Chapter 2: Peripherals

At the end of this chapter you will be able to write firmware for MCU peripherals (e.g. GPIOs, PWMs, ADCs, UART, and I2C) and to interface with shield peripherals (e.g. ambient light sensor, motion sensor, and TFT 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

Document conventions

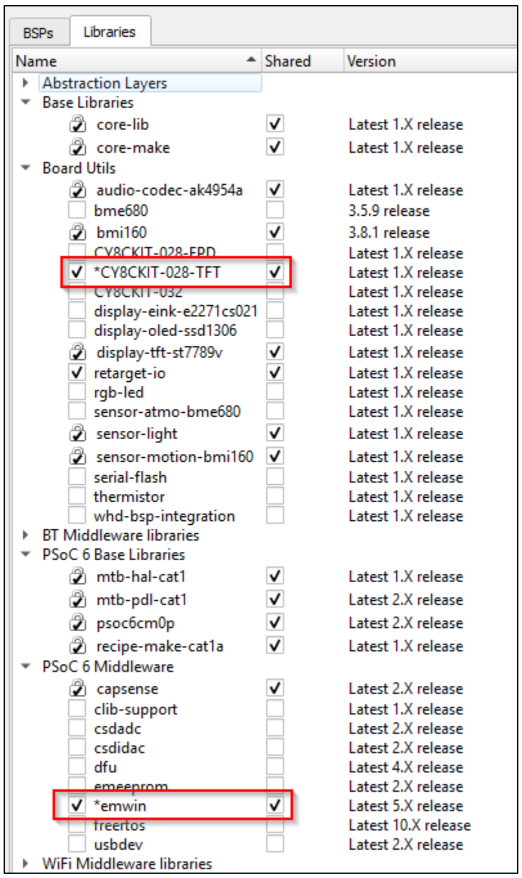
Convention	Usage	Example
`Courier New`	Displays code and text commands	`CY_ISR_PROTO(MyISR);` `make build`
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > 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 15: and Exercise 16:

ModusToolbox™ software includes libraries that make it easier to work with the peripherals on a given kit. In our case, we are using a PSoC™ 6 kit along with a display 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-TFT and emWin libraries.



The CY8CKIT-028-TFT library provides support for initializing/freeing all the hardware peripheral resources on the shield, defining all pin mappings from the baseboard's Arduino interface to the shield peripherals, and providing access to each of the underlying peripherals on the shield. This library makes use of several support libraries: display-tft-st7789v, sensor-light, sensor-motion-bmi160, and audio-codec-ak4954a. These libraries are added automatically when you added the CY8CKIT-028-TFT library.

The emWin library is a graphics package created by SEGGER that has been optimized for the embedded environment. It allows you to call simple functions like `GUI_DispString("Hello World!")` to easily create complex graphical displays and user interfaces.

Documentation for each of these libraries can be found in the *mtb_shared/<lib name>/<version>/docs* directory.

Next, open the project's *Makefile*. In order to be able to use the display, you need to enable the `EMWIN_NOSNTS` library option by adding it to the `COMPONENTS` variable.

```
COMPONENTS+=EMWIN_NOSNTS
```

2.2 PDL vs. HAL

As you leaned in the ModusToolbox™ 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 PSoC™ 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 an easy to use API to initialize your pin. The PSoC™ 4 and PSoC™ 6 device families each have their own unique PDL. 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 uartConfig =
{
    .uartMode                   = CY_SCB_UART_STANDARD,
    .enableMutliProcessorMode   = false,
    .smartCardRetryOnNack       = false,
    .irdaInvertRx               = false,
    .irdaEnableLowPowerReceiver = false,

    .oversample                 = 12UL,

    .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            = 0UL,
    .receiverAddressMask        = 0UL,
    .acceptAddrInFifo           = false,

    .enableCts                  = false,
    .ctsPolarity                = CY_SCB_UART_ACTIVE_LOW,
    .rtsRxFifoLevel             = 0UL,
    .rtsPolarity                = CY_SCB_UART_ACTIVE_LOW,

    .rxFifoTriggerLevel  = 0UL,
    .rxFifoIntEnableMask = 0UL,
    .txFifoTriggerLevel  = 0UL,
    .txFifoIntEnableMask = 0UL,
};
```

HAL UART Initialization Structure

```
const cyhal_uart_cfg_t uart_config =
{
    .data_bits      = DATA_BITS_8,
    .stop_bits      = STOP_BITS_1,
    .parity         = CYHAL_UART_PARITY_NONE,
    .rx_buffer      = rx_buf,
    .rx_buffer_size = RX_BUF_SIZE
};
```

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 ModusToolbox™ Quick Panel or in the docs directory of the library.

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. Typically, applications written for PSoC™ 4 devices follow this trend, as PSoC™ 4 devices have relatively small amounts of flash memory. PSoC™ 6 applications on the other hand typically include the HAL, since they are not nearly as memory limited as PSoC™ 4 devices. In this chapter we will look at how to initialize and use GPIOs, PWMs, ADCs, UART, and I^{2}C, using the PSoC™ 4 PDL (Cat2), the PSoC™ 6 PDL (Cat1), and the HAL. The HAL exercises in this chapter are done with the PSoC™ 6 kit, since PSoC™ 4 applications almost never use the HAL, but the PDL exercises can be done using either the PSoC™ 4 kit or the PSoC™ 6 kit.

The abbreviation "Cat" stands for "Category". Since the PDL is architecture-specific, different categories are created whenever a new architecture requires different PDL functions. The Cat1 and Cat2 PDL libraries have nearly identical APIs, but they do differ slightly in some places. For the specifics of each library it is best to refer to their respective documentation:

- CAT1 PDL Reference Manual
- CAT2 PDL Reference Manual
- CAT1 HAL Reference Manual
- CAT2 HAL Reference Manual

Note: *Most of Infineon's PSoC™ 6 code examples use the HAL while most PSoC™ 4 code examples use the PDL. If you want to create a PSoC™ 6 application using the PDL and would like a code example for reference, you could create a relevant PSoC™ 4 code example and use that to see how the device configuration is performed and how the API is used.*

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

For example, the HAL API documentation has sections for UART, I^{2}C, and SPI. However, in PSoC™ 6, those functions are all implemented in a programmable serial communication block (SCB). So, the PDL API documentation for all three functions will be found under the heading SCB.

As a second example, the HAL has and API for analog to digital conversion (ADC). In PSoC™ 6, the ADC is implemented using a successive approximation register ADC so the PDL API documentation will be found under the heading SAR.

As a final example, the HAL has separate APIs for Timer/Counter and PWM. In the PSoC™ 6, those functions are implemented by the TCPWM hardware block so 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, care should be used when exercising this method to be certain that the PDL functions do not interfere with the HAL operation.

2.4 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 here, we will only discuss the basic flow of 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`.

2.4.1 GPIO

You will use this in Exercise 1: , Exercise 3: , Exercise 5: and Exercise 7:

2.4.1.1 Drive Mode

A drive mode is essentially a specific electrical configuration that a GPIO can take on. The PSoC™ 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 I^2C.
- 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 **CAT2 Peripheral Driver Library > PDL API Reference > GPIO > Pin drive mode**

2.4.1.2 HAL

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

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.

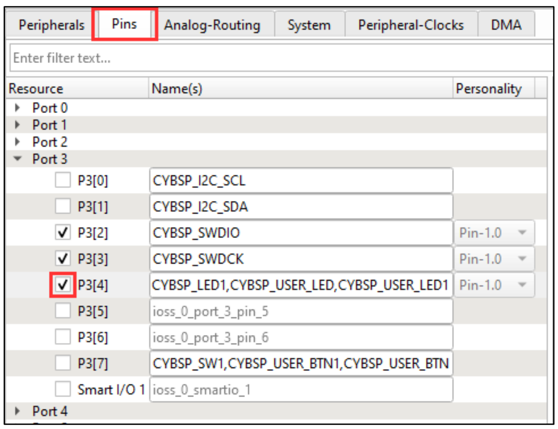
```
cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT, CYHAL_GPIO_DRIVE_STRONG, 1);


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 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 HAL, to reconfigure the pin 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**.

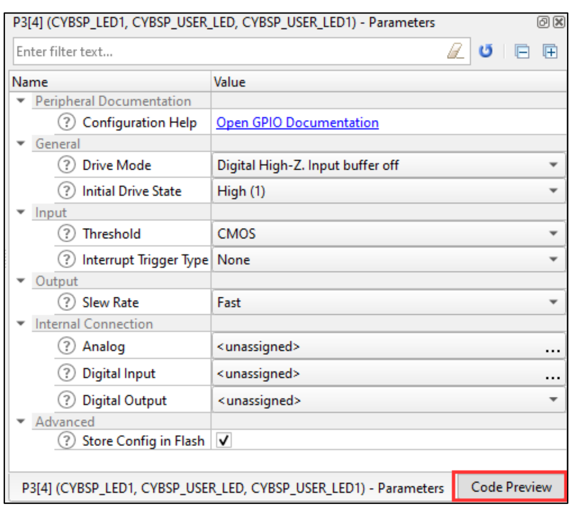
2.4.1.3 PDL

To initialize a GPIO using the PDL, 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 enter your own application-specific name if you desire, 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 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 how you want, then do **File > Save**, and exit the device configurator.

All that's left for you to do now 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 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 example 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 PSoC PDL GPIO functions can be found under **Peripheral Driver Library > PDL API Reference > GPIO > Functions**.

2.4.1.4 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.

2.4.1.5 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

2.4.2 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. In PSoC™ 4 and PSoC™ 6 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.

2.4.2.1 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 or you can 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%.

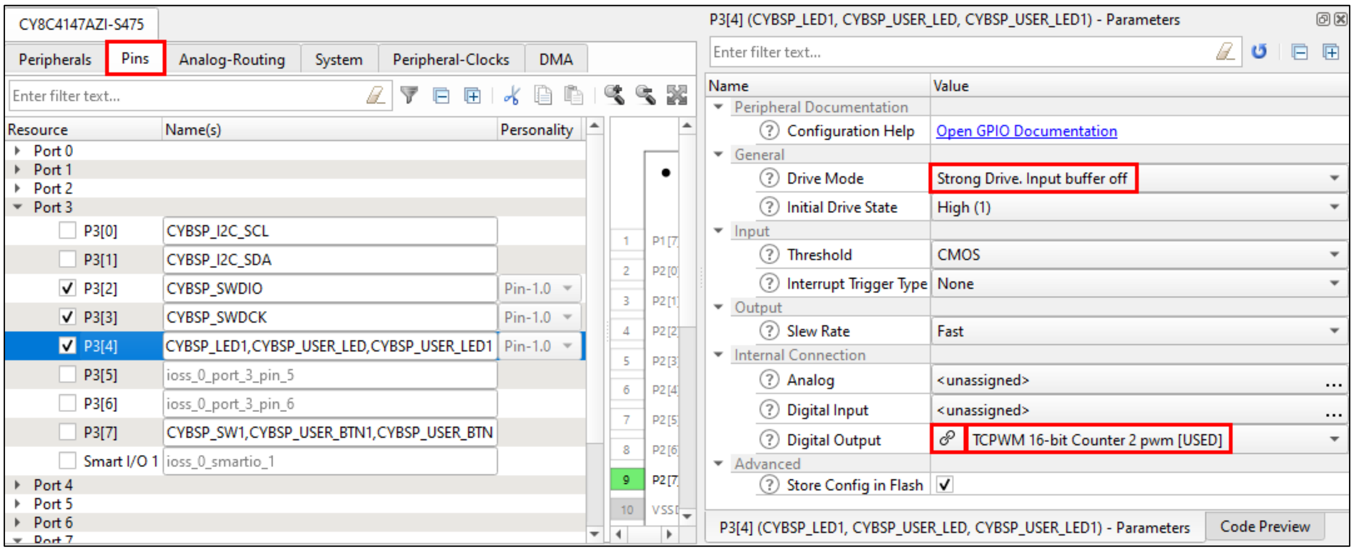
```
cyhal_pwm_t pwm_obj;
cyhal_pwm_init(&pwm_obj, CYBSP_USER_LED, NULL);
cyhal_pwm_start(&pwm_obj);
cyhal_pwm_set_duty_cycle(&pwm_obj, 50.0, 1);
```

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

2.4.2.2 PDL

If you are using the PDL, the first thing you need to do is figure out what pin you want the PWM to output on. Once you've got that, 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.

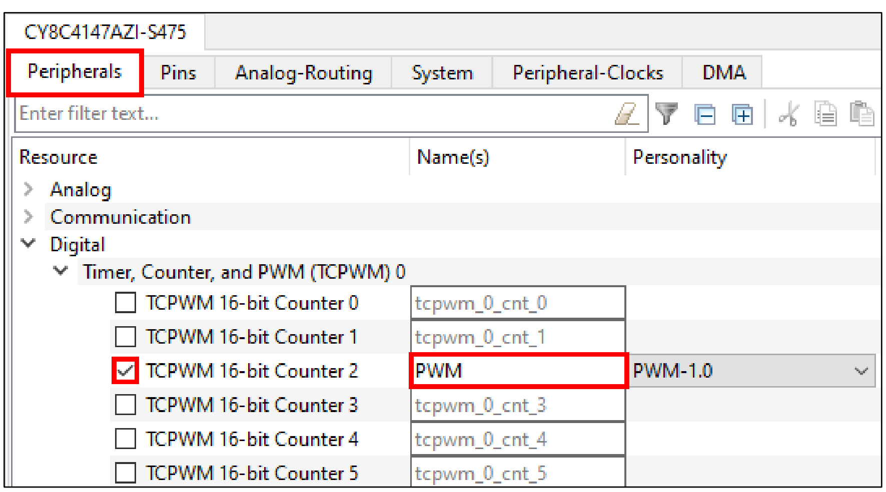
Note: If you don't select a valid drive mode for a PWM, you will see messages in the Notice List when you save the configuration.



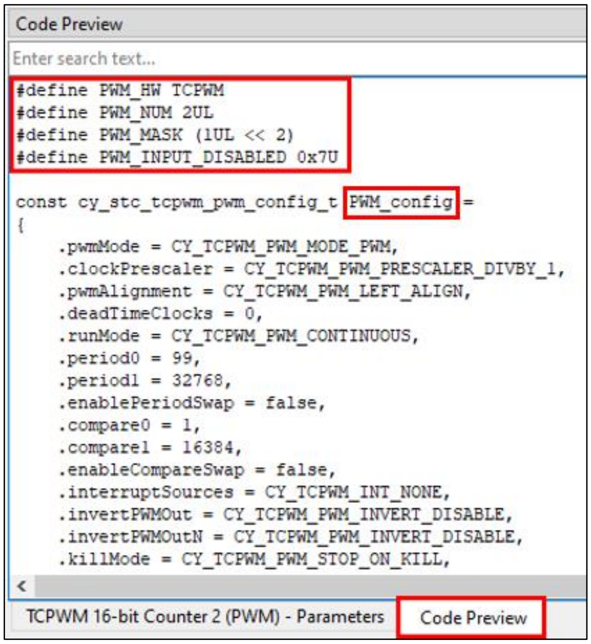
*Note: The **pwm_n** output is just the compliment of the **pwm** output.*

Then click on the "link" button  that appears in the **Digital Output** parameter.

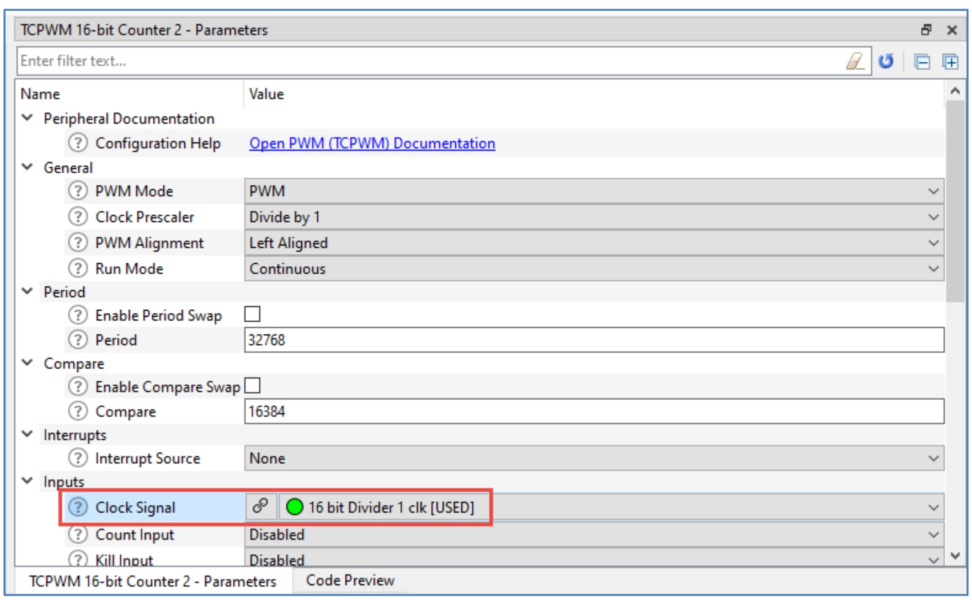
This will take you to **Peripherals** tab for the TCPWM you selected. Enable the counter you need by checking its box (it will appear with a clip-board next to its name), select **PWM-<version>** from the popup and click **OK**.



You can use the default name but it is a good idea 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:



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_TriggerReloadOrIndex`.

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

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

```
Cy_TCPWM_PWM_Init(PWM_HW, PWM_NUM, &PWM_config);
Cy_TCPWM_PWM_Enable(PWM_HW, PWM_NUM);
Cy_TCPWM_TriggerReloadOrIndex(PWM_HW, PWM_MASK);
```

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

2.4.2.3 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, if you want to do that you will need to use the PDL.

When using the HAL, you are required to connect the output of your PWM to a GPIO. When using the PDL this is not a requirement, the PWM output can be connected to a GPIO or directly to other hardware blocks.

2.4.2.4 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

2.4.3 ADC

You will use this in Exercise 11: and Exercise 12:

This section describes the SAR ADC that is available on most PSoC™ devices. Some devices have additional ADCs that can be used. For example, the CSD HW block on some devices has an ADC that is available for general use when it is not being used for CAPSENSE™ features. The CSDADC is not described here. For information on that ADC, add the CSDADC library to the application and view its documentation.

2.4.3.1 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.

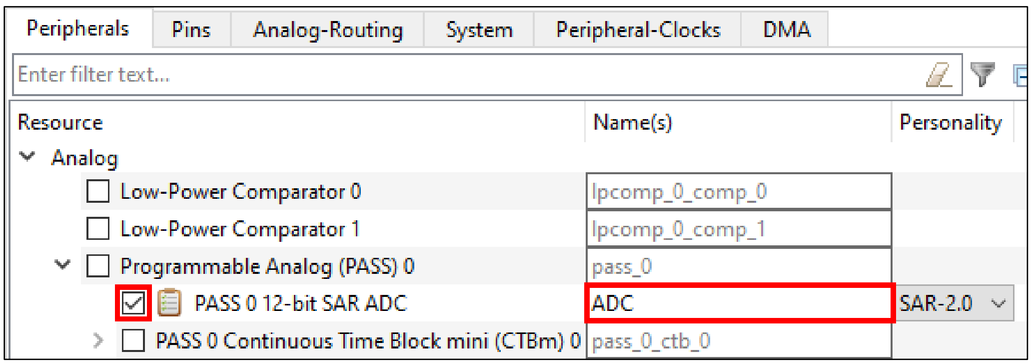
```
/* ADC block and channel objects */
cyhal_adc_t adc_obj;
cyhal_adc_channel_t adc_chan_0_obj;

/* 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 > ADC**.

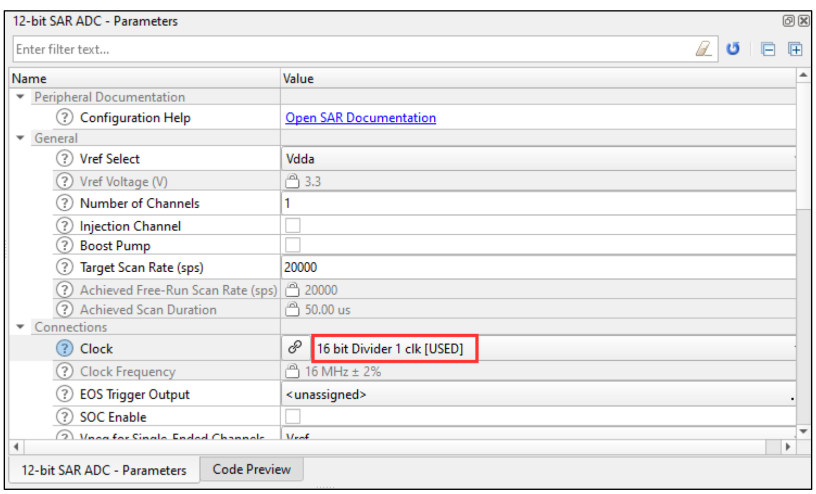
2.4.3.2 PDL

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



Give the ADC a sensible name like "ADC" and make a note of it to use in the code.

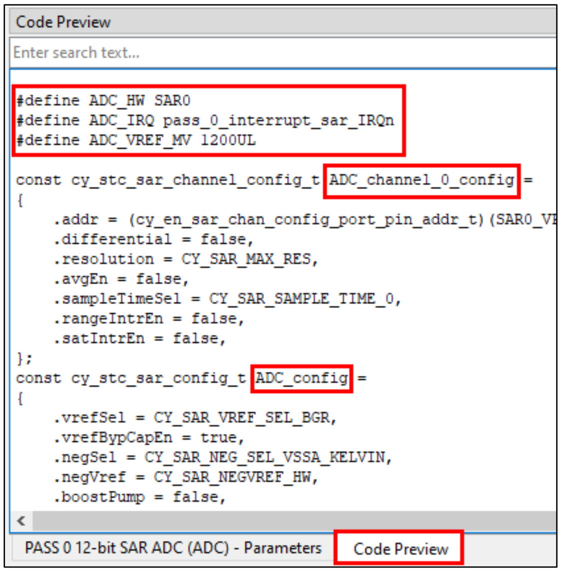
Then on the right side of the screen you can configure the ADC how you want. One parameter you need to change is **Clock**:



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 choose:



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

In the example shown above, the reference is selected as Vdda. 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_SAR_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_SAR_Enable` to enable your ADC.

To start a conversion, you can call the function `Cy_SAR_StartConvert`. You can then either configure your ADC to produce an interrupt when a conversion finishes, or you can use the function `Cy_SAR_IsEndConversion` to check if the conversion has finished from your firmware.

Once the conversion has finished, you can get the result by calling the function `Cy_SAR_GetResult16`. Which will return the conversion result in counts as a 16-bit integer.

Note: *There is also a 32-bit version of this function called* `Cy_SAR_GetResult32`.

Finally, to convert your ADC result into volts, the PDL provides several functions:

- `Cy_SAR_CountsTo_Volts` – Converts counts to Volts
- `Cy_SAR_CountsTo_mVolts` – Converts counts to miliVolts
- `Cy_SAR_CountsTo_uVolts` – Converts counts to microVolts

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.

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

/* Initialize and enable the ADC */
Cy_SAR_Init(ADC_HW, &ADC_config);
Cy_SAR_Enable(ADC_HW);

/* Start a single conversion */
Cy_SAR_StartConvert(ADC_HW, CY_SAR_START_CONVERT_SINGLE_SHOT);

/* Wait for the ADC to finish and then get the result */
if(Cy_SAR_IsEndConversion(ADC_HW, CY_SAR_WAIT_FOR_RESULT) == CY_SAR_SUCCESS)
{
      ADCresult = Cy_SAR_GetResult32(ADC_HW, 0);
      microVolts = Cy_SAR_CountsTo_uVolts(ADC_HW, 0, ADCresult);

}
```

Note: *Waiting for the conversion result is very inefficient – normally an interrupt would be used to read the result so that the CPU could do other useful work while the conversion is taking place.*

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

2.4.3.3 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.

2.4.3.4 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

2.4.4 UART

You will use this in Exercise 2: , Exercise 6: , Exercise 17: , Exercise 18: , Exercise 19: and Exercise 20:

UARTs are implemented using a Serial Communication Block called an SCB for short. Each SCB can be configured to implement UART, SPI, I^{2}C or EZI2C. In PSoC™ 4 and PSoC™ 6 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.

2.4.4.1 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. It is 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 PSoC™ 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 is useful if the serial terminal emulator you are using doesn't support that option and you don't want to use `\n\r` in the all of the `printf` statements. This is most 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 almost all 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™.

2.4.4.2 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 details on these functions 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 to other tasks instead of waiting.*

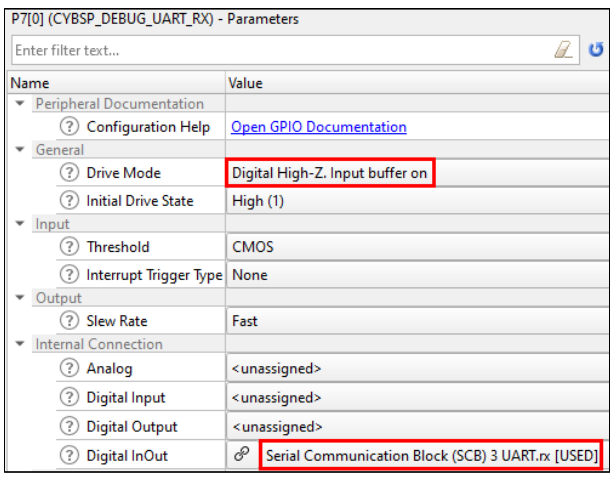
The documentation for the HAL UART functions can be found under **Hardware Abstraction Layer > HAL API Reference > HAL Drivers > ADC**.

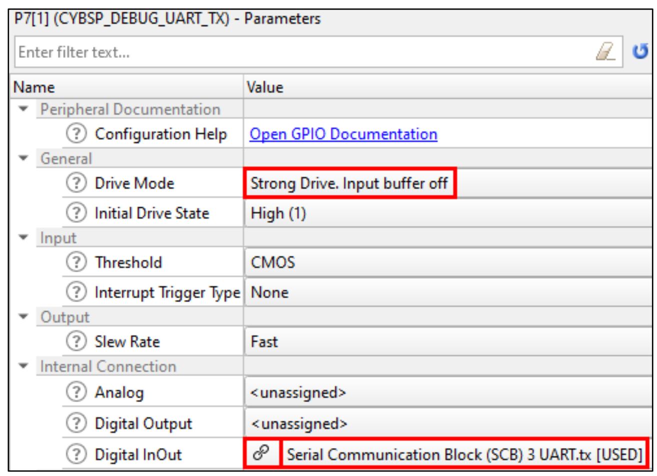
2.4.4.3 PDL

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

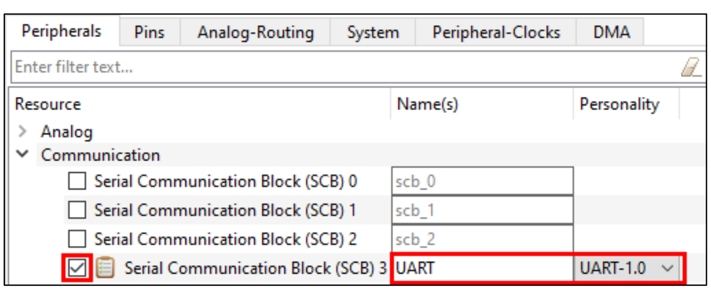
In the case of the CY8CKIT-149, the debug UART TX is connected to pin 7.1 and the RX is connected to pin 7.0.

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. Do 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.

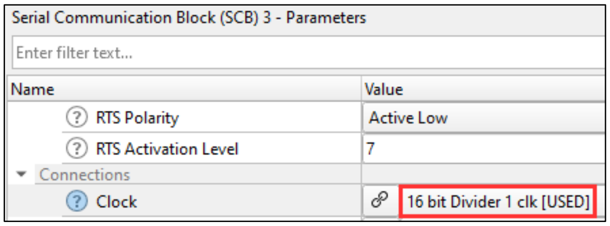




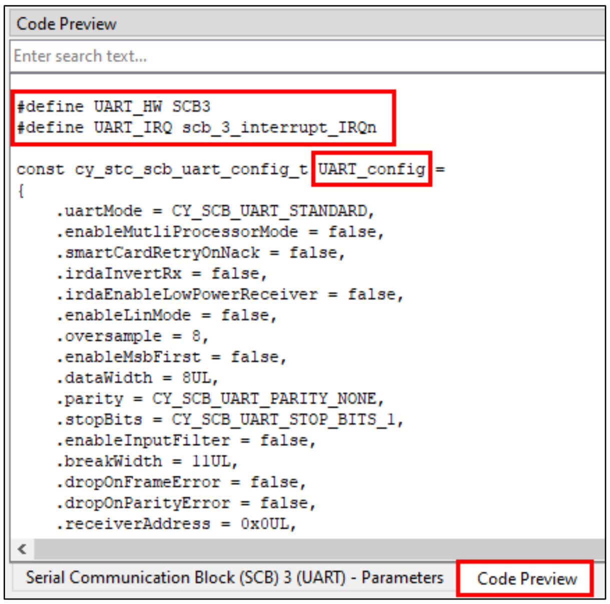
Click the "chain" button next to the **Digital InOut** connection on either pin. This will take you to **Peripherals** tab for the SCB you selected. Enable the SCB by checking its box and select **UART-<version>** from the popup menu and give it a name:



On the right side of the screen you can configure the UART 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:



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 generated 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`.

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**.

2.4.4.4 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 provides by default, you will need to use the PDL.

2.4.4.5 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

2.4.5 I²C

You will use this in Exercise 13: and Exercise 14:

I²C uses the same Serial Communication Hardware block as the UART. Again, in PSoC™ 4 and PSoC™ 6 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, the PSoC™ will be an I²C master while the sensor will be an I²C slave.

2.4.5.1 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 perform I2C reads and writes using a block of data at a specified memory address.

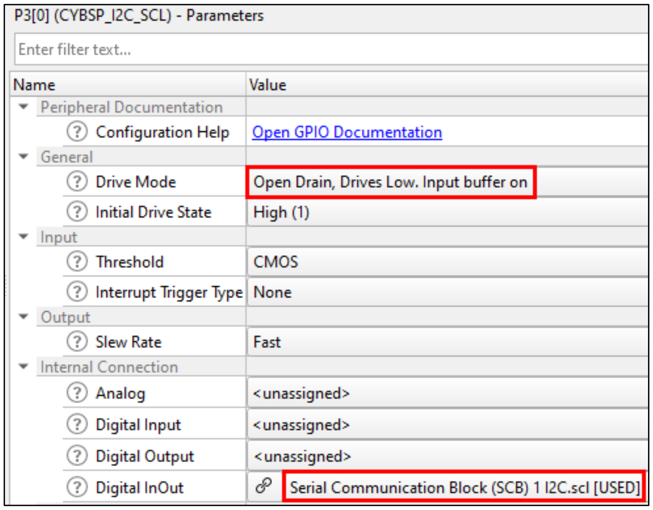
There is also a function called `cyhal_i2c_master_transfer_async` which can do a read, a write, or both.

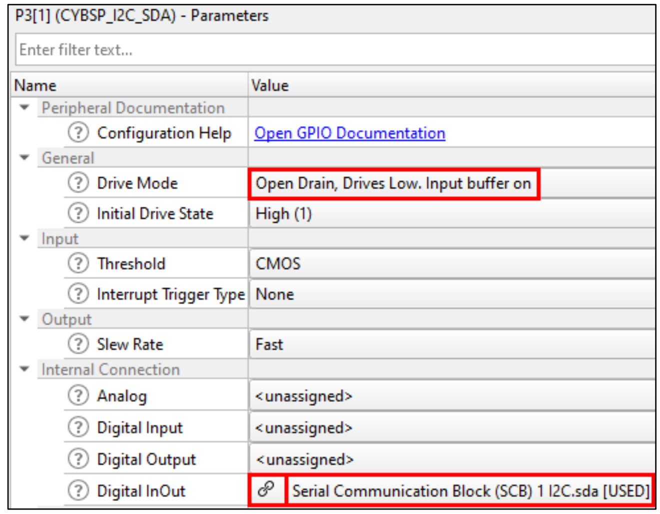
On the slave side, the functions to configure the slave's read and write buffers are: `cyhal_i2c_slave_config_read_buffer` and `cyhal_i2c_slave_config_read_buffer`.

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.

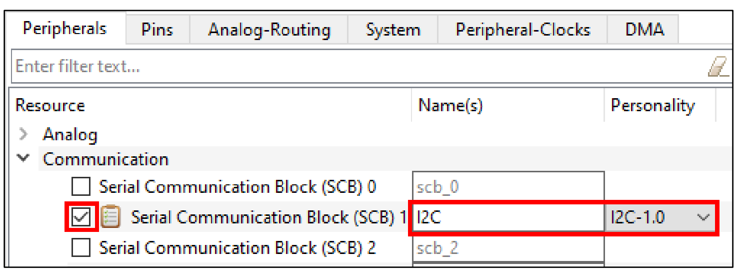
2.4.5.2 PDL

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** depending on whether external pull-up resistors are present on the board.

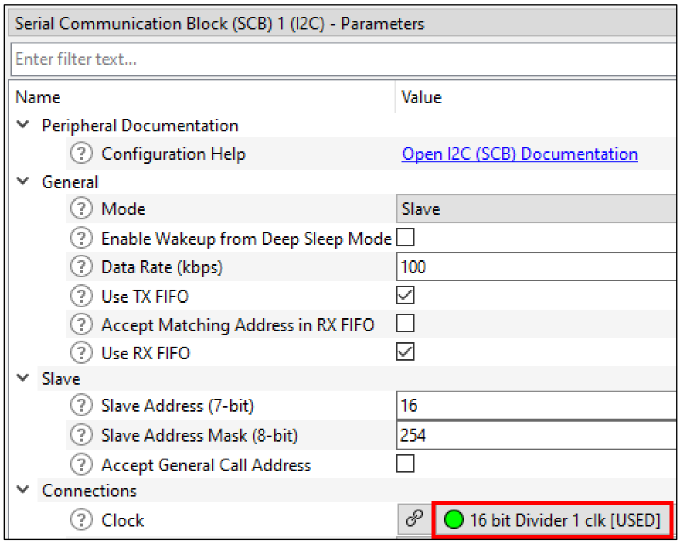




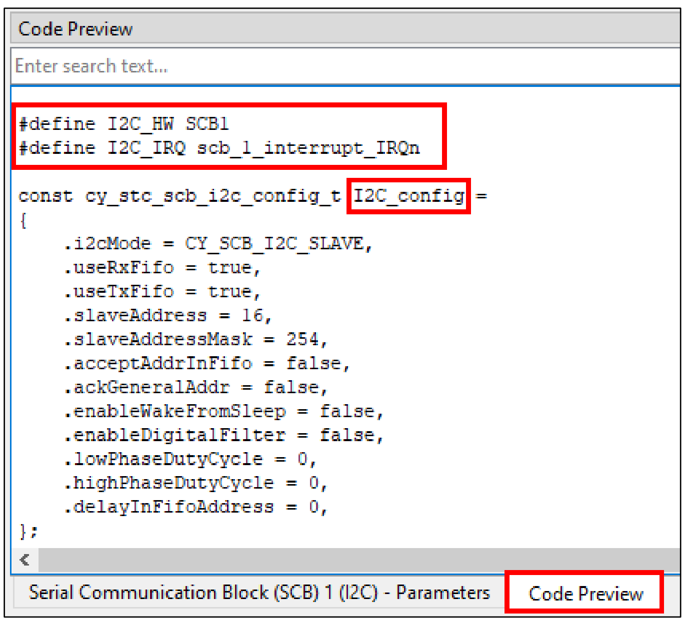
Once the pins are configured, click the "chain" button next to the **Digital InOut** connection on one of the pins. This will take you to **Peripherals** tab for the SCB you selected. Enable the SCB by checking its box, selecting **I2C-<version>** from the popup menu and giving it a name:



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 I^{2}C 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 I^{2}C.

There are several I^{2}C functions to send and receive data that you can read about in the documentation.

The documentation for the PDL I^{2}C functions can be found under **Peripheral Driver Library > PDL API Reference > SCB > I2C**. It includes several usage examples along with the full API description.

2.4.5.3 PDL vs. HAL

The HAL API has no way to directly configure the following I^2C parameters:

- Digital Filter
- SCL Low Phase
- SCL High Phase

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.

2.4.5.4 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 I^2C 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 I^2C error is detected

2.4.6 EZI2C Slave

EZI2C adds a protocol on top of I^2C slaves that allows a master to have random access to a block of memory on the EZI2C slave (a.k.a. a data buffer). The EZI2C component can be configured to have either 1 or 2 bytes of address offset (also called the sub-address). The default is 1 byte which means the data buffer can be up to 256 bytes. The first byte (or first two bytes if configured for a 2-byte offset) sent by the master in a write sequence is an offset which specifies which location in the buffer to start from. The offset will also be used in any following read sequences.

This protocol is very common and it (or one very similar to it) is used by most memory devices that are accessed using I^2C.

2.4.6.1 HAL

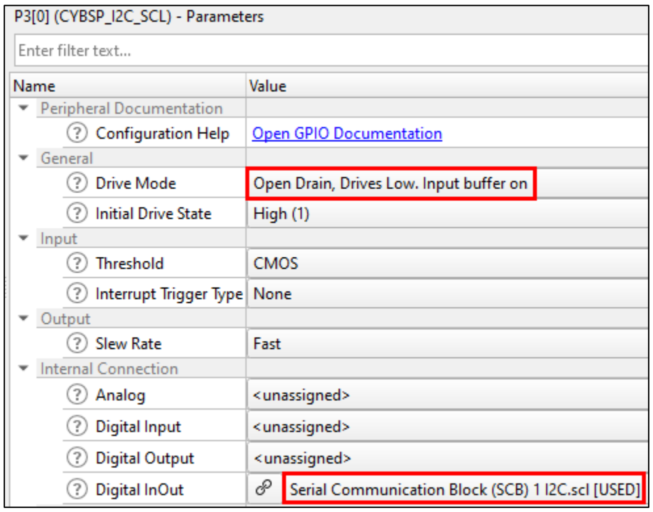
If you are using the HAL, the function to initialize an SCB block for EZI2C is `cyhal_ezi2c_init`. Among other things, this function takes a pointer to a configuration structure which allows configuration of the EZI2C slave. The configuration structure in turn points to a structure containing the lower level I2C slave configuration. The lower level configuration structure is where you specify the block of memory that will be accessible to the I2C master. See the API documentation for details and usage examples.

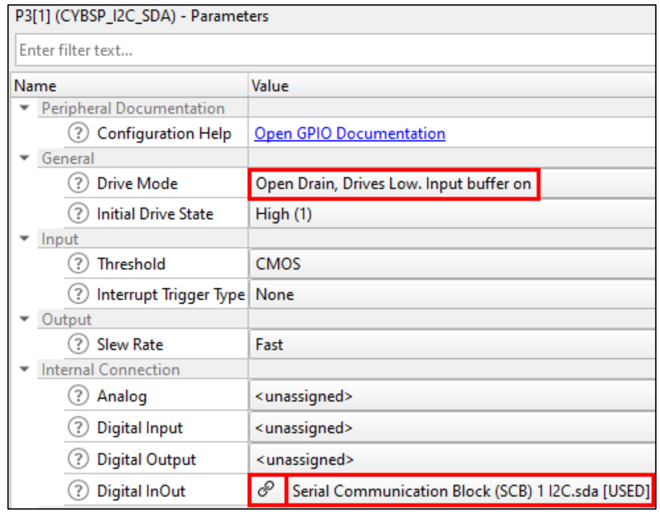
Alternately, you can use `cyhal_ezi2c_init_cfg` if you want to use the device configurator to set up the EZI2C block.

The documentation for the HAL I2C functions can be found under **Hardware Abstraction Layer > HAL API Reference > HAL Drivers > EZI2C**. It includes several usage examples along with the full API description.

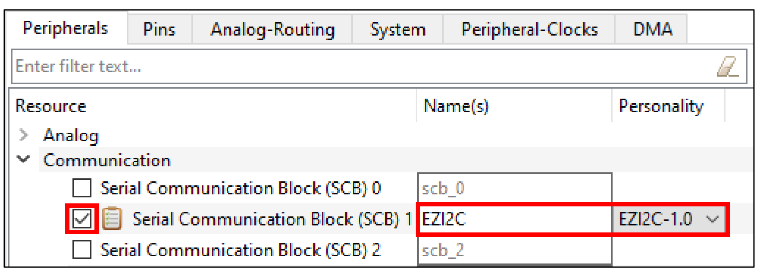
2.4.6.2 PDL

To initialize EZI2C using the PDL, open the Device Configurator and setup the pins exactly the same way as for I²C:

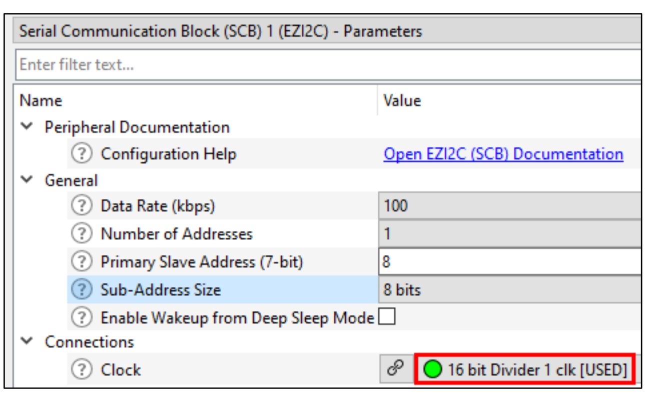




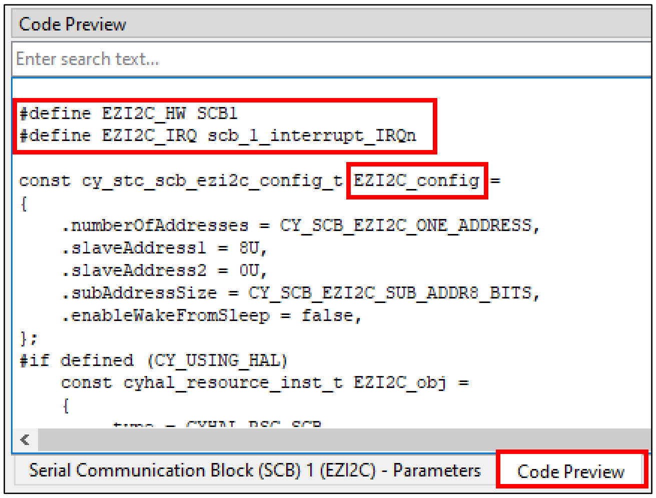
Then click on the "chain" button from either pin to go to the appropriate SCB on the **Peripherals** tab. Check the box of the **Serial Communication Block** that you want to enable and choose **EZI2C-<version>** from the popup menu. Give it a sensible name:



On the right side of the screen you can configure your EZI2C 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 EZI2C how you want it, do **File > Save**, and exit the Device Configurator.

In your application code you need to call the function `Cy_SCB_EZI2C_Init` to initialize your EZI2C. 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_EZI2C_Enable` to enable your EZI2C.

There are several EZI2C functions to send and receive data that you can read about in the documentation.

The documentation for the PDL EZI2C functions can be found under **Peripheral Driver Library > PDL API Reference > SCB > EZI2C**. In addition to the full API documentation, it includes several usage examples and a description of what common read and write operations look like.

2.4.6.3 PDL vs. HAL

The HAL API is able to directly configure all EZI2C slave parameters.

2.4.6.4 Interrupt Events

EZI2C slaves are able to trigger interrupts in the following scenarios:

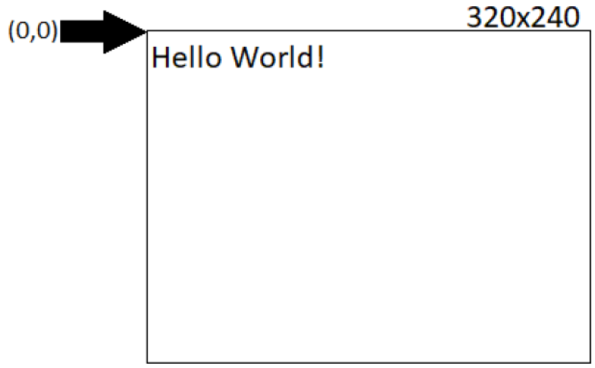
- OK – When an operation completed successfully
- Read1 – When the read transfer for the primary slave address is complete
- Write1 – When the write transfer for the primary slave address is complete
- Read2 – When the read transfer for the secondary slave address is complete
- Write2 – When the write transfer for the secondary slave address is complete
- Busy – When a transfer intended for the primary or secondary slave address is in progress
- Error – When an error occurred during a transfer for the primary or secondary slave address

2.4.7 TFT Display

This will be used in This will be used in Exercise 15: and Exercise 16:

The CY8CKIT-028-TFT shield contains a 320x240 pixel TFT screen with a Sitronix ST7789 driver chip.

The *emWin* library allows you to draw shapes and write text on the TFT screen. For example, `GUI_DispString("Hello World!")` will display text like this:



Note: The x and y coordinates start in the top left corner.

The steps to write text to the TFT screen on the CY8CKIT-028-TFT shield are:

1. Add the *emwin* and *CY8CKIT-028-TFT* libraries to your project via the Library Manager.

Note: The CY8CKIT-028-TFT library will include the following support libraries: display-tft-st7789v, sensor-light, sensor-motion-bmi160, audio-codec-ak4954a.

Note: Documentation for these libraries can be found in the mtb_shared/latest-vX.X/<lib>/docs directories or from the Quick Panel in Eclipse.

2. Enable the `EMWIN_NOSNTS` emWin library option by adding it to the *Makefile* `COMPONENTS` variable:

   ```
   COMPONENTS+=EMWIN_NOSNTS
   ```

3. Include *cy8ckit_028_tft.h* and call `cy8ckit_028_tft_init` to initialize the shield.

4. Include *GUI.h* and call the function `GUI_Init` with no arguments to initialize emWin's internal data structures and variables.

5. Each time you want to display a string you call `GUI_DispString`. This function takes the string you want to display as its only argument.

   For example, assuming the proper libraries have been included and the *Makefile* has been updated, the following *main.c* will print the string "PSoC":

   ```
   #include "cy_pdl.h"
   #include "cyhal.h"
   #include "cybsp.h"
   #include "GUI.h"
   #include "cy8ckit_028_tft.h"
   ```

```
int main(void){
    cy_rslt_t result;
    /* Initialize the device and board peripherals */
    result = cybsp_init();
    CY_ASSERT(result == CY_RSLT_SUCCESS);

    __enable_irq();

    /* Initialize the shield peripherals */
    result = cy8ckit_028_tft_init (NULL, NULL, NULL, NULL);

    GUI_Init();
    GUI_DispString("PSoC");

    for(;;){}
}
```

Note: *The code above initializes all of the shield's peripherals (e.g. display, light sensor and motion sensor) rather than just the TFT display. If you want to initialize just the TFT display you could do the following instead:*

a. Include *cy8ckit_028_tft_pins.h* and *mtb_st7789v.h*.

b. Set up a structure of type `mtb_st7789v_pins_t` to specify the shield's pins.

```
const mtb_st7789v_pins_t tft_pins =
{
    .db08 = CY8CKIT_028_TFT_PIN_DISPLAY_DB8,
    .db09 = CY8CKIT_028_TFT_PIN_DISPLAY_DB9,
    .db10 = CY8CKIT_028_TFT_PIN_DISPLAY_DB10,
    .db11 = CY8CKIT_028_TFT_PIN_DISPLAY_DB11,
    .db12 = CY8CKIT_028_TFT_PIN_DISPLAY_DB12,
    .db13 = CY8CKIT_028_TFT_PIN_DISPLAY_DB13,
    .db14 = CY8CKIT_028_TFT_PIN_DISPLAY_DB14,
    .db15 = CY8CKIT_028_TFT_PIN_DISPLAY_DB15,
    .nrd  = CY8CKIT_028_TFT_PIN_DISPLAY_NRD,
    .nwr  = CY8CKIT_028_TFT_PIN_DISPLAY_NWR,
    .dc   = CY8CKIT_028_TFT_PIN_DISPLAY_DC,
    .rst  = CY8CKIT_028_TFT_PIN_DISPLAY_RST
};
```

c. Instead of calling `cy8ckit_028_tft_init`, initialize just the display by calling the function `mtb_st7789v_init8(&tft_pins)`.

2.5 Interrupts

You will use this in Exercise 4: and Exercise 8:

Interrupts are an event-triggered means 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 of the PSoC™ 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.

2.5.1 Global Interrupt Enable

If you want to make use of interrupts in your application it is important to first 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.

2.5.2 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:

2.5.2.1 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 */

    }

}
```

2.5.2.2 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 has 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);
}
```

2.5.3 PDL

The documentation for the PDL interrupt API can be found under **CAT2 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.

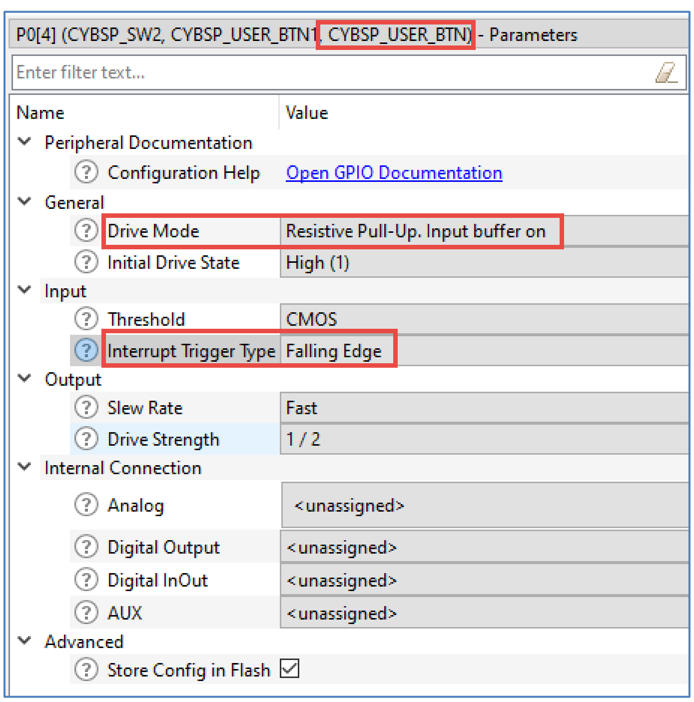
The details vary slightly between peripherals, so you should refer to the documentation, but generally to use a PDL interrupt the procedure is:

1. Use the Device Configurator to setup the peripheral and to select its interrupt source(s).
    a. 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.
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`.
    a. The NVIC is an Arm® Cortex® hardware block that (among other things) maps interrupts from each of the interrupt sources into the CPU.
6. Use PDL functions to initialize and start the peripheral as usual (if necessary).

GPIO interrupt and PWM interrupt examples are shown here. These examples do the same thing as the HAL examples shown above.

2.5.3.1 PDL GPIO Interrupt

The code snippet below shows a PDL GPIO interrupt for falling edges on an input pin (`CY_GPIO_INTR_FALLING`). This is useful for cases such as a mechanical button that pulls the pin low when pressed. The Device Configurator was used to configure the pin and its interrupts and its name is specified as `CYBSP_USER_BTN` as shown here:

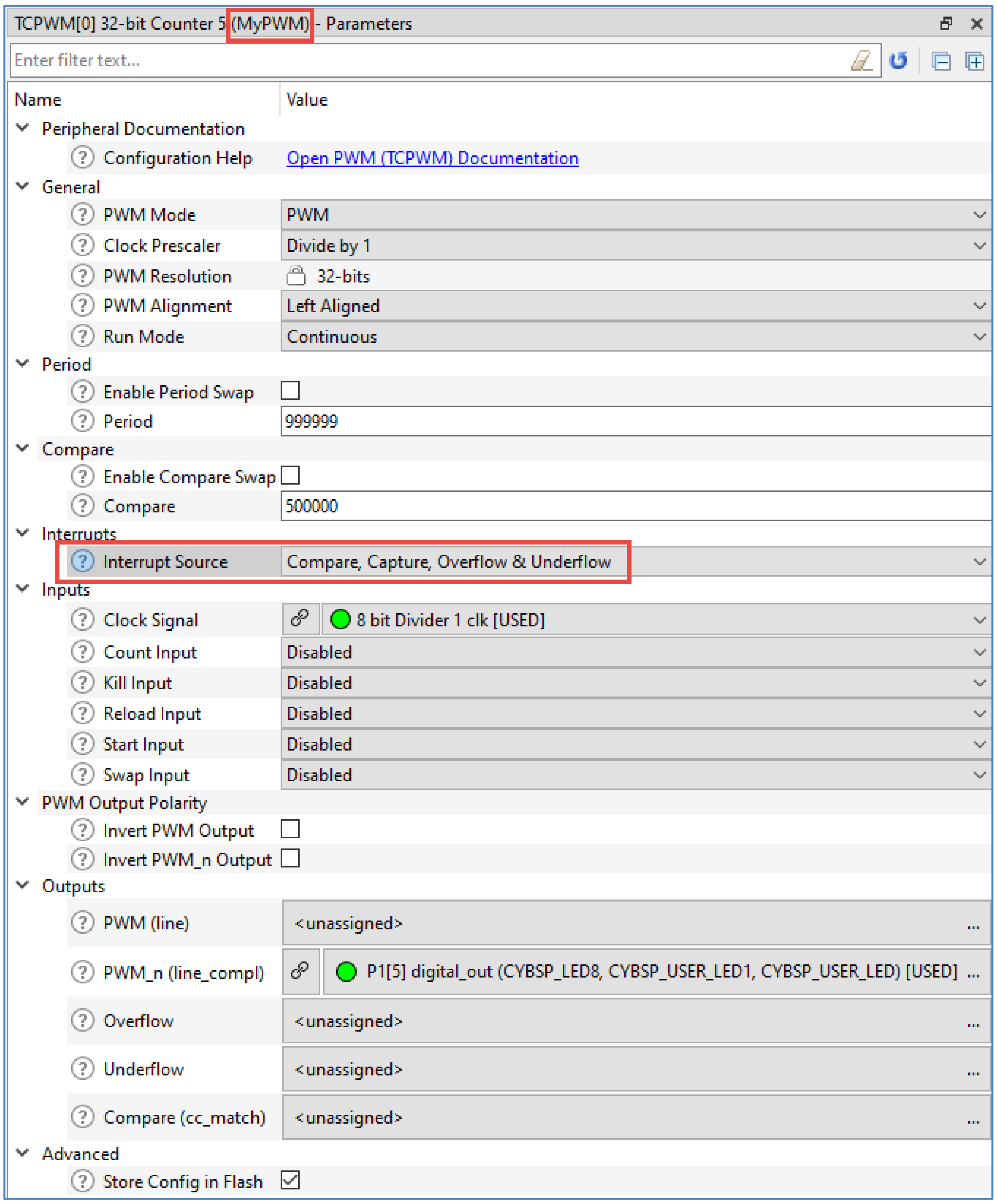


```
#define GPIO_INTERRUPT_PRIORITY (7u)
#define PORT_INTR_MASK (0x00000001UL << CYBSP_USER_BTN_PORT_NUM)

/* 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 =*/ 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(intrCfg.intrSrc);

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

2.5.3.2 PDL PWM Interrupt

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

The Device Configurator was used to configure the PWM and the associated pin and clock correctly and that the PWM's name is specified as `MyPWM`. In addition, the Device Configurator was 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 =*/ 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(intrCfg.intrSrc);

    /* 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_TriggerReloadOrIndex(MyPWM_HW, MyPWM_MASK);

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

2.6 Exercises

Each exercise in this section uses either the HAL or PDL to drive peripherals. The HAL exercises use the PSoC™ 6 CY8CKIT-062S2-43012 kit while most of the PDL exercises use both the PSoC™ 4 CY8CKIT-149 kit and the PSoC™ 6 kit. Some of the PDL exercises use only the PSoC™ 6 kit due to required board resources. Depending on your needs, you can pick and choose which exercises to try. You should quickly see the ease-of-use advantages of using the HAL over the PDL. In fact, if you are using the PSoC™ 6 kit, you may want to focus only on the HAL exercises first and then come back to PDL later should you want to learn about advanced functionality.

Note: The TFT display exercise uses the CY8CKIT-062S2-43012 with an additional CY8CKIT_028-TFT shield.

Exercise 1: (GPIO-HAL) Blink an LED

This exercise uses the CY8CKIT-062S2-43012. This material is covered in 2.4.1.

☐ 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.

Exercise 2: (GPIO-HAL) Add debug printing to the LED blink project

This exercise uses the CY8CKIT-062S2-43012. This material is covered in 2.4.4.

☐ 1. Use Project Creator to create a new application called **ch03_ex02_HAL_blinkled_print** using the **Browse** button on the **Select Application** page to select your previous exercise (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 or else you can 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 3: (GPIO-HAL) Read the state of a mechanical button

This exercise uses the CY8CKIT-062S2-43012. This material is covered in 2.4.1.

☐ 1. Use Project Creator to create a new application called **ch02_ex03_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 re-start the firmware execution from the beginning.

Exercise 4: (GPIO-HAL) Use an interrupt to toggle the state of an LED

This exercise uses the CY8CKIT-062S2-43012. This material is covered in 2.5.

☐ 1. Use Project Creator to create a new application called **ch02_ex04_HAL_interrupt** using the **Browse** button on the **Select Application** page to 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 what 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`.

- Build the project so that Intellisense will work properly. Then 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.

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

Note: You can use the function `cyhal_gpio_toggle`.

5. Program your project to your kit and press the button to observe the behavior.

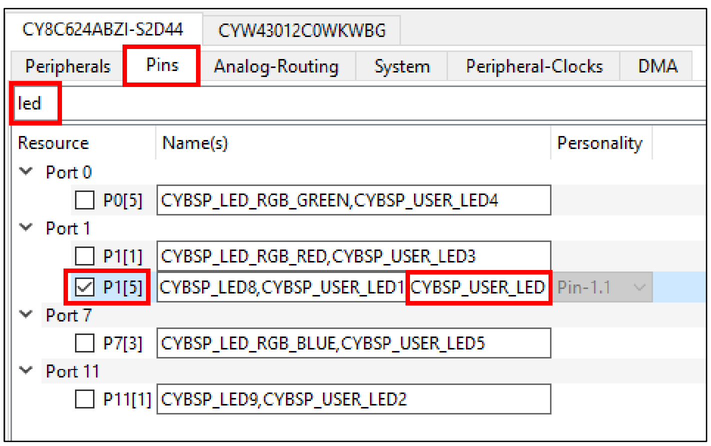
Exercise 5: (GPIO-PDL) Blink an LED

This exercise uses either the CY8CKIT-149 or the CY8CKIT-062S2-43012. This material is covered in 2.4.1.

1. Use Project Creator to create a new application called **ch02_ex05_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":



Note: The above screenshot is from the PSoC™ 6 kit. If you are using the PSOC™ 4 kit, the pin you enable will be different.

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.

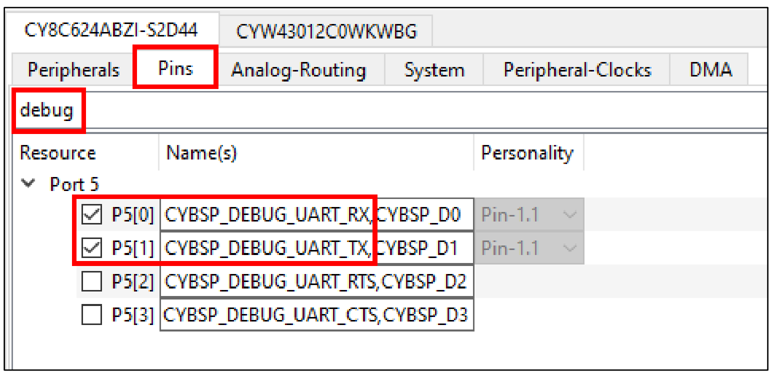
Exercise 6: (GPIO-PDL) Add debug printing to the LED blink project

This exercise uses either the CY8CKIT-149 or the CY8CKIT-062S2-43012. This material is covered in 2.4.4.

☐ 1. Use Project Creator to create a new application called **ch02_ex06_PDL_blinkled_print** using the **Browse** button on the **Select Application** page to select your previous exercise (ch02_ex05_PDL_blinkled) as a template.

☐ 2. Use the Device Configurator to enable the debug UART.

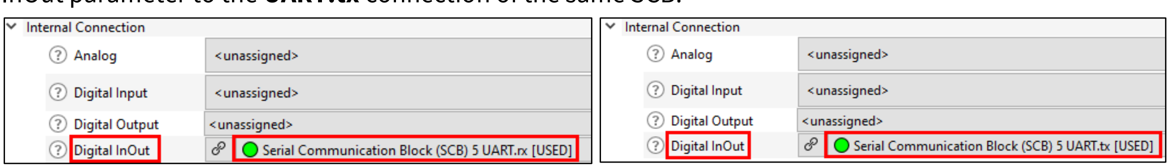
To do this, navigate to the **Pins** Tab and enter the filter text "debug". Enable the pins that have the names "CYBSP_DEBUG_UART_RX" and "CYBSP_DEBUG_UART_TX":



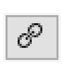
Note: The above screenshot is from the PSoC™ 6 kit. If you are using the PSOC™ 4 kit, the pins you enable will be different.

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:



Note: The above screenshots are from the PSoC™ 6 kit. If you are using the PSOC™ 4 kit, the SCB you connect will be different.

Click the "chain" button  next to the Digital InOut parameter you just selected. This will take you to the 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 fun`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.*

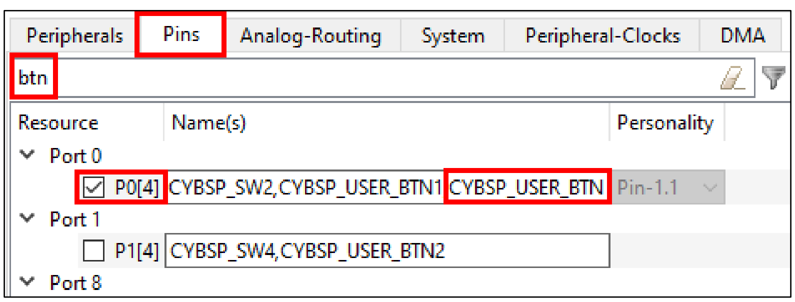
Exercise 7: (GPIO-PDL) Read the state of a mechanical button

This exercise uses either the CY8CKIT-149 or the CY8CKIT-062S2-43012. This material is covered in 2.4.1.

1. Use Project Creator to create a new application called **ch02_ex07_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 (**Restive Pull-Up. Input buffer on**) and 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 that has the name "CYBSP_USER_BTN":



Note: *The above screenshot is from the PSoC™ 6 kit. If you are using the PSoC™ 4 kit, the pin you enable will be different.*

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

This exercise uses either the CY8CKIT-149 or the CY8CKIT-062S2-43012. This material is covered in 2.5.

☐ 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 previous exercise (ch02_ex07_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/CAT2 Peripheral Driver Library > PDL API Reference > SysInt** documentation for how to set up interrupts.*

Note: *See the **CAT1/CAT2 Peripheral Driver Library > PDL API Reference > GPIO > Functions > Port Interrupt Functions** for how to enable falling edge interrupts and how to clear 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 `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_GetInterruptCause` if you're using the PSoC™ 4 kit, or the function `Cy_GPIO_GetInterruptCause0` if you're using the PSoC™ 6 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 CY8CKIT-062S2-43012. This material is covered in 2.4.2.

☐ 1. Use 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 is fast enough so that even when divided by 100 it is faster than a human eye can see so that 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.

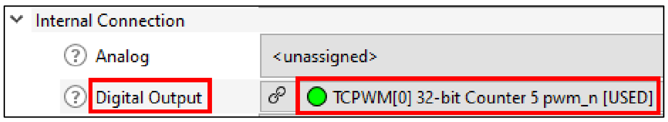
Exercise 10: (PWM-PDL) LED Brightness

This exercise uses either the CY8CKIT-149 or the CY8CKIT-062S2-43012. This material is covered in 2.4.2.

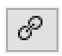
☐ 1. Use Project Creator to create a new application called **ch02_ex10_PDL_pwm** using **Empty App** as the template.

☐ 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:



Note: The above screenshot is from the PSoC™ 6 kit. If you are using the PSOC™ 4 kit, the TCPWM you connect will be different.

Click the "chain" button  next to the Digital Output parameter you just selected. This will take you to the page where you can configure the TCPWM you just connected.

☐ 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 is fast enough so that even when divided by 100 it is faster than a human eye can see so that 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_TriggerReloadOrIndex` 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 CY8CKIT-062S2-43012. This material is covered in 2.4.3.

☐ 1. Use Project Creator to create a new application called **ch02_ex11_HAL_adcread** 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 sticker on the back of the kit.*

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.

Note: *The default ADC VREF is 1.2V, so the maximum you will be able to read is 2.4V. Other VREF selections (and other settings) are available by using `cyhal_adc_configure`.*

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

This exercise uses the CY8CKIT-062S2-43012, as the CY8CKIT-149 does not have any built-in hardware on the kit that can generate an analog voltage such as a potentiometer. This material is covered in 2.4.3.

☐ 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".
- Set the SAR ADC parameter **Ch0 Vplus** to the pin that the POT is connected to. You can find this pin number by looking at the sticker on the back of your kit.

Note: *To take advantage of the full range of the potentiometer, set the SAR ADC parameters **Vref** and **Vneg for Single-Ended Channels** to "Vdda" and "Vref" respectively.*

Note: *On the CY8CKIT-062S2-43012 kit you also need to enable the AREF Programmable Analog Peripheral to use the ADC. Change its name of the block to AREF to simplify the name needed for the API function calls. Don't forget to initialize and enable this block by calling the functions:*

- `Cy_SysAnalog_Init`
- `Cy_SysAnalog_Enable`

☐ 5. Update 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: You can use the function `Cy_SAR_CountsTo_uVolts` to get a result in microvolts.

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

This exercise uses the CY8CKIT-062S2-43012. This material is covered in 2.4.5.

☐ 1. Use Project Creator to create a new application called **ch02_ex13_HAL_i2cread** using the **Empty App** as the template.

☐ 2. Use the Library Manager to add the CY8CKIT-028-TFT library (to get access to the motion 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 acceleration and gyroscope data are read from the I²C slave.

Note: Look at the documentation in the BMI160 Motion Sensor library to see an example of how to read the data. Don't forget 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 acceleration and gyroscope values to the terminal using `printf`.

☐ 6. Make sure the TFT shield is plugged in (that's where the motion sensor is), program your kit and observe the results on the UART when you move and turn the kit.

Exercise 14: (I2C READ-PDL) Read sensor values over I²C

This exercise uses the CY8CKIT-062S2-43012 and the CY8CKIT-028-TFT. This material is covered in 2.4.5.

☐ 1. Use Project Creator to create a new application called **ch02_ex14_PDL_i2cread** using the **Empty App** as the template.

☐ 2. Use the Library Manager to add the bmi160 library (to get access to the motion sensor).

Note: The Infineon sensor-motion-bmi160 library makes use of the HAL, so instead of using it, we will use the lower level bmi160 library provided by Bosch.

☐ 3. The bmi160 library contains a directory called *examples* that we need to exclude from the build. To do this, add the following line to the *Makefile*:

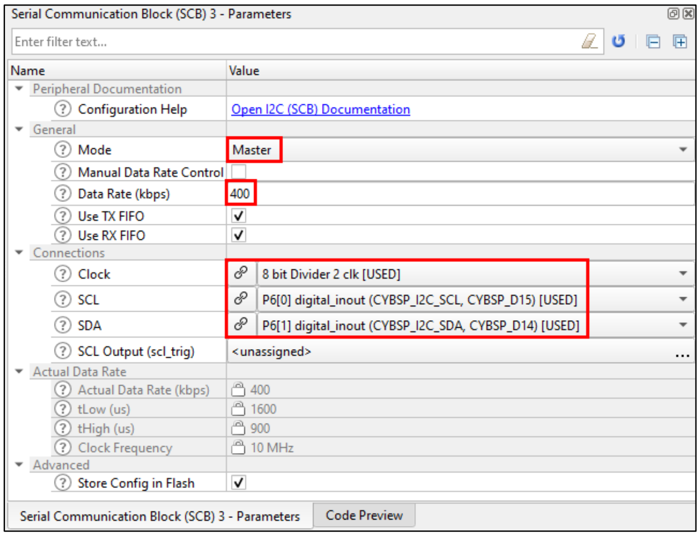
```
CY_IGNORE = $(SEARCH_BMI160_driver)/examples
```

Note: The search path variable for the BMI160 driver library is defined in the file libs/mtb.mk. That file automatically contains the path for all libraries included in the application.

☐ 4. Use the Device Configurator to enable the debug UART.

5. Use the Device Configurator to enable/configure the pins and the SCB block that connects to the appropriate pins. Select the I²C personality for the SCB block and use an easy to remember name for the such as I²C.

- Set the **Mode** to **Master**
- Set the **Data Rate** to 400 kbps
- Connect the I²C block to your kits SCL and SDA pins
- Don't forget to choose a **Clock**



6. Replace the code in *main.c* with the following:

```
#include "cy_pdl.h"
#include "cybsp.h"
#include "bmi160.h"
#include "stdio.h"

#define I2C_WRITE_BUFFER_LENGTH   32

// I2C context variable
cy_stc_scb_i2c_context_t I2C_context;

/*
 * I2C ISR
 */
void I2C_Isr(void){
    Cy_SCB_I2C_MasterInterrupt(SCB3, &I2C_context);
}

/*
 * Motion sensor I2C write function
 */
static int8_t i2c_write_bytes(uint8_t dev_addr, uint8_t reg_addr, uint8_t *data, uint16_t len){
  // Create write buffer
    CY_ASSERT(len + 1 < I2C_WRITE_BUFFER_LENGTH);
    uint8_t buf[I2C_WRITE_BUFFER_LENGTH];
    buf[0] = reg_addr;
    for(uint16_t i=0; i<len; i++){
        buf[i+1] = data[i];
    }

    // I2C transfer configuration structure
    cy_stc_scb_i2c_master_xfer_config_t xferConfig = {
             .slaveAddress = dev_addr,
             .buffer = buf,
             .bufferSize = len + 1,
             .xferPending = false
  };

    // Write
    cy_en_scb_i2c_status_t result = Cy_SCB_I2C_MasterWrite(SCB3, &xferConfig, &I2C_context);
    // Wait for write to complete
  while (0UL != (CY_SCB_I2C_MASTER_BUSY & Cy_SCB_I2C_MasterGetStatus(SCB3, &I2C_context))){}

  // Return the result of the write
    return (CY_SCB_I2C_SUCCESS == result) ? BMI160_OK : BMI160_E_COM_FAIL;
}

/*
```

```
 * Motion sensor I2C read function
 */
static int8_t i2c_read_bytes(uint8_t dev_addr, uint8_t reg_addr, uint8_t *data, uint16_t len){
  // I2C transfer configuration structure
  cy_stc_scb_i2c_master_xfer_config_t xferConfig = {
              .slaveAddress = dev_addr,
              .buffer = &reg_addr,
              .bufferSize = 1,
              .xferPending = true
  };

  // Write
  cy_en_scb_i2c_status_t result = Cy_SCB_I2C_MasterWrite(SCB3, &xferConfig, &I2C_context);
  // Wait for write to complete
  while (0UL != (CY_SCB_I2C_MASTER_BUSY & Cy_SCB_I2C_MasterGetStatus(SCB3, &I2C_context))){}

  // If write was successful we can now read
    if (CY_SCB_I2C_SUCCESS == result){
              // Reconfigure transfer configuration structure for read
              xferConfig.buffer = data;
              xferConfig.bufferSize = len;
              xferConfig.xferPending = false;

              // Read
              result = Cy_SCB_I2C_MasterRead(SCB3, &xferConfig, &I2C_context);
              // Wait for read to complete
              while (0UL != (CY_SCB_I2C_MASTER_BUSY & Cy_SCB_I2C_MasterGetStatus(SCB3, &I2C_context))){}
    }

    // Return the result of the read
    return (CY_SCB_I2C_SUCCESS == result) ? BMI160_OK : BMI160_E_COM_FAIL;
}

/*
 * Motion sensor delay function
 */
static void delay_wrapper(uint32_t ms){
    (void)Cy_SysLib_Delay(ms);
}

int main(void)
{
    cy_rslt_t result;

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


    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    __enable_irq();

    // UART context variable
  cy_stc_scb_uart_context_t UART_context;

    // Configure and enable the UART peripheral
  Cy_SCB_UART_Init(SCB5, &scb_5_config, &UART_context);
  Cy_SCB_UART_Enable(SCB5);

  // Initialize and enable the I2C peripheral
  Cy_SCB_I2C_Init(SCB3, &scb_3_config, &I2C_context);

  // I2C interrupt config - Interrupt is needed to use the PDL's I2C Master High-level functions
  const cy_stc_sysint_t i2cIntrConfig =
  {
              .intrSrc      = scb_3_IRQ,
              .intrPriority = 7UL,
  };
  Cy_SysInt_Init(&i2cIntrConfig, &I2C_Isr);
  NVIC_EnableIRQ(scb_3_IRQ);
  Cy_SCB_I2C_Enable(SCB3);

  // Motion sensor device configuration struct
 struct bmi160_dev motionSensor = {
                    .id        = BMI160_I2C_ADDR,
                    .intf      = BMI160_I2C_INTF,
                    .read      = (bmi160_read_fptr_t)i2c_read_bytes,
                    .write     = (bmi160_write_fptr_t)i2c_write_bytes,
                    .delay_ms  = delay_wrapper
  };

   // Intialize and configure the motion sensor
   int8_t status = bmi160_init(&motionSensor);
   if(status != BMI160_OK){
              Cy_SCB_UART_PutString(SCB5, "Motion Sensor Initialization Failed!\n");
              CY_ASSERT(0);
    }
    // Select the Output data rate, range of accelerometer sensor
  motionSensor.accel_cfg.odr = BMI160_ACCEL_ODR_1600HZ;
  motionSensor.accel_cfg.range = BMI160_ACCEL_RANGE_2G;
  motionSensor.accel_cfg.bw = BMI160_ACCEL_BW_NORMAL_AVG4;

  // Select the power mode of accelerometer sensor
```

```
  motionSensor.accel_cfg.power = BMI160_ACCEL_NORMAL_MODE;

  // Select the Output data rate, range of gyroscope sensor
  motionSensor.gyro_cfg.odr = BMI160_GYRO_ODR_3200HZ;
  motionSensor.gyro_cfg.range = BMI160_GYRO_RANGE_2000_DPS;
  motionSensor.gyro_cfg.bw = BMI160_GYRO_BW_NORMAL_MODE;

  // Gyroscope power mode
  motionSensor.gyro_cfg.power = BMI160_GYRO_NORMAL_MODE;

  // Set the sensor configuration
  status = bmi160_set_sens_conf(&motionSensor);
  if(status != BMI160_OK){
             Cy_SCB_UART_PutString(SCB5, "Motion Sensor Configuration Failed!\n");
             CY_ASSERT(0);
  }

  // Vars to hold motion data
  struct bmi160_sensor_data accel; // Accelerometer data
  struct bmi160_sensor_data gyro; // Gyroscope data

    for (;;){
             // Read data from the motion sensor and print it to the Debug UART every 100ms
             status = bmi160_get_sensor_data((BMI160_ACCEL_SEL | BMI160_GYRO_SEL | BMI160_TIME_SEL), &accel, &gyro,
&motionSensor);
             if(status != BMI160_OK){
                        Cy_SCB_UART_PutString(SCB5, "Motion Sensor Read Failed!\n");
                        CY_ASSERT(0);
             }
             char printBuffer[50];
             sprintf(printBuffer, "Accel: X:%6d Y:%6d Z:%6d\r\n", accel.x, accel.y, accel.z);
             Cy_SCB_UART_PutString(SCB5, printBuffer);
             sprintf(printBuffer, "Gyro : X:%6d Y:%6d Z:%6d\r\n\r\n", gyro.x, gyro.y, gyro.z);
             Cy_SCB_UART_PutString(SCB5, printBuffer);
             Cy_SysLib_Delay(100);
    }
}

/* [] END OF FILE */
```

Note: *If you entered a custom name for either the UART or I²C in the device configurator be sure to update the init functions in the code to use your custom name(s).*

☐ 7. Read through the code and make sure you understand what is going on.

☐ 8. Make sure the TFT shield is plugged in (that's where the motion sensor is), program your kit and observe the results on the UART when you move and turn the kit.

Exercise 15: Install shield support libraries and use TFT display

This exercise uses the CY8CKIT-062S2-43012 and the CY8CKIT-028-TFT. This material is covered in 2.1 and 2.4.7.

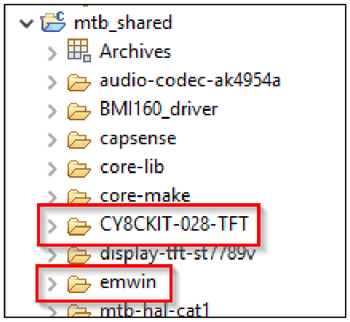
☐ 1. Install the libraries for the appropriate kit/shield combination into your SDK workspace.

   a. 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_ex15_tft**.

   b. Launch the Library Manager tool and add the *CY8CKIT-028-TFT* and *emWin* libraries.

   c. Update the *Makefile* `COMPONENTS` variable to read `COMPONENTS+=EMWIN_NOSNTS`

☐ 2. 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.



☐ 3. Replace the code in *main.c* with the following:

```
#include "cyhal.h"
#include "cybsp.h"
#include "GUI.h"
#include "cy8ckit_028_tft.h"

int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    __enable_irq();

    /* Initialize the CY8CKIT_028_TFT board */
    cy8ckit_028_tft_init (NULL, NULL, NULL, NULL);

    GUI_Init();
    GUI_SetColor(GUI_WHITE);
    GUI_SetBkColor(GUI_BLACK);
    GUI_SetFont(GUI_FONT_32B_1);
    GUI_SetTextAlign(GUI_TA_CENTER);
    /* Change this text as appropriate */
    GUI_DispStringAt("I feel good!", GUI_GetScreenSizeX()/2,
              GUI_GetScreenSizeY()/2 - GUI_GetFontSizeY()/2);

    for(;;){}
}
```

☐ 4. Program your project to your kit and observe the TFT display.

Exercise 16: Display sensor information on the TFT display

This exercise uses the CY8CKIT-062S2-43012 and the CY8CKIT-028-TFT. This material is covered in 2.1 and 2.4.7.

☐ 1. Use Project Creator to create a new application called **ch02_ex16_sensorData** using the **Browse** button to select your previous exercise (ch02_ex15_tft) as a template.

☐ 2. Add the retarget-io library.

☐ 3. Add the required includes for the new library.

☐ 4. Update the code so that the ambient light, acceleration, and gyro values are read from the shield and displayed to the screen every ½ second.

Note: *You will need to call* `cy8ckit_028_tft_get_light_sensor` *and* `cy8ckit_028_tft_get_motion_sensor` *to get access to the objects that are used to report the data from the ADC (for the light sensor) and I2C (for the motion sensor). These objects are setup when the shield library is initialized.*

Note: *Look at the documentation for the sensor-light and sensor-motion-bmi160 libraries for examples of how to read sensor data.*

Note: *If you don't want to use all of the shield resources, you can initialize them individually (e.g. just the TFT, just the light sensor or just the motion sensor). See the individual library documentation if you require that use case.*

☐ 5. Program your project to your kit.

☐ 6. Shine light on the light sensor and move the kit to see the values update.

Exercise 17: (UART-HAL) Read a value using the standard UART functions

This exercise uses the CY8CKIT-062S2-43012. This material is covered in 2.4.4.

☐ 1. Use Project Creator to create a new application called **ch02_ex17_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 and observe the LED turn on/off.

Exercise 18: (UART-HAL) Write a value using the standard UART functions

This exercise uses the CY8CKIT-062S2-43012. This material is covered in 2.4.4.

☐ 1. Use Project Creator to create a new application called **ch02_ex18_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 user button on the kit 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 19: (UART-PDL) Read a value using the standard UART functions

This exercise uses either the CY8CKIT-149 or the CY8CKIT-062S2-43012. This material is covered in 2.4.4.

☐ 1. Use Project Creator to create a new application called **ch02_ex19_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 20: (UART-PDL) Write a value using the standard UART functions

This exercise uses either the CY8CKIT-149 or the CY8CKIT-062S2-43012. This material is covered in 2.4.4.

☐ 1. Use Project Creator to create a new application called **ch02_ex20_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 user button on the kit and observe the value displayed in the terminal.

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

Published by

Infineon Technologies AG

81726 Munich, Germany