Reference > SCB > SPI**. It includes several usage examples along with the full API description.

2.3.6.3 PDL vs. HAL

- Enable/Disable input glitch filter.

To configure the above parameter, you will need to use the PDL.

2.3.6.4 Interrupt Events

SPI is able to trigger interrupts in the following scenarios.

- SPI master transfer done – When SPI master transfer is completed.
- SPI Bus Error – Slave deselected unexpectedly in the SPI transfer.
- TX FIFO is not full – When the HW TX FIFO is not full.
- TX FIFO is empty - When the HW TX FIFO buffer is empty.
- TX FIFO overflows - When an attempt to write to a full HW TX FIFO buffer occurs.
- TX FIFO underflows - When an attempt to read from an empty HW RX FIFO buffer occurs.
- RX FIFO is full - When the HW RX FIFO buffer is full.
- RX FIFO is not empty - When the HW RX FIFO buffer is not empty.
- RX FIFO overflows - When an attempt to write to a full HW RX FIFO buffer occurs.
- RX FIFO underflows - When an attempt to read from an empty HW RX FIFO buffer occurs.

2.3.7 OLED Display

This will be used in [Exercise 17](#):

The CY8CKIT-028-SENSE shield contains a 128 x 64-pixel dot matrix OLED display driven by an SSD1306 controller. In order to utilize this display, we will use the OLED Display library (SSD1306). The library has been designed to work with third party graphics libraries including emWin and u8g2.

Follow the steps bellow in order to create a simple emWin application and display “Hello World” on it using I2C.

1. Create an empty application
2. Add the *display-oled-ssd1306* library to the application.
3. Add the *emWin* library to the application
4. Enable the `EMWIN_NOSNTS` *emWin* library option by adding it to the *Makefile* `COMPONENTS` list:

```
COMPONENTS+=EMWIN_NOSNTS
```

5. Place the following code in the `main.c` file:

```
#include "cybsp.h"
#include "mtb_ssd1306.h"
#include "GUI.h"

int main(void)
{
    cy_rslt_t result;
    cyhal_i2c_t i2c_obj;

    /* Initialize the device and board peripherals */
    result = cybsp_init();

    CY_ASSERT(result == CY_RSLT_SUCCESS);

    /* Initialize the I2C to use with the OLED display */
    result = cyhal_i2c_init(&i2c_obj, CYBSP_I2C_SDA, CYBSP_I2C_SCL, NULL);

    CY_ASSERT(result == CY_RSLT_SUCCESS);

    /* Initialize the OLED display */
    result = mtb_ssd1306_init_i2c(&i2c_obj);

    CY_ASSERT(result == CY_RSLT_SUCCESS);

    __enable_irq();

    GUI_Init();
    GUI_DispString("Hello world!\n");

    for(;;)
    {
    }
}
```

2.4 Interrupts

You will use this in [Exercise 7:](#) and [Exercise 8:](#)

Interrupts are an event-triggered method of context switching. They allow the CPU to do something else until an event occurs that requires the CPU's attention, at which point the CPU will stop whatever it is doing and go service the interrupt. Servicing the interrupt typically involves executing an interrupt callback function. The interrupt callback function is sometimes called an interrupt service routine (ISR) or an interrupt handler. All three terms mean the same thing.

No matter what you call it, you should minimize the amount of processing that is done inside an interrupt callback function because it will block the CPU from doing anything else until it finishes or until a higher priority interrupt occurs. This is especially true when using a real-time operating system (RTOS), which we will discuss in a later chapter.

Nearly all the peripherals are able to trigger interrupts in some way. For each peripheral we discuss in this chapter, we will briefly cover what interrupts can be set up for it. In many cases you can enable interrupts for more than one event on a given peripheral. In that case, the interrupt callback function must check the reason for the interrupt and behave accordingly.

2.4.1 Global Interrupt Enable

Before you begin making use of interrupts in your application it is important to call the `__enable_irq` function (the name starts with 2 underscores) to globally enable interrupts. If your application no longer needs interrupts you can call the `__disable_irq` function.

2.4.2 HAL

The HAL interrupt API is peripheral specific. Its documentation can be found within the HAL documentation, under each peripheral that supports interrupts.

The details vary slightly between peripherals, so you should refer to the documentation, but generally to use a HAL interrupt the procedure is:

1. Define an interrupt callback function. This may also require defining an interrupt callback data structure.
2. Initialize the peripheral as usual using HAL function(s).
3. Call a HAL function to register the callback function defined above.
4. Call a HAL function to enable the desired interrupts events and to set the interrupt priority.

The documentation for most HAL peripherals contains a Quick Start section with examples of interrupts. As additional examples, a GPIO interrupt and a PWM interrupt are shown here:

2.4.2.1 HAL GPIO Interrupt

The following code snippet shows a HAL GPIO interrupt for falling edges on an input pin (`CYHAL_GPIO_IRQ_FALL`). This is useful for cases such as a mechanical button that pulls the pin low when pressed.


```
#define GPIO_INTERRUPT_PRIORITY (7u)

/* Interrupt callback function */
static void button_isr(void *handler_arg, cyhal_gpio_event_t event)
{
    /* Place interrupt code here */
}

/* GPIO callback initialization structure */
cyhal_gpio_callback_data_t cb_data =
{
    .callback      = button_isr,
    .callback_arg  = NULL
};

int main(void)
{
    /* Initialize the device and board peripherals */
    cybsp_init();

    /* Initialize the button and setup the interrupt */
    cyhal_gpio_init(CYBSP_USER_BTN, CYHAL_GPIO_DIR_INPUT,
                   CYHAL_GPIO_DRIVE_PULLUP, CYBSP_BTN_OFF);
    cyhal_gpio_register_callback(CYBSP_USER_BTN, &cb_data);
    cyhal_gpio_enable_event(CYBSP_USER_BTN, CYHAL_GPIO_IRQ_FALL,
                           GPIO_INTERRUPT_PRIORITY, true);

    __enable_irq();

    for (;;)
    {
        /* Place main application code here */
    }
}
```

2.4.2.2 HAL PWM Interrupt

The following code snippet shows a HAL PWM interrupt for any event (CYHAL_PWM_IRQ_ALL). In this case, the interrupt callback function checks the event that caused the interrupt to decide what to do. It performs different actions for compare events and terminal count events.

```
void pwm_event_handler(void* callback_arg, cyhal_pwm_event_t event)
{
    (void)callback_arg;

    if ((event & CYHAL_PWM_IRQ_COMPARE) == CYHAL_PWM_IRQ_COMPARE)
    {
        /* Compare event triggered */
        /* Insert application code to handle event */
    }
    else if ((event & CYHAL_PWM_IRQ_TERMINAL_COUNT) == CYHAL_PWM_IRQ_TERMINAL_COUNT)
    {
        /* Terminal count event triggered */
        /* Insert application code to handle event */
    }
}
```

```
int main(void)
{
    cyhal_pwm_t pwm_obj;

    /* Initialize the device and board peripherals */
    cybsp_init();

    /* Enable global interrupts */
    __enable_irq();

    /* Initialize PWM */
    cyhal_pwm_init(&pwm_obj, CYBSP_USER_LED, NULL);
    cyhal_pwm_set_duty_cycle(&pwm_obj, 50, 1);

    /* Register interrupt callback function */
    cyhal_pwm_register_callback(&pwm_obj, pwm_event_handler, NULL);
    /* Enable all events to trigger the callback */
    cyhal_pwm_enable_event(&pwm_obj, CYHAL_PWM_IRQ_ALL, 3, true);

    /* Start the PWM output */
    cyhal_pwm_start(&pwm_obj);
}
```

2.4.3 PDL

The documentation for the PDL interrupt API can be found under **CAT1 Peripheral Driver Library > PDL API Reference > SysInt**. There are also PDL interrupt API functions that are specific to certain peripherals. The documentation for those functions can be found within the PDL documentation under each peripheral.

The details vary slightly between peripherals, so you should refer to the documentation, but generally to use a PDL interrupt the procedure is:

1. Use the Device Configurator to setup the peripheral and to select its interrupt source(s).

Note: This can also be done manually in the code using the appropriate PDL functions for the peripheral.

2. Define an interrupt callback function. The interrupt callback function must clear the interrupt.
3. Initialize a structure to specify the peripheral that is causing the interrupt and its priority and also select the required multiplexer.

Note: If there is only one interrupt then you can select any multiplexer which connects to the NVIC.

4. Use `Cy_SysInt_Init` to specify the structure and register the callback function defined above.
5. Route the peripheral's interrupt line to the nested vector interrupt controller by calling `NVIC_EnableIRQ`.

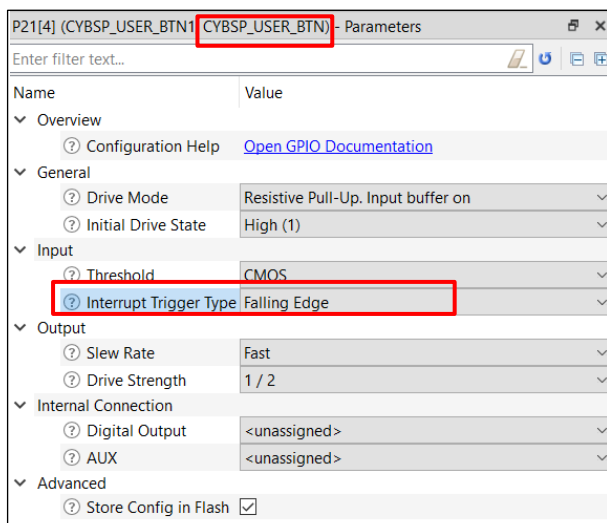
Note: The NVIC is an Arm® Cortex® hardware block that (among other things) maps interrupts from each of the interrupt sources into the CPU.

6. If necessary, use PDL functions to initialize and start the peripheral as usual.

GPIO interrupt and PWM interrupt examples are shown here. These examples accomplish the same result as the HAL examples shown above.

2.4.3.1 PDL GPIO Interrupt

The code snippet below shows a PDL GPIO interrupt for falling edges on an input pin (CY_GPIO_INTR_FALLING). For example, this would be useful in the case of a mechanical button that pulls the pin low when pressed. The Device Configurator used to configure the pin and its interrupts/name are specified as CYBSP_USER_BTN shown here:



```
#define GPIO_INTERRUPT_PRIORITY (7u)

#define PORT_INTR_MASK (0x00000001UL << CYBSP_USER_BTN_PORT_NUM)

/* Interrupt callback function */
void GPIO_Interrupt_Handler(void){
    /* Get interrupt cause */
    uint32_t intrSrc = Cy_GPIO_GetInterruptCause0();
    /* Check if the interrupt was from the user button's port */
    if(PORT_INTR_MASK == (intrSrc & PORT_INTR_MASK)){
        /* Clear the interrupt */
        Cy_GPIO_ClearInterrupt(CYBSP_USER_BTN_PORT, CYBSP_USER_BTN_NUM);

        /* Place any additional interrupt code here */
    }
}

int main(void)
{
    /* Initialize the device and board peripherals */
    cybsp_init();

    /* Enable global interrupts */
    __enable_irq();

    /* Interrupt config structure */
    cy_stc_sysint_t intrCfg =
    {
        /*.intrSrc=*/ ((NvicMux0_IRQn << 16) | CYBSP_USER_BTN_IRQ),
        /*.intrPriority=*/ GPIO_INTERRUPT_PRIORITY
    };

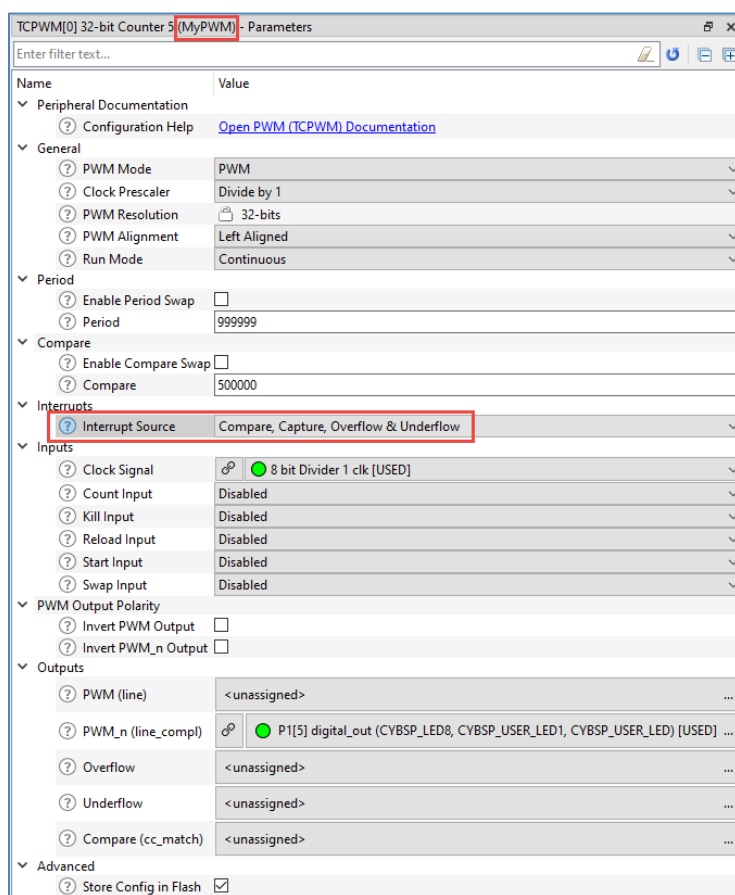
    /* Initialize the interrupt and register interrupt callback */
    Cy_SysInt_Init(&intrCfg, &GPIO_Interrupt_Handler);
    /* Enable the interrupt in the NVIC */
    NVIC_EnableIRQ((IRQn_Type) NvicMux0_IRQn);
}
```

```
for (;;)
{
    /* Place main application code here */
}
```

2.4.3.2 PDL PWM Interrupt

The following code snippet shows a PDL PWM interrupt for all events (Compare, Capture, Overflow & Underflow). In this case, the interrupt callback function checks the event that caused the interrupt and decides what to do.

The Device Configurator is used to configure the PWM and the associated pin and clock correctly. Then the PWM's name is specified as `MyPWM`. In addition, the Device Configurator is used to enable interrupts on the desired PWM events, as shown here:



```
#define PWM_INTERRUPT_PRIORITY (7u)

/* Interrupt callback function */
void PWM_Interrupt_Handler(void) {

    /* Get interrupt cause */
    uint32_t intrSrc = Cy_TCPWM_GetInterruptStatus(MyPWM_HW, MyPWM_NUM);

    if((intrSrc & CY_TCPWM_INT_ON_CC0) == CY_TCPWM_INT_ON_CC0)
    {
        /* Compare event triggered */
        /* Insert application code to handle event */
    }
    else if((intrSrc & CY_TCPWM_INT_ON_TC) == CY_TCPWM_INT_ON_TC)
    {
        /* Terminal count event triggered */
        /* Insert application code to handle event */
    }

    /* Clear all interrupt sources */
    Cy_TCPWM_ClearInterrupt(MyPWM_HW, MyPWM_NUM, intrSrc);
}

int main(void)
{
    /* Initialize the device and board peripherals */
    cybsp_init() ;

    /* Enable global interrupts */
    __enable_irq();

    /* Interrupt config structure */
    cy_stc_sysint_t intrCfg =
    {
        /*.intrSrc =*/ ((NvicMux0_IRQn << 16) | MyPWM_IRQ)
        /*.intrPriority =*/ PWM_INTERRUPT_PRIORITY
    };

    /* Initialize the interrupt and register interrupt callback */
    Cy_SysInt_Init(&intrCfg, &PWM_Interrupt_Handler);

    /* Enable the interrupt in the NVIC */
    NVIC_EnableIRQ((IRQn_Type) NvicMux0_IRQn);

    /* Initialize the TCPWM block */
    Cy_TCPWM_PWM_Init(MyPWM_HW, MyPWM_NUM, &MyPWM_config);
    /* Enable the TCPWM block */
    Cy_TCPWM_PWM_Enable(MyPWM_HW, MyPWM_NUM);
    /* Start the PWM */
    Cy_TCPWM_TriggerStart_Single(MyPWM_HW, MyPWM_NUM);

    for (;;)
    {
        /* Place main application code here */
    }
}
```

2.5 Exercises

Each exercise in this section uses either the HAL or PDL to drive peripherals. Depending on your needs, you can pick and choose which exercises to practice.

Exercise 1: (GPIO-HAL) Blink an LED

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.1](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex01_HAL_blinkled** using **Empty App** as the template.
- ☐ 2. Add code to *main.c* before the infinite loop to initialize `CYBSP_USER_LED` as a strong drive digital output.

Note: This must be placed after the call to `cybsp_init` so that the board is initialized first.

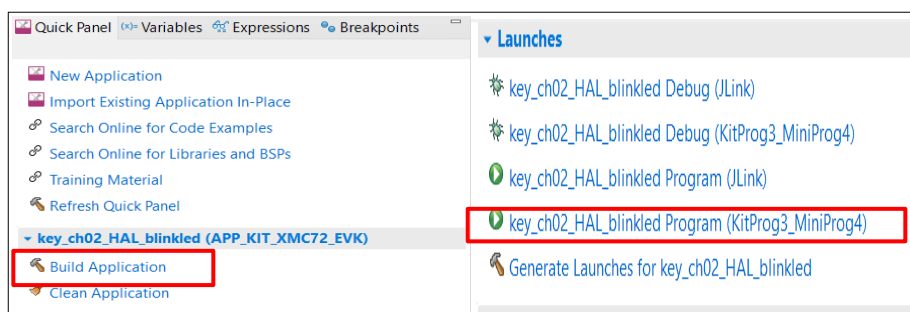
- ☐ 3. Add code to *main.c* in the infinite loop to do the following:
 - a. Drive `CYBSP_USER_LED` low
 - b. Wait 250 ms
 - c. Drive `CYBSP_USER_LED` high
 - d. Wait 250 ms

Note: See the HAL API documentation for the GPIO functions to drive the pin high and low.

Note: Use the `cyhal_system_delay_ms` function for the delay.

- ☐ 4. Program your project to your kit and verify its behavior.

Note: If you are using the Eclipse IDE for ModusToolbox™, use the link in the Quick Panel that says "ch02_ex01_HAL_blinkled Program (KitProg3_MiniProg4) to build the application and then program the kit as shown in the example image.

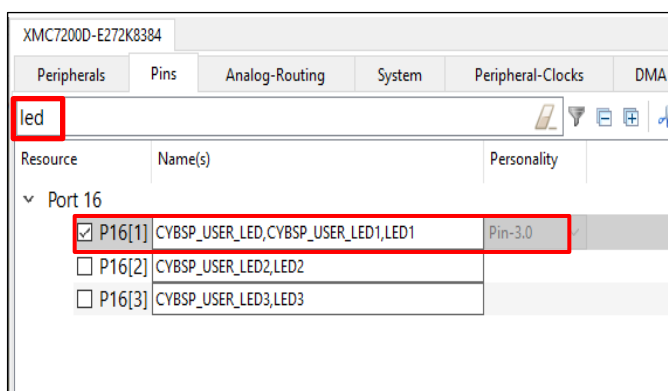


Exercise 2: (GPIO-PDL) Blink an LED

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.1](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex02_PDL_blinkled** using the **Empty App** as the template.
- ☐ 2. Use the Device Configurator to enable the pin connected to the user LED.

To do this, navigate to the **Pins** Tab and enter the filter text "led." Enable the pin that has the name "CYBSP_USER_LED":



- ☐ 3. Set the pin's **Drive Mode** parameter to **Strong Drive. Input buffer off**.
- ☐ 4. Add code to *main.c* in the infinite loop to do the following:
 - a. Drive the pin low
 - b. Wait 250 ms
 - c. Drive the pin high
 - d. Wait 250 ms

Note: See the PDL API documentation for the GPIO functions to drive the pin high and low.

Note: Use the *Cy_SysLib_Delay* function for the delay.

Note: The LED is active low, meaning it will turn on when you drive the pin low.

- ☐ 5. Program your project to your kit and verify its behavior.

Exercise 3: (GPIO-HAL) Add debug printing to the LED blink project

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.4](#).

- ☐ 1. Use Project Creator to create a new application called **ch03_ex03_HAL_blinkled_print** using the **Browse** button on the **Select Application** page to select your exercise 1 (ch02_ex01_HAL_blinkled) as a template.
- ☐ 2. Include the retarget-io library using the Library Manager.
- ☐ 3. In *main.c*, before the infinite loop, call the following function to initialize retarget-io to use the debug UART port:

```
cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,  
CY_RETARGET_IO_BAUDRATE);
```

Note: Remember to #include "cy_retarget_io.h".

- ☐ 4. Add `printf` calls to print "LED OFF" and "LED ON" at the appropriate times.

Note: Remember to use `\n` to create a new line so that information is printed on a new line each time the LED changes.

Note: If your serial terminal emulator does not support adding a carriage return for each new line, you may want to use `\n\r` instead of just `\n` each `printf` statement. You can also add `CY_RETARGET_IO_CONVERT_LF_TO_CRLF` to the `DEFINES` variable in the Makefile to automatically convert `\n` (new line) to `\n\r` (new line and carriage return) when it is sent out over the UART.

- ☐ 5. Program your project to your kit.
- ☐ 6. Open a serial terminal window with a baud rate of 115200 and observe the messages being printed.

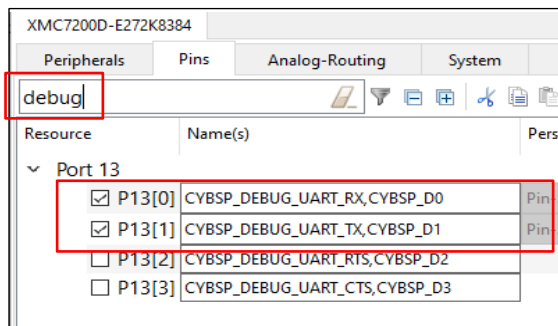
Note: If you need a refresher on using a serial terminal emulator, see ModusToolbox™ Level 1 Getting Started class, Tools chapter: Serial Terminal Emulator section.

Exercise 4: (GPIO-PDL) Add debug printing to the LED blink project

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.4](#).

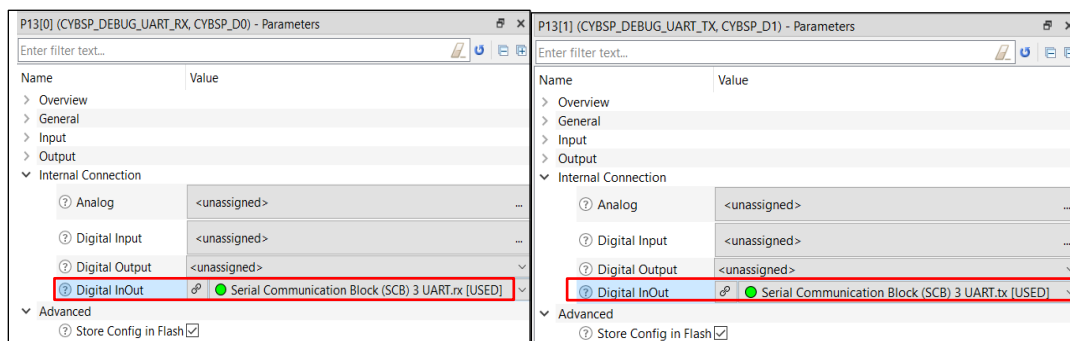
- ☐ 1. Use Project Creator to create a new application called **ch02_ex04_PDL_blinkled_print** using the **Browse** button on the **Select Application** page to select your exercise 2 (ch02_ex02_PDL_blinkled) as a template.
- ☐ 2. Use the Device Configurator to enable the debug UART.


To do this, navigate to the **Pins** Tab and enter the filter text "debug." Enable the pins that have the names "CYBSP_DEBUG_UART_RX" and "CYBSP_DEBUG_UART_TX":



Set the RX pin's Drive Mode parameter to **Digital High-Z. Input buffer on** and the TX pin's Drive Mode to **Strong Drive. Input buffer off**.

Set the RX pin's Digital InOut parameter to the **UART.rx** connection of an SCB and the TX pin's Digital InOut parameter to the **UART.tx** connection of the same SCB:



Click the "chain" button  next to the Digital InOut parameter you just selected. This will take you to a page where you can configure the SCB you just connected. You must enable the SCB block, select UART for the personality, and select a clock divider (pick any unused divider).

Give the UART a name that will be easy to use in the code, such as UART.

- ☐ 3. In *main.c*, add calls to `Cy_SCB_UART_Init` and `Cy_SCB_UART_Enable` to initialize and enable your UART peripheral.
- ☐ 4. Add calls to function `Cy_SCB_UART_PutString` to print "LED OFF" and "LED ON" at the appropriate times.

Note: Remember to use \n to create a new line so that information is printed on a new line each time the LED changes.

- ☐ 5. Program your project to your kit.
- ☐ 6. Open a terminal window with a baud rate of 115200 and observe the messages being printed.

Note: If you need a refresher on using a serial terminal emulator, see ModusToolbox™ Level 1 Getting Started class, Tools chapter: Serial Terminal Emulator section.

Exercise 5: (GPIO-HAL) Read the state of a mechanical button

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.1](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex05_HAL_button** using the **Empty App** as the template.
- ☐ 2. In *main.c*, initialize the pin for the button (CYBSP_USER_BTN) as an input with a resistive pullup and initialize the LED (CYBSP_USER_LED) as a strong drive output.

Note: The button pulls the pin to ground when pressed. An input with a resistive pullup is required so that the pin is pulled high when the button is not being pressed.

- ☐ 3. In the infinite loop, check the state of the button. Turn the LED ON if the button is pressed and turn it OFF if the button is not pressed.
- ☐ 4. Program your project to your kit and press the button to observe the behavior.

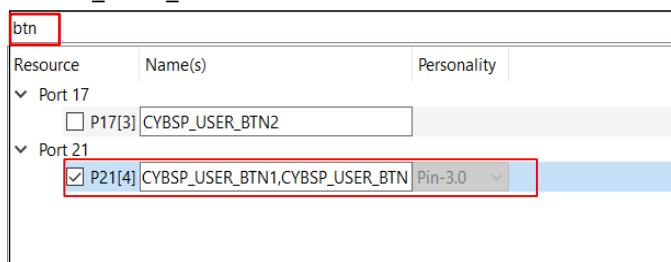
Note: Be sure to press the correct user button, not the reset button. If you press the reset button, the kit will reset and will restart the firmware execution from the beginning.

Exercise 6: (GPIO-PDL) Read the state of a mechanical button

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.1](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex06_PDL_button** using the **Empty App** as the template.
- ☐ 2. Using the Device Configurator, enable the pin connected to a user button as a resistive pullup input (**Resistive Pull-Up. Input buffer on**) then enable the pin connected to the user LED as a strong drive output (**Strong Drive. Input buffer off**).

To enable the button pin, navigate to the **Pins** Tab and enter the filter text "btn." Enable the pin named "CYBSP_USER_BTN" as shown below:



*Note: The button pulls the pin to ground when pressed. A drive mode of **Resistive Pull-Up, Input buffer on** is required so that the pin is pulled high when the button is not being pressed.*

- ☐ 3. In the infinite loop, check the state of the button. Turn the LED ON if the button is pressed and turn it OFF if the button is not pressed.
- ☐ 4. Program your project to your kit and press the button to observe the behavior.

Exercise 7: (GPIO-HAL) Use an interrupt to toggle the state of an LED

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.4](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex07_HAL_interrupt** using the **Browse** button on the **Select Application** page and select your previous exercise (ch02_ex03_HAL_button) as a template.

- ☐ 2. In main.c, register a callback function to the button by calling `cyhal_gpio_register_callback`.

Note: Look at the function documentation to find out which arguments it takes.

Note: Use `NULL` for `callback_arg`

- ☐ 3. Set up a falling edge interrupt for the GPIO connected to the button.

Note: See the documentation for `cyhal_gpio_enable_event`.

- ☐ 4. In your C code:
- Type `cyhal_gpio_enable_event`.
 - Highlight `cyhal_gpio_enable_event`, right click on it, and select **Open Declaration**. This will show the required parameters for the function.
 - Highlight `cyhal_gpio_event_t`, right click on it, and select **Open Declaration**.
 - Identify the correct value to use for a falling edge interrupt.

- ☐ 5. Create the interrupt service routine (ISR) so that it toggles the state of the LED each time the button is pressed.

Your ISR should resemble this:

```
void button_isr(void *handler_arg, cyhal_gpio_event_t event)
{
    <your code here>
}
```

Note: You can use the function `cyhal_gpio_toggle`.

- ☐ 6. Program your project to your kit and press the button to observe the behavior.

Exercise 8: (GPIO-PDL) Use an interrupt to toggle the state of an LED

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.4](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex08_PDL_interrupt** using the **Browse** button on the **Select Application** page to select your exercise 6 (ch02_ex06_PDL_button) as a template.
- ☐ 2. Open the Device Configurator and set the button pin to have a falling edge interrupt.
- ☐ 3. In the *main.c* file, set up a falling edge interrupt for the GPIO connected to the button.

Note: See the **CAT1 Peripheral Driver Library > PDL API Reference > SysInt** documentation for how to set up interrupts.

Note: You will need to call the following functions:

- `Cy_SysInt_Init`
- `NVIC_EnableIRQ`

Note: You will need to create a structure of type `cy_stc_sysint_t` to configure the interrupt. The `.intrSrc` member of this struct should be set to `(NvicMux0_IRQn << 16 | CYBSP_USER_BTN_IRQ)`. This macro is defined in the file `cycfg_pins.h`, which is automatically generated by the Device Configurator.

Note: Don't forget to enable interrupts by calling the function `__enable_irq`.

Note: Optionally, you can call the function `Cy_GPIO_SetFilter`. This will route the pin's input through a 50ns glitch filter.

- ☐ 4. Create the interrupt service routine (ISR) so that it toggles the state of the LED each time the button is pressed.

Your ISR should look something like this:

```
void Interrupt_Handler(void)
{
    <Your code here>
}
```

Note: You can use the function `Cy_GPIO_Inv` to invert the GPIO's state.

Note: Don't forget to clear the interrupt in your ISR. You can use the function `Cy_GPIO_ClearInterrupt`.

Note: Optionally, in your ISR you can use the function `Cy_GPIO_GetInterruptCause0` if you're using the XMC72_EVK kit to verify what GPIO port generated the interrupt.

- ☐ 5. Remove all of the code from the infinite `for(;;)` loop since the LED will be controlled by the interrupt.
- ☐ 6. Program your project to your kit and press the button to observe the behavior.

Exercise 9: (PWM-HAL) LED Brightness

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.2](#).

- ☐ 1. Utilize Project Creator to create a new application called **ch02_ex09_HAL_pwm** using the **Empty App** as the template.
- ☐ 2. In the C file, use a PWM to drive `CYBSP_USER_LED` instead of using the GPIO functions.
- ☐ 3. Configure the PWM and change the duty cycle in the main loop so that the LED gradually changes intensity.

Note: If you chose a period of 100, you can easily set the duty cycle from 0 to 100 by changing the compare value. Just be sure to use a clock that (even when divided by 100) is faster than what is visible by the human eye. This way, the LED appears dim instead of blinking.

Note: Don't forget to call the `cyhal_pwm_start` function after you call `cyhal_pwm_init`.

Note: Use a delay so that the intensity goes from 0% to 100% in one second.

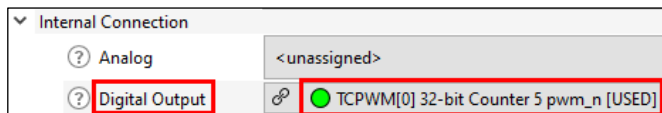
- ☐ 4. Program your project to your kit and observe the behavior.


Exercise 10: (PWM-PDL) LED Brightness

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.2](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex10_PDL_pwm** using **Empty App** as the template.
- ☐ 2. Use the Device Configurator to enable a PWM connected to the user LED.

To do this, navigate to the **Pins** Tab and enable the user LED pin. Select a TCPWM for this pin's Digital Output parameter:



Click the "chain" button  next to the Digital Output parameter you just selected. This will take you to a page where you can configure the TCPWM you just connected. Remember to set the period to 100 in the TCPWM configuration.

- ☐ 3. Change the PWM's duty cycle in the main loop so that the LED gradually changes intensity.

Note: Don't forget to change the name of the PWM to something convenient that you can use in the code.

Note: If you chose a period of 100, you can easily set the duty cycle from 0 to 100 by changing the compare value. Just be sure to use a clock that (even when divided by 100) is faster than what is visible by the human eye. This way, the LED appears dim instead of blinking.

Note: Use the `Cy_TCPWM_PWM_SetCompare0` function to do this.

Note: Don't forget to call the `Cy_TCPWM_TriggerStart_Single` function after you call `Cy_TCPWM_PWM_Init` and `Cy_TCPWM_PWM_Enable`.

Note: Use a delay so that the intensity goes from 0% to 100% in one second.

- ☐ 4. Program your project to your kit and observe the behavior.

Exercise 11: (ADC READ-HAL) Read potentiometer sensor value via an ADC

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.3](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex11_HAL_adc_read** using the **Empty App** as the template.
- ☐ 2. Add the retarget-io library and initialize it in main.c. Don't forget to include the header.
- ☐ 3. Update the code so that every 100 ms the potentiometer's voltage is read using an ADC. Print the values using `printf`.

Note: You can find POT pin number by looking at the Device Configurator.

Note: You can use the function `cyhal_adc_read_uv` to get a result in **microvolts**.

- ☐ 4. Program your project to your kit and observe the range of values for the potentiometer.

Exercise 12: (ADC READ-PDL) Read potentiometer sensor value via an ADC

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.3](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex12_PDL_adcread** using the **Empty App** as the template.
- ☐ 2. At the top of *main.c* add `#include <stdio.h>`
- ☐ 3. Use the Device Configurator to enable/configure the debug UART pins and associated SCB block as a UART. Give it an easy to use name such as UART.

- ☐ 4. Use the Device Configurator to enable the SAR ADC. Give it an easy to use name such as ADC.

Select any unused clock divider.

Set the SAR ADC parameter **Number of Channels** to "1".
Provide the Potentiometer pin as input to this channel.

Note: You can find POT pin number by looking at the Device Configurator.

Note: Don't forget to enable the group end as it is used to check interrupt status.

- ☐ 5. Write the code so that every 100 ms the potentiometer's voltage is read using an ADC. Print the values using `Cy_SCB_UART_PutString`.

Note: Do the required calculation to convert counts to microvolts. SAR2 ADC doesn't provide a predefined API to convert counts to volt.

Note: Use the *stdio.h* function *sprintf* to create the strings to pass to `Cy_SCB_UART_PutString`.

- ☐ 6. Program your project to your kit and observe the range of values for the potentiometer.

Exercise 13: (UART-HAL) Read a value using the standard UART functions

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.4](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex13_HAL_UartReceive** using the **Browse** button to select your previous exercise (ch02_ex02_HAL_blinkled_print) as a template.

- ☐ 2. Update the code so that it uses the HAL to look for characters from the UART.
If it receives a 1, turn on an LED. If it receives a 0, turn off an LED. Ignore any other characters.

Note: Remove the code for the button press and its interrupt.

Note: The HAL function to receive a single character over UART is `cyhal_uart_getc`

- ☐ 3. Program your project to your kit.
- ☐ 4. Open a terminal window and press the 1 and 0 keys on the keyboard. Then observe the LED turn on/off.

Exercise 14: (UART-PDL) Read a value using the standard UART functions

This exercise uses either the KIT_XMC72_EVK. This material is covered in section [2.3.4](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex14_PDL_UartReceive** using the **Browse** button to select your ch02_ex06_PDL_blinkled_print exercise as a template.

Note: The starting application already has the debug UART enabled in the configurator.

- ☐ 2. Update the code in the `for (;)` loop so that it looks for characters from the UART.
If it receives a 1, turn on an LED. If it receives a 0, turn off an LED. Ignore any other characters.

Note: Remove the code that blinks the LED and prints to the UART.

Note: Use the function `Cy_SCB_UART_Get` to receive the data.

- ☐ 3. Program your project to your kit.
- ☐ 4. Open a terminal window and press the 1 and 0 keys on the keyboard and observe the LED turn on/off.

Exercise 15: (UART-HAL) Write a value using the standard UART functions

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.4](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex15_HAL_Uartsend** using the **Browse** button to select your ch02_ex04_HAL_interrupt exercise as a template.
- ☐ 2. Modify the C file so that the number of times the button has been pressed is sent out over the UART interface whenever the button is pressed.

For simplicity, just count from 0 to 9 and then wrap back to 0 so that you only have to send a single character each time.

Note: Set a flag variable inside the ISR and then do the UART send function in the main application loop. Make sure the flag variable is defined as a volatile global variable.

Note: The function to send a single character over UART is `cyhal_uart_putc`

- ☐ 3. Program your project to your kit.
- ☐ 4. Open a terminal window. Press the button and observe the value displayed in the terminal.

Note: If you are using `printf` rather than `cyhal_uart_putc`, you will need to also send a '\n' character as well to send the data.

Exercise 16: (UART-PDL) Write a value using the standard UART functions

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.4](#).

- ☐ 1. Use Project Creator to create a new application called **ch02_ex16_PDL_Uartsend** using the **Browse** button to select your ch02_ex08_PDL_interrupt exercise as a template.
- ☐ 2. Use the Device Configurator to enable/configure the debug UART pins and the appropriate SCB block. Use an easy to remember name for the SCB such as UART.
- ☐ 3. Modify the C file so that the number of times the button has been pressed is sent out over the UART interface whenever the button is pressed.

For simplicity, just count from 0 to 9 and then wrap back to 0 so that you only have to send a single character each time.

Note: Set a flag variable inside the ISR and then do the UART send function in the main application loop. Make sure the flag variable is defined as a volatile global variable.

Note: Try using a function other than Cy_SCB_UART_PutString to send the data, such as the function Cy_SCB_UART_Put

- ☐ 4. Program your project to your kit.
- ☐ 5. Open a terminal window. Press the button and observe the value displayed in the terminal.

Exercise 17: Install shield support libraries and use the OLED display

This exercise uses the KIT_XMC72_EVK and CY8CKIT-028-SENSE shield. This material is covered in sections [2.1](#) and [2.3.7](#).

- ☐ 1. Launch the Project Creator tool from the Eclipse IDE, select your kit name, and use the **Empty App** example application as a template. Name your application **ch02_ex17_OLEDdisplay**.
 - ☐ 2. Launch the Library Manager tool and add the *display-oled-ssd1306* and *emWin* libraries.
 - ☐ 3. Update the *Makefile* COMPONENTS variable to read COMPONENTS+=EMWIN_NOSNTS
 - ☐ 4. Once you have installed the libraries, click on the *mtb_shared* directory from inside the Eclipse IDE Project Explorer. You should see the libraries that you just installed.
 - ☐ 5. In main.c, Follow these steps:
 - Include the following header files:

```
#include "cybsp.h"
#include " mtb_ssd1306.h"
#include "GUI.h"
```
 - Initialize the device and board peripherals.
 - Initialize the I2C to use with the OLED display.
 - Use the function GUI_DispString to display the text in OLED Display.
- Note:* refer to the *OLED Display* section of this document to learn how to initialize I2C for OLED display and how to display text.
- ☐ 6. Program your project to your kit and observe the OLED display.

Exercise 18: (I2C READ-HAL) Read sensor values over I2C

This exercise uses the KIT_XMC72_EVK and CY8CKIT-028-SENSE shield. This material is covered in sections [2.1](#) and [2.3.5](#)

- ☐ 1. Use Project Creator to create a new application called **ch02_ex18_I2Cread** and use the **Empty App** as the template.
- ☐ 2. Use the library manager to add the *sensor-xensiv-dps3xx* library to get access to the pressure sensor and the *retarget-io* library to allow `printf`.
- ☐ 3. Initialize the *retarget-io* library in *main.c* and include the header file.
- ☐ 4. Add code so that every 100ms the motion sensor's pressure and temperature data are read from the I2C slave.

*Note: Look at the documentation in the *sensor-xensiv-dps3xx* library to view an example of how to read the data. Remember to include the header file.*

Note: Aliases for your kit's I2C SCL and SDA pins are defined in the BSP. You can use the Device Configurator to find them by entering "I2C" in the search box on the Pins tab.

- ☐ 5. Print the temperature and pressure values to the terminal using `printf`.
- ☐ 6. Make sure the CY8CKIT-028-SENSE shield is plugged in (this is where the pressure sensor is located), program your kit and observe the results on the UART.

Exercise 19: (SPI-HAL) Transfer data using SPI

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.6](#).

In this exercise two SCB instances on the same device are used. The pins for each SCB interface will be wired together so that data can be sent between them. One SCB instance is configured as a master while the other SCB instance is configured as a slave. If available, you can also use two separate kits.

You must connect the MISO, MOSI, CLK and SSEL pins of both the SCB instances using jumper wires.


- ☐ 1. Use Project Creator to create a new application called **ch02_ex19_HAL_SPI** using the **Empty App** as the template.
- ☐ 2. In main.c, Initialize the user LED using the function `cyhal_gpio_init`.
- ☐ 3. Initialize the SPI master using one SCB instance by providing the correct port and pin numbers in the `cyhal_spi_init` function with SPI-mode-00. Also set the required frequency for SPI master.
- ☐ 4. Similarly Initialize the SPI slave using another SCB instance by providing the pin numbers and set the same frequency as the SPI master.
- ☐ 5. Write the code in main.c such that the SPI master sends the commands LED_ON (1) and LED_OFF (0) alternately with a delay in between (e.g. 1000 ms). The command values are read by the SPI slave. Pass the command received by the SPI slave as an argument to the `cyhal_gpio_write` function to turn the user LED ON when 1 is received and OFF when 0 is received.
- ☐ 6. Connect wires between the MISO, MOSI, CLK and SSEL pins of the two SCB instances.
- ☐ 7. Program the kit and observe the blinking LED.






Exercise 20: (SPI-PDL) Transfer data using SPI

This exercise uses the KIT_XMC72_EVK. This material is covered in section [2.3.6](#).

In this exercise two SCB instances on the same device are used. The pins for each SCB interface will be wired together so that data can be sent between them. One SCB instance is configured as a master while the other SCB instance is configured as a slave. If available, you can also use two separate kits.

You must connect the MISO, MOSI, CLK and SSEL pins of both the SCB instances using jumper wires.

- ☐ 1. Use Project Creator to create a new application called **ch02_ex20_PDL_SPI** using the **Empty App** as the template.
- ☐ 2. Configure the User LED pin to drive the LED.
- ☐ 3. Open Device Configurator and select **miso**, **mosi**, **clk** and **ssel** pins. You must select a set of pins that connect to the same SCB. Configure the corresponding drive mode for each pin. This configuration is for master SPI instance.
- ☐ 4. Click the  button next to one of the pins to go to the associated SCB block. Next enable the SCB block and provide a name such as “mSPI” and set up the required configuration as shown below.

Serial Communication Block (SCB) 4 (mSPI) - Parameters	
Enter filter text...	
Name	Value
<div>Overview</div> <div> <div>Configuration Help</div> <div>Open SPI SCB Documentation</div> </div>	
<div>General</div> <div> <div>Mode</div> <div>Master</div> </div> <div> <div>Sub Mode</div> <div>Motorola</div> </div> <div> <div>SCLK Mode</div> <div>CPHA = 0, CPOL = 0</div> </div> <div> <div>Data Rate (kbps)</div> <div>1000</div> </div> <div> <div>Oversample</div> <div>16</div> </div> <div> <div>Enable Input Glitch Filter</div> <div><input type="checkbox"/></div> </div> <div> <div>Enable MISO Late Sampling</div> <div><input checked="" type="checkbox"/></div> </div> <div> <div>SCLK Free Running</div> <div><input type="checkbox"/></div> </div> <div> <div>Parity</div> <div>No Parity</div> </div>	
<div>Data Configuration</div> <div> <div>Bit Order</div> <div>MSB First</div> </div> <div> <div>RX Data Width</div> <div>8</div> </div> <div> <div>TX Data Width</div> <div>8</div> </div>	
<div>Slave Select</div> <div> <div>Deassert SS Between Data Element</div> <div><input type="checkbox"/></div> </div> <div> <div>Setup Delay</div> <div>0.75 Clock Cycles</div> </div> <div> <div>Hold Delay</div> <div>0.75 Clock Cycles</div> </div> <div> <div>Inter-dataframe Delay</div> <div>1.5 Clock Cycles</div> </div> <div> <div>SS0 Polarity</div> <div>Active Low</div> </div> <div> <div>SS1 Polarity</div> <div>Active Low</div> </div> <div> <div>SS2 Polarity</div> <div>Active Low</div> </div> <div> <div>SS3 Polarity</div> <div>Active Low</div> </div>	
<div>Connections</div> <div> <div>Clock</div> <div> 8 bit Divider 0 clk [USED]</div> </div> <div> <div>SCLK</div> <div> P10[2] digital_inout (CYBSP_SPI_CLK, CYBSP_D13) [USED]</div> </div> <div> <div>MOSI</div> <div> P10[1] digital_inout (CYBSP_SPI_MOSI, CYBSP_D11) [USED]</div> </div> <div> <div>MISO</div> <div> P10[0] digital_inout (CYBSP_SPI_MISO, CYBSP_D12) [USED]</div> </div> <div> <div>SS0</div> <div> P10[3] digital_inout (CYBSP_SPI_CS, CYBSP_D10) [USED]</div> </div>	

- ☐ 5. Repeat steps 2 and 3 to configure another set of SPI pins and SCB instance for the SLAVE SPI. Then save and close the Device Configurator.
- ☐ 6. In main.c initialize and enable the SPI-SCB instance for the master and slave.
- ☐ 7. In the for loop, alternately send commands 0 and 1 from the SPI master with a delay in between (e.g. 1000 ms). Receive the commands through the SPI slave. You can wait until data transmission is completed by the SPI master using the `Cy_SCB_SPI_IsTxComplete` function and then wait until the command is received in the slave FIFO using the `Cy_SCB_SPI_GetNumInRxFifo` function.
- ☐ 8. Pass the received command as an argument in the `Cy_GPIO_Write` function so that the LED is turned ON and OFF based on the command sent by the SPI slave.
- ☐ 9. Connect wires between the MISO, MOSI, CLK and SSEL pins of the two SCB instances.
- ☐ 10. Program your project to your kit and observe the blinking LED.

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Published by
Infineon Technologies AG
81726 Munich, Germany

© 2022 Infineon Technologies AG.
All Rights Reserved.

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.