# PSOC™ Digital Peripherals Training Manual

## About this document

This training manual covers the labs for the introduction to common peripherals available on PSOC™ 4, PSOC™ 6, and PSOC™ Edge microcontrollers.

### Scope and purpose

The training will cover project creation, GPIO control, UART and retartget-IO, SCBs, TCPWMs, interrupts, Systick, and WDT control using ModusToolbox™ and the device Peripheral Driver Library (PDL).

### Intended audience

The intended audiences for this document are design engineers, technicians, and developers of electronic systems.

Table of contents

# Table of contents

**About this document** ..... **1**
**Table of contents** ..... **2**
**1 Introduction** ..... **4**
**2 Required development tools and prerequisites** ..... **5**
2.1 Tools ..... 5
2.2 Prerequisites ..... 5
**3 Empty project creation and GPIO control** ..... **6**
3.1 Objective ..... 6
3.2 Description ..... 6
3.3 Project Creation ..... 7
3.4 GPIO control through the Device Configurator ..... 10
3.5 GPIO control through only the Peripheral Driver Library (PDL) ..... 19
3.6 Reading a GPIO input ..... 20
3.7 Interrupt on a GPIO ..... 22
3.8 Conclusion ..... 24
**4 Serial Communication Block with UART** ..... **25**
4.1 Objective ..... 25
4.2 Description ..... 25
4.3 Adding a UART SCB ..... 26
4.4 Receiving data from UART ..... 31
4.5 Enabling interrupts for UART ..... 32
4.6 Enabling retarget-IO middleware ..... 34
4.7 Conclusion ..... 35
**5 Timer-Counter-PWMs** ..... **36**
5.1 Objective ..... 36
5.2 Description ..... 36
5.3 Timer-compare counter ..... 37
5.4 PWM ..... 41
5.5 Timer-capture for event counting ..... 45
5.6 Timer-capture for event measuring ..... 52
5.7 Conclusion ..... 57
**6 Use the EZI2C driver to communicate with an I2C host** ..... **58**
6.1 Objective ..... 58
6.2 Description ..... 58
6.3 Enabling EZI2C slave peripheral ..... 60
6.4 Updating the EZI2C buffer with Systick up time ..... 66
6.5 Controlling an LED using an I2C write command ..... 70
6.6 Conclusion ..... 71
**7 Serial Communication Block with SPI** ..... **72**
7.1 Objective ..... 72
7.2 Description ..... 72
7.3 SPI master ..... 74
7.4 SPI slave ..... 79
7.5 SPI slave with callbacks ..... 84
7.6 Conclusion ..... 85
**8 Watch dog timer** ..... **86**

## Table of contents

8.1 Objective ..... 86
8.2 Description ..... 86
8.3 WDT with a system reset on expiration ..... 88
8.4 Conclusion ..... 90
**9 Appendix ..... 91**
9.1 Disabling the HAL on PSOC™ 6 BSPs ..... 91
9.2 Disabling FreeRTOS from PSOC™ Edge Empty Application ..... 93
9.3 PSOC™ Edge MTB-HAL for retarget-IO ..... 98
9.4 PSOC™ 6 system and peripheral clocks ..... 99
9.5 Evaluation Kits ..... 101
**References ..... 102**
**Glossary ..... 103**
**Revision history ..... 104**
**Disclaimer ..... 105**

# 1 Introduction

This manual provides instructions to create an empty project for PSOC™ devices. It will cover adding in peripherals such as GPIOs, SCBs, TCPWMs, and WDT. It will also cover the use of peripheral driver libraries (PDL) and libraries like retarget-IO which is used for debug print statements.

# 2 Required development tools and prerequisites

## 2.1 Tools

- ModusToolbox™ software v3.5 or later (tested with v3.5)
- Supported evaluation kits:
    - CY8CPROTO-040T-MS, the PSOC™ 4000T Multi-Sense Prototyping Kit
    - CY8CPROTO-041TP, the PSOC™ 4100T Plus Prototyping Kit
    - CY8CKIT-041S-MAX, the PSOC™ 4100S Max Pioneer Kit
    - CY8CKIT-062S2-43012, the PSOC™ 6 Wi-Fi BT Pioneer Kit
    - KIT_PSE84_EVAL, the PSOC™ Edge E84 Evaluation Kit
- Jumper wires

Note: Only the CY8CKIT-041S-MAX, CY8CKIT-062S2-43012, and KIT_PSE84_EVAL support the SPI lab. The SPI lab requires two evaluation kits, one to act as the master and one as the slave.

## 2.2 Prerequisites

1. Install ModusToolbox™ software v3.5
2. Eclipse IDE for ModusToolbox™ (Can be installed via the ModusToolbox™ setup tool)

# 3 Empty project creation and GPIO control

## 3.1 Objective

The objective of this lab is to cover how to use a GPIO as an output, input, and an interrupt. The lab will cover the ModusToolbox™ software tools to generate a project, setup the device, and navigate to the peripheral driver layer (PDL) documentation.

## 3.2 Description

The GPIO system on PSoC™ microcontrollers provide the interface between the CPU core and peripheral components to the outside world.

Shown here is a block diagram for a PSOC™ 4100T Plus GPIO. Most PSOC™ MCUs will have a similar GPIO block diagram, sharing concepts like the digital input and output paths, the analog path, and the high-speed IO matrix. In the digital input path, the control registers are used to configure items like the buffer mode selection, the high-speed IO matrix, and the interrupt logic. In the digital path, the control registers are used to configure items like drive mode, slew rate control, as well as HSIOM connections in the output direction. The analog path allows for some dedicated connections for analog resources like the SAR ADC and CAPSENSE™. There are also some HSIOM options in the analog path enabled by analog mux buses that allow for a very high configurability of the analog peripherals to the GPIOs.

The high-speed IO matrix allows for connections between GPIOs and many digital and analog peripherals. This matrix has restrictions, so it is critical to check the device's datasheet for pin connections.

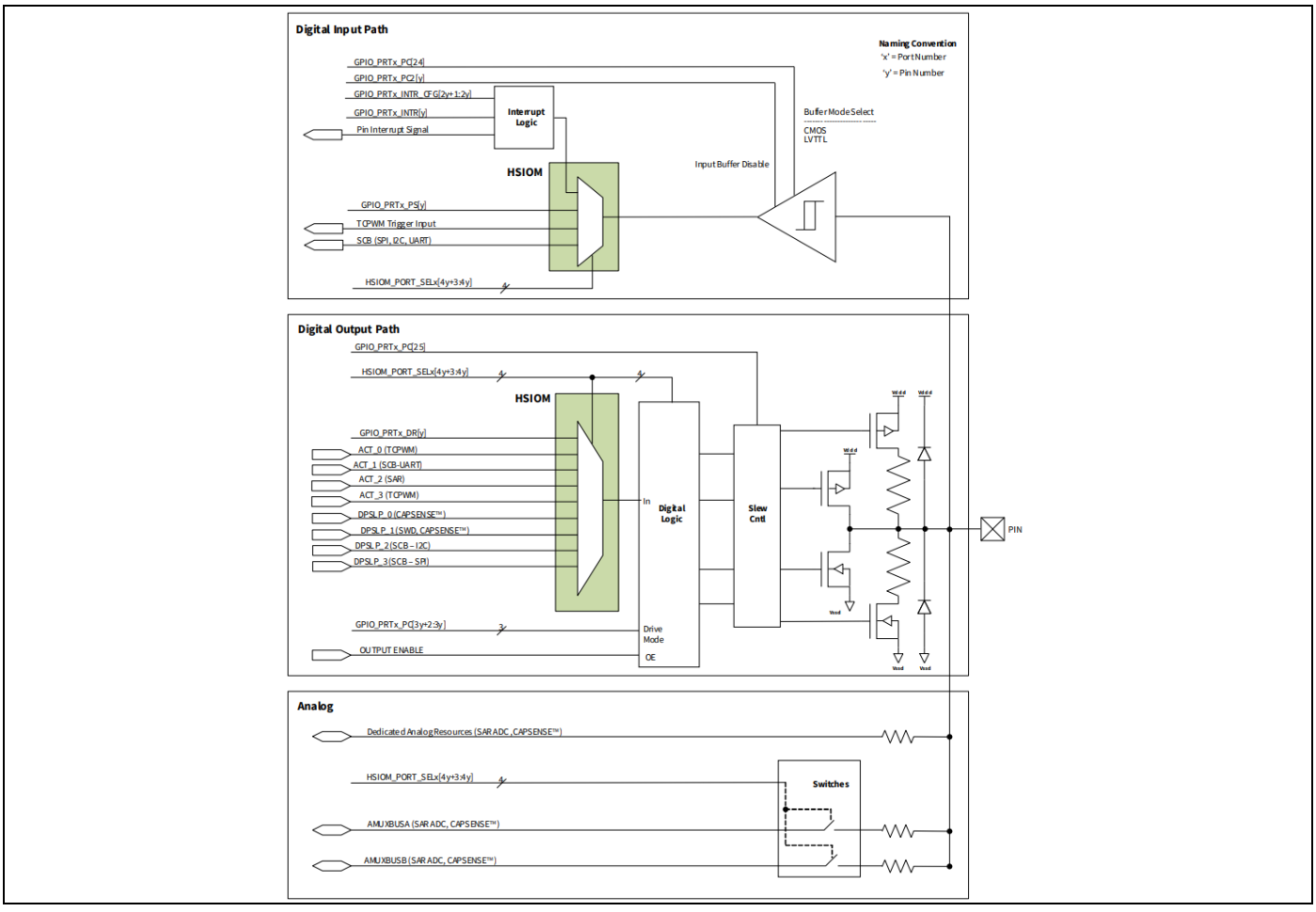



**Figure 1 PSOC™ 4100T Plus GPIO block diagram**

## 3.3 Project Creation

1. Before creating the first project in this training series we need to create our workspace in Modus™ Toolbox. Open **ModusToolbox™** from the **Windows Start** menu.

   Note: If you have not installed ModusToolbox™, see the Required development tools section.

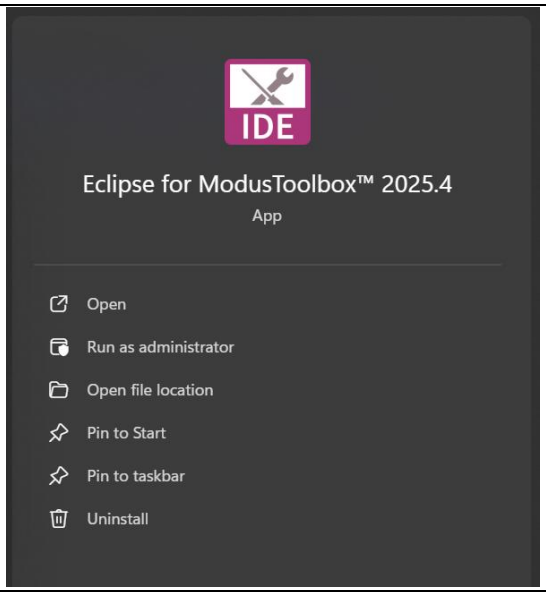



**Figure 2 Running Eclipse IDE from the Windows 11 Search Bar**

2. **Next**, choose a **workspace directory** for your project. After selecting a suitable directory, click the Launch button.

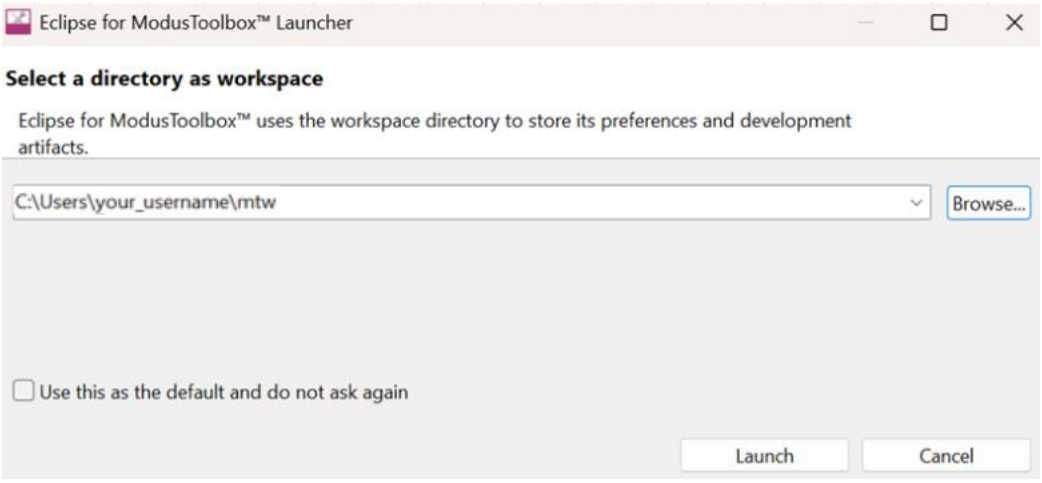



**Figure 3 Launching the Eclipse workspace**

## Empty project creation and GPIO control

3. Clicking Launch opens a new ModusToolbox™ workspace as shown below.

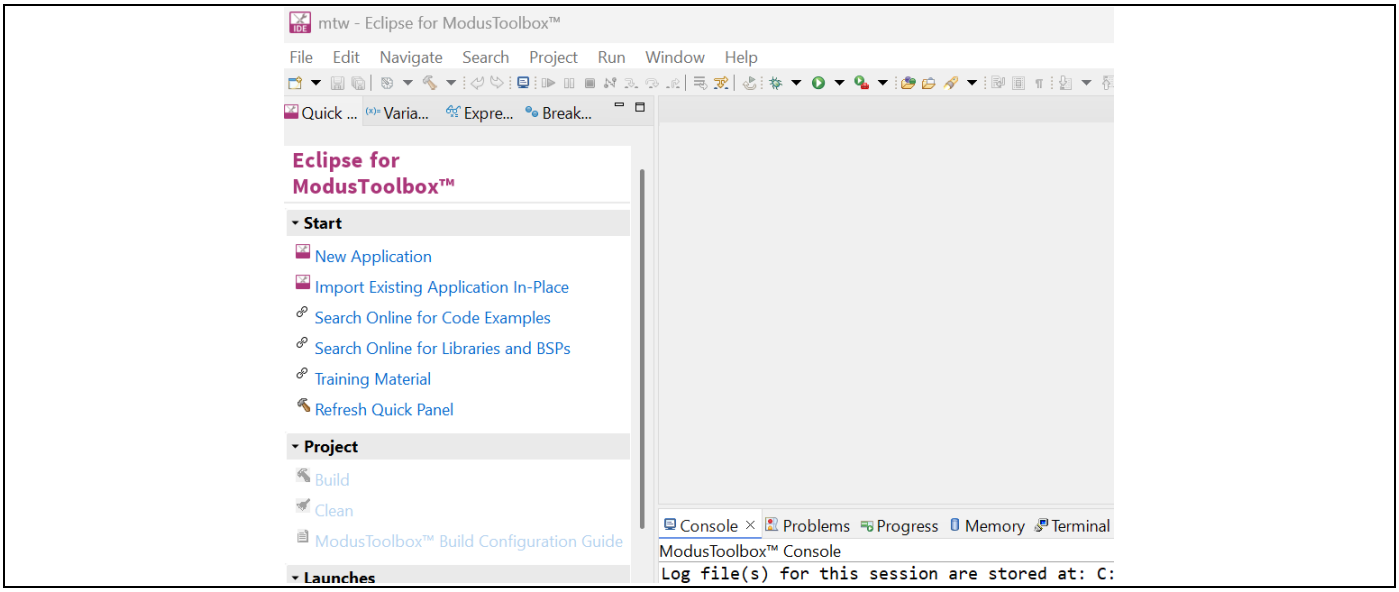



**Figure 4 Eclipse IDE after ModusToolbox™ Launch**

4. Now select **New Application** from the Quick Panel. Alternatively, go to **File > New > ModusToolbox™ Application**.
5. Clicking New Application opens the **Project Creator Tool** and prompts you to choose a **BSP** (Board Support Package).

   BSPs are aligned with our development/evaluation kits; they provide files for basic device functionality. A BSP typically has a **design.modus** file that configures clocks and other board-specific capabilities. That file is used by the ModusToolbox™ configurators. A BSP also includes the required device support code for the device on the board. You can modify the configuration to suit your application.
6. Once **Project Creator** launches, Select the BSP for the evaluation kit you are using under the PSOC™ family of BSPs you are using. Then click **Next**.

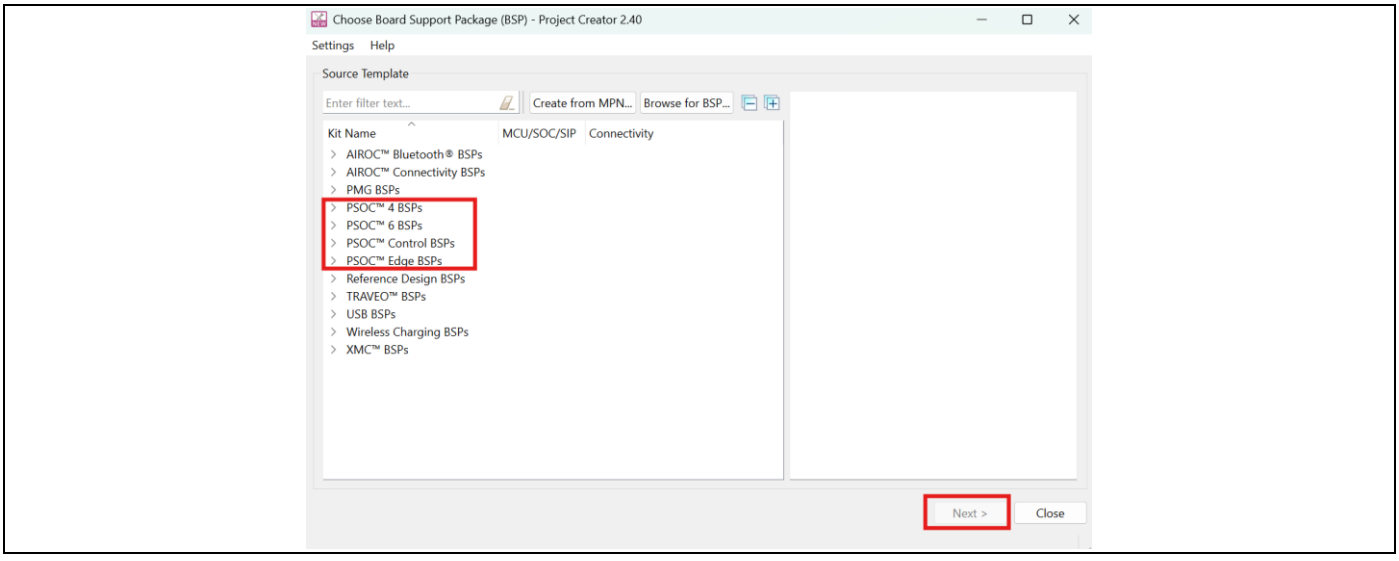



**Figure 5 Project Creator BSP Selection**

## Empty project creation and GPIO control

7. Clicking **Next** takes you to the next menu of the Project Creator which asks you to select an application. Select the **Empty PSOCx App** under the Getting Started section. You can do this by selecting the checkbox near the application. Click Create.

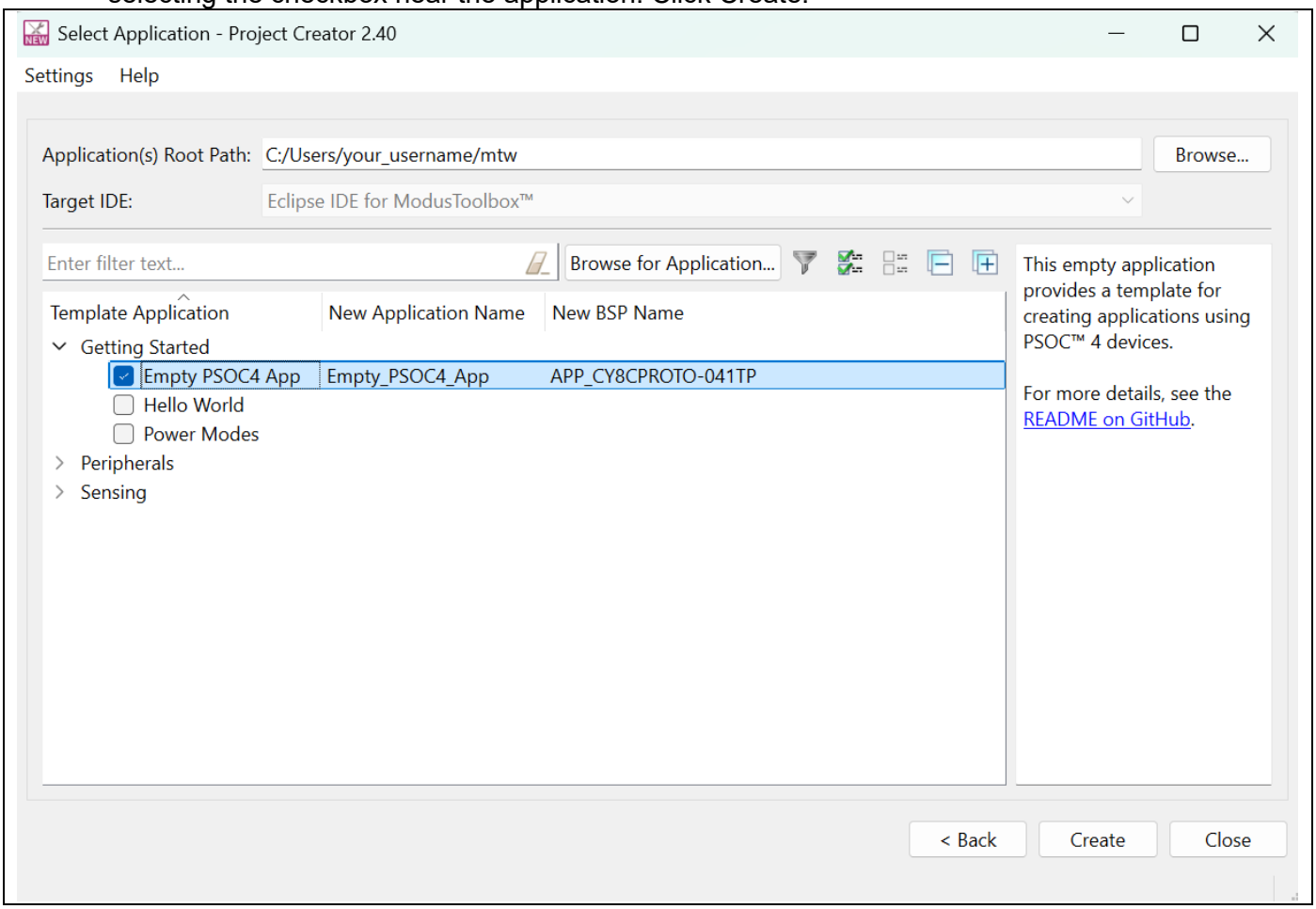



**Figure 6 Project Creator example code project creation**

8. When you click **Create**, a new project is created, and you can see all the files in the Project Explorer window.

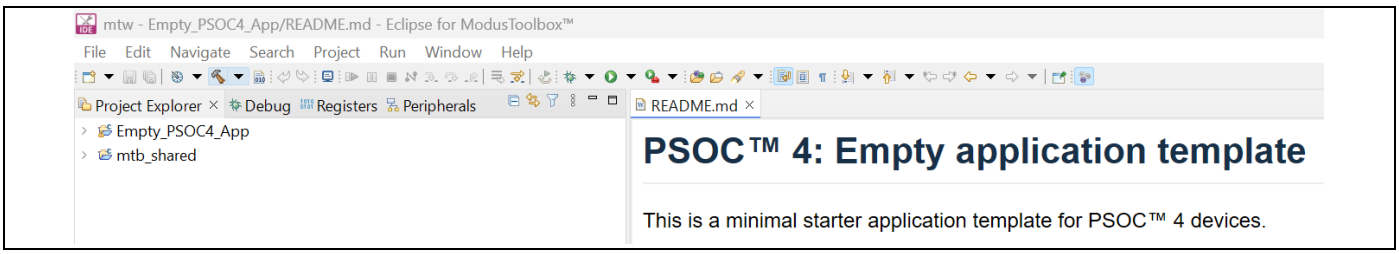



**Figure 7 Eclipse IDE after project creation**

Note 1: If you are using a PSOC™ 6 evaluation kit, please see the appendix for how to disable the HAL for PSOC™ 6 BSPs.

Note 2: If you are using a PSOC™ Edge evaluation kit, please see the appendix for how to disable FreeRTOS for the PSOC™ Edge Empty Application.

## 3.4 GPIO control through the Device Configurator

When configuring a GPIO for a PSOC™ microcontroller, there are 3 main settings to consider: drive mode, input threshold, and output slew rate.

**Drive Mode**: The 8 primary drive modes shown here in their simplified output driver diagrams

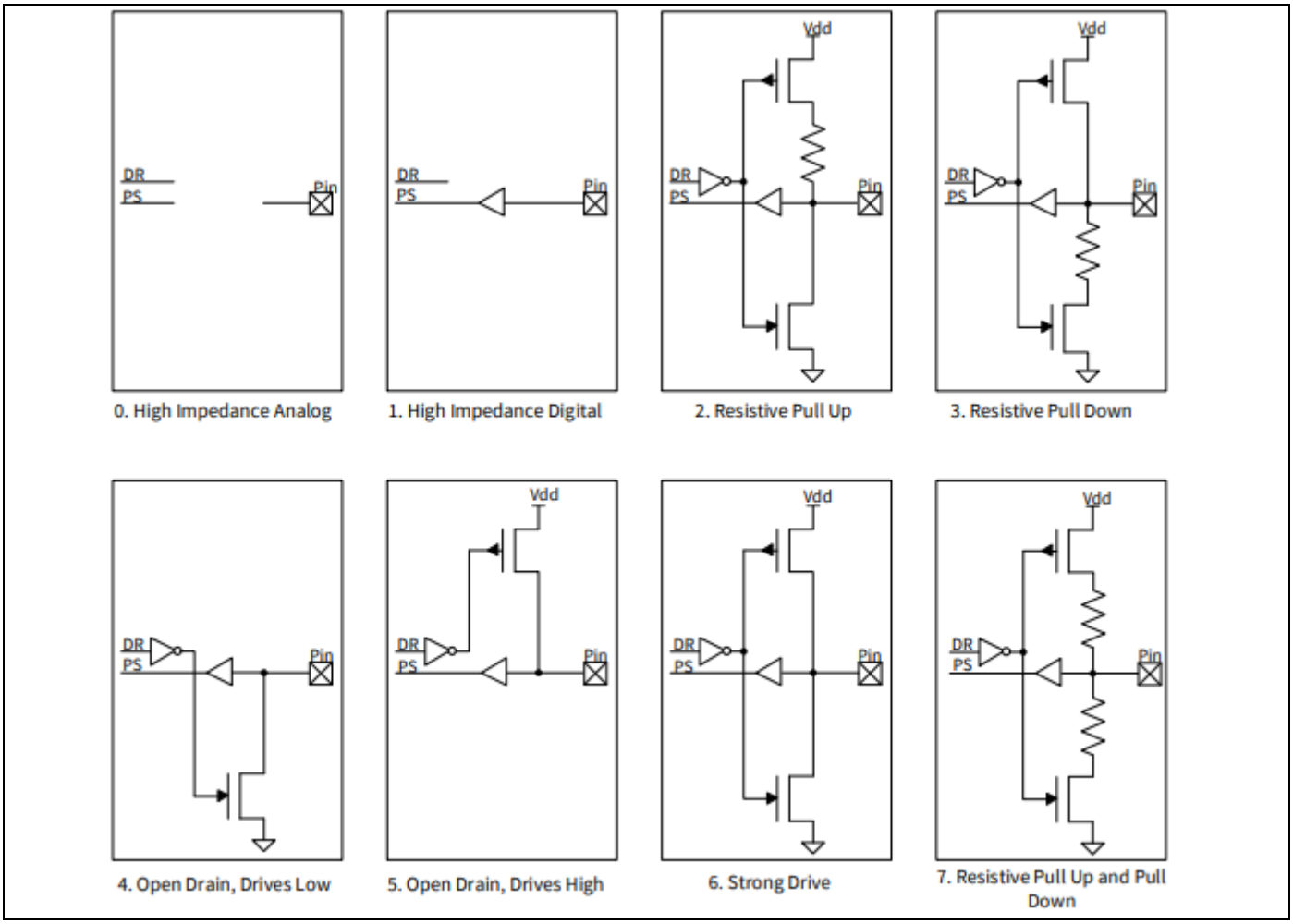



**Figure 8 Simplified GPIO output driver diagrams**

High impedance analog and digital are used when the GPIO is to be used as an input, analog or digital. Resistive modes provide a series resistance in one of the data states and strong drive in the other, or in the pull-up and pull-down case there is a series resistor in both data states. Open drain modes provide high impedance in one of the data states and strong drive in the other. The strong drive mode is the standard digital output mode for pins; it provides a strong CMOS output drive in both high and low states.

When using a GPIO as an input, the thresholds (Vih and Vil) are configurable by selecting the input buffer mode. The options are CMOS (Complementary Metal-Oxide-Semiconductor) or LVTTL (low voltage transistor-transistor logic). Typically, CMOS input thresholds will be slightly lower than LVTTL thresholds. Most of the time, LVTTL inputs are only required when interfacing with circuits that require TTL compatibility.

The slew rate for a GPIO configured in strong drive mode is configurable. The default is fast slew rate, but for certain circuits that require a slower slew rate, the configuration option is available.

1. Click on the **Empty_PSOC4_App** project in the Project Explorer and the quick panel will populate the application information, tools, and API documentation. Open the Device Configurator tool from the quick panel.

## Empty project creation and GPIO control

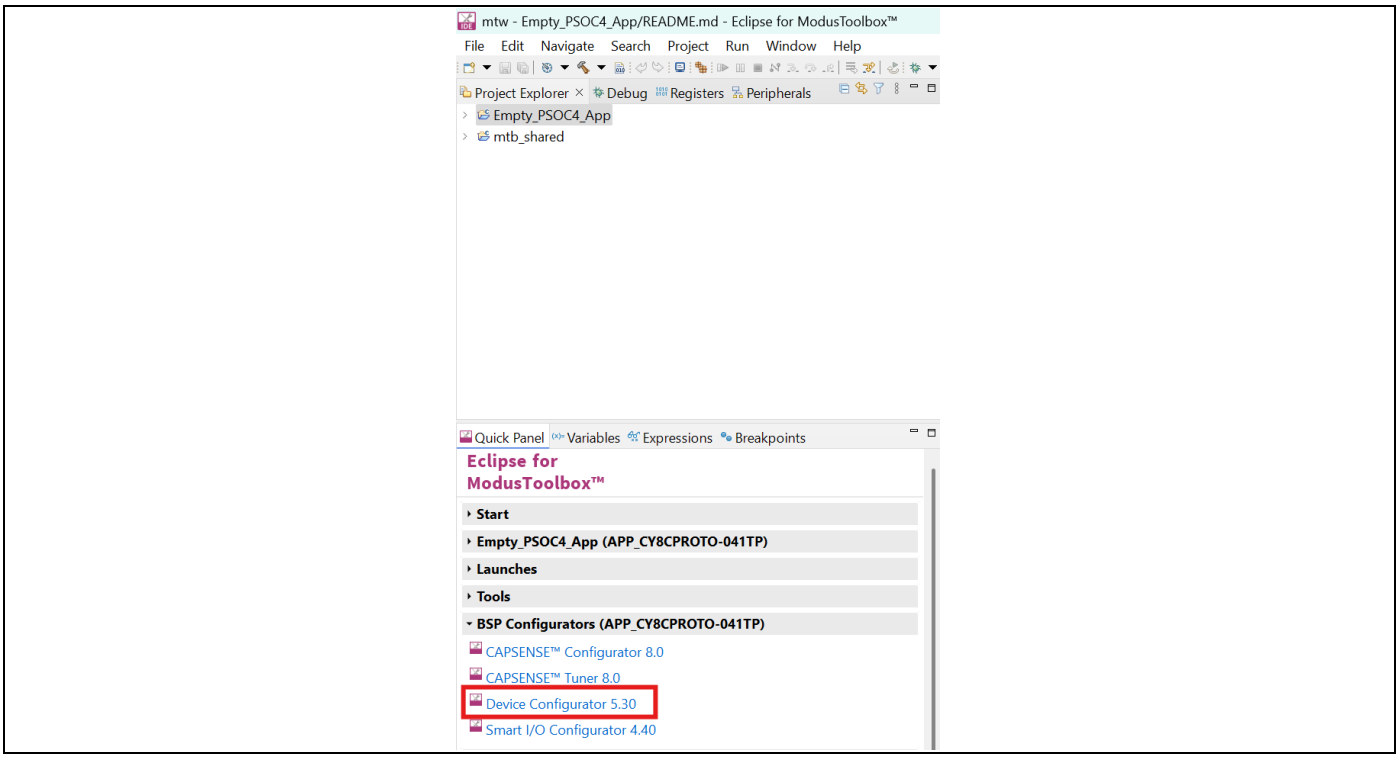



**Figure 9 Selecting Device Configurator from Eclipse IDE quick panel**

2. Navigate to the **Pins** tab and locate the port with pins used for LED control. This information can be found by searching through each port in the Device Configurator, or by reviewing the schematic for the evaluation kit used.

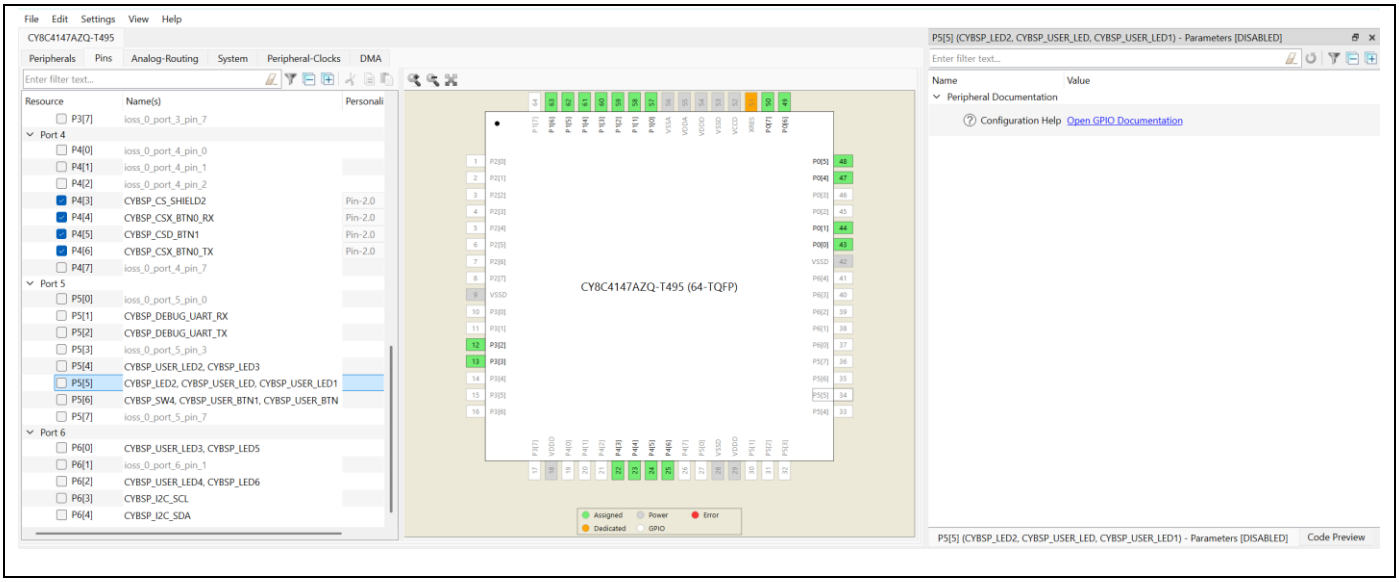



**Figure 10 Pins tab of the Device Configurator**

## Empty project creation and GPIO control

3. Enable the GPIO by pressing the check box of the desired GPIO, then in the **Parameters** window on the right, adjust the **Drive Mode** to **Strong Drive. Input buffer on**. Most LEDs on PSOC™ evaluation kits will be active high, meaning setting the corresponding GPIO high will turn the LED on. This means the **Initial Drive State** should be set to **Low (0)** so that the LED only turns on once the code written in the following steps executes. Once the changes have been made, save them by pressing the save icon, or by pressing ctrl+s.

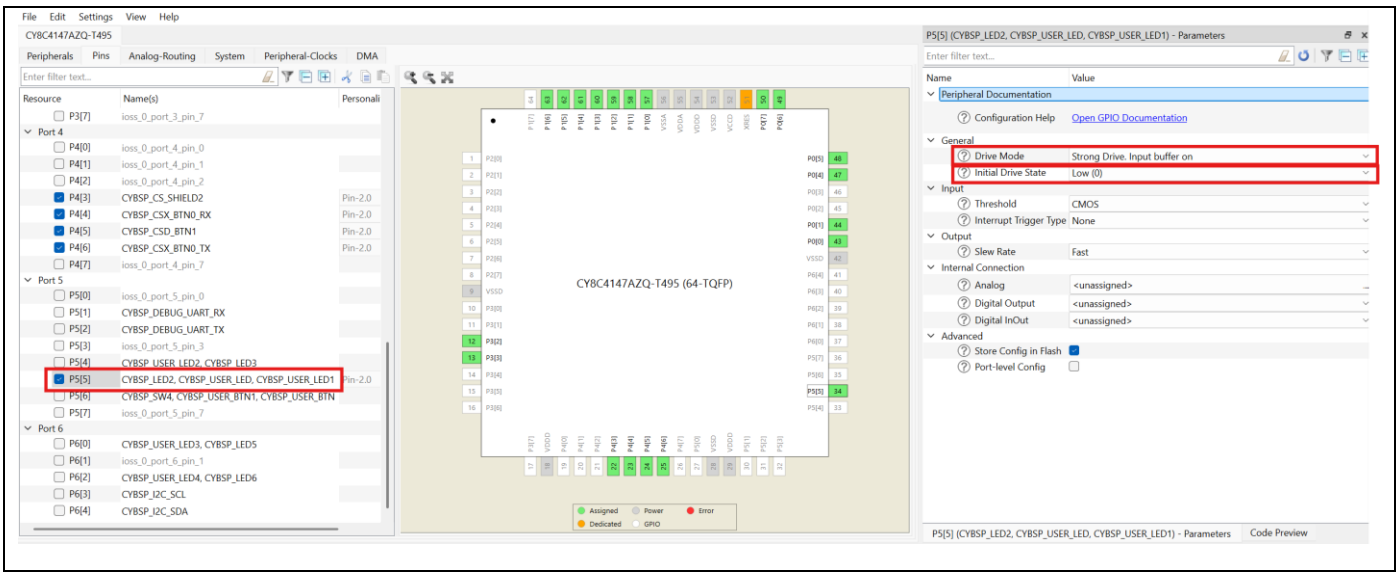



**Figure 11 Enabling a GPIO in the Device Configurator for LED control**

Notice in the **Resource** window on the left, that the GPIO enabled has one or multiple aliases. This can be modified, or the default BSP aliases can be used. This alias will be used later when calling the API functions to set and clear the GPIO. Typically for evaluation kit BSPs, the LEDs will have an alias like "**CYBSP_LEDx**" or "**CYBSP_USER_LEDx**".

4. In the **Parameters** window, there is a link for **Configuration Help**. Clicking this link will open the supporting documentation for the GPIO peripheral. It contains detailed information about the peripheral as well as API documentation for the PDL. Let's click on it to find the correct API to use for GPIO control.

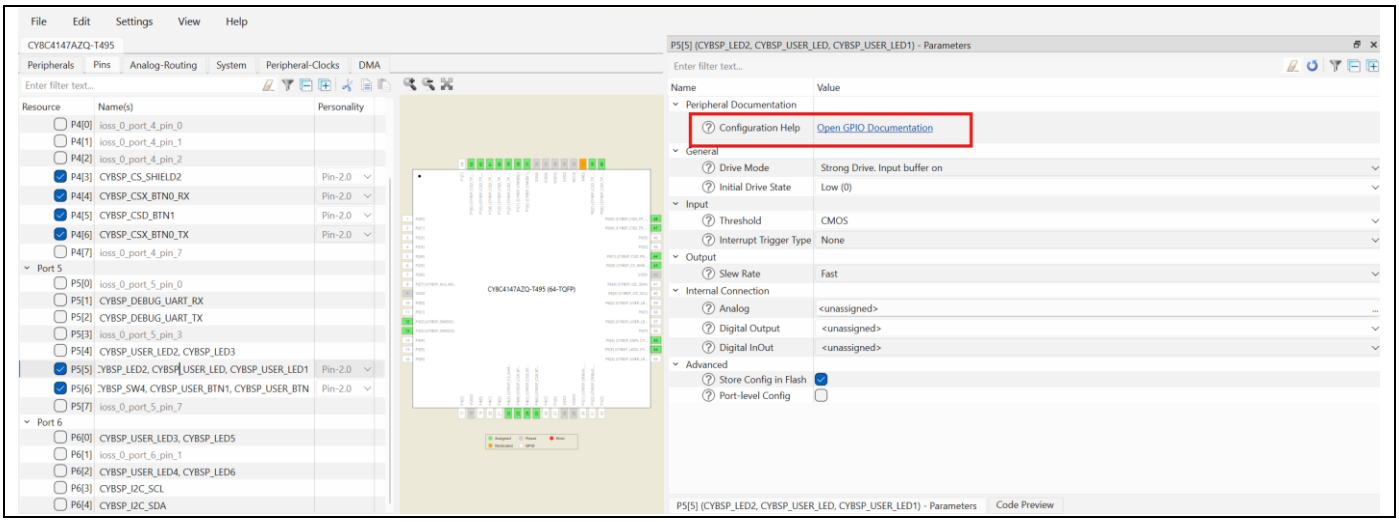



**Figure 12 Configuration help in parameter window**

## Empty project creation and GPIO control

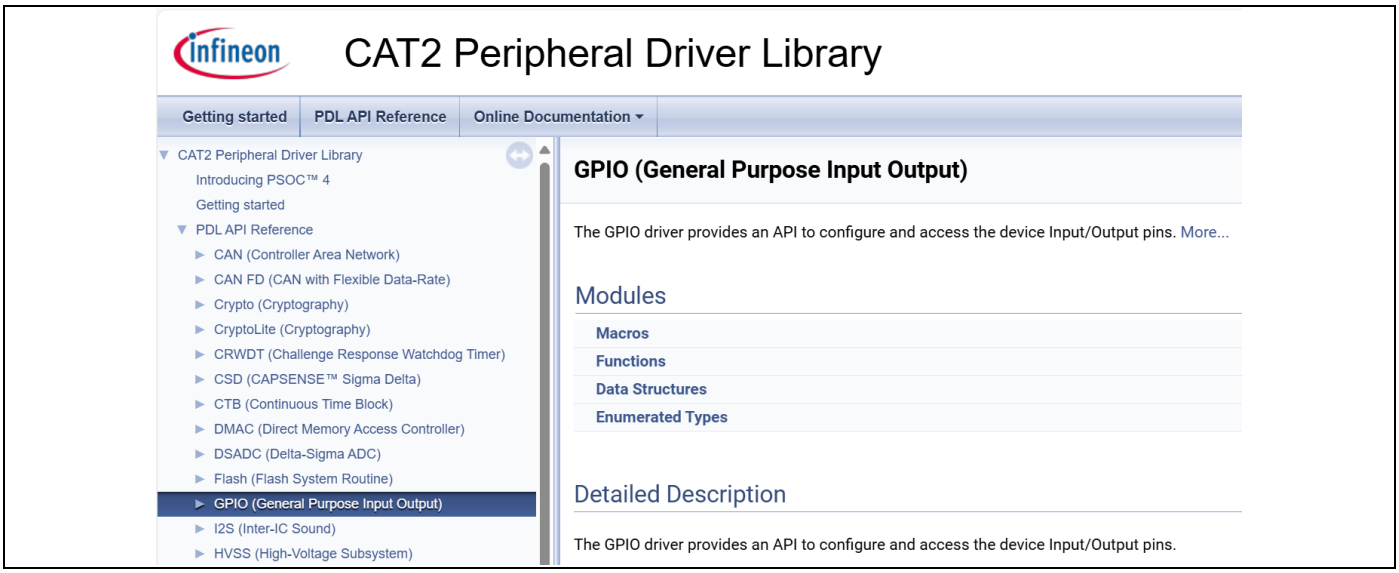



**Figure 13 PDL Documentation landing site for the GPIO peripheral**

5. Next, on the main GPIO window, under Modules, click on **Functions**.

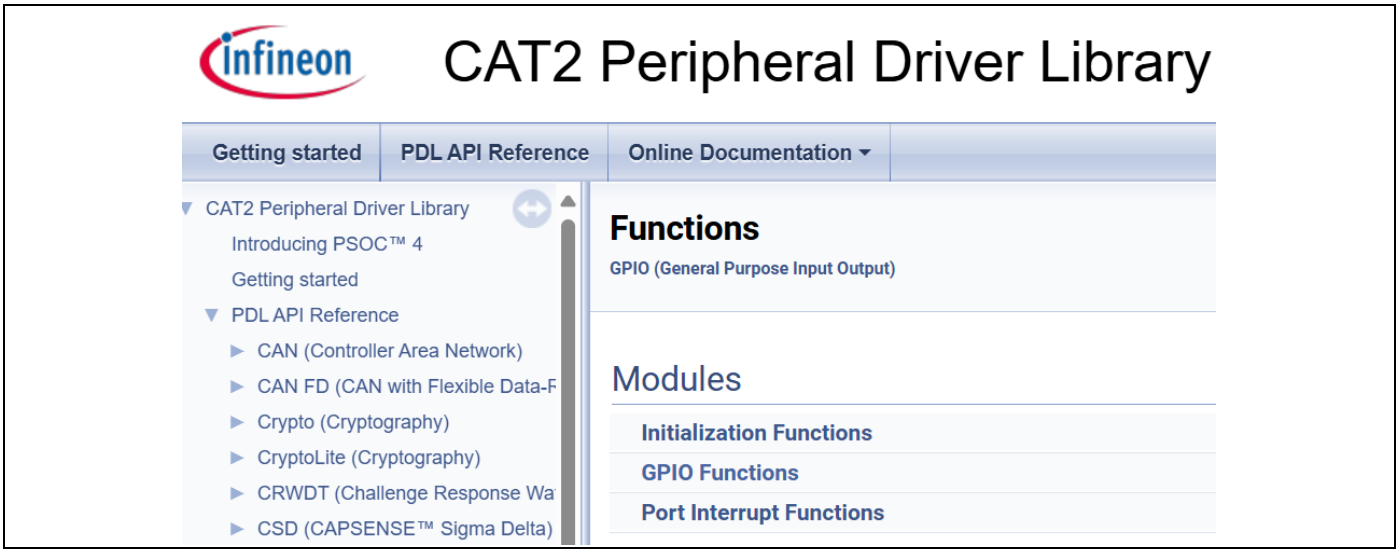



**Figure 14 PDL Documentation GPIO Function Modules**

## Empty project creation and GPIO control

6. Click on **GPIO Functions**.

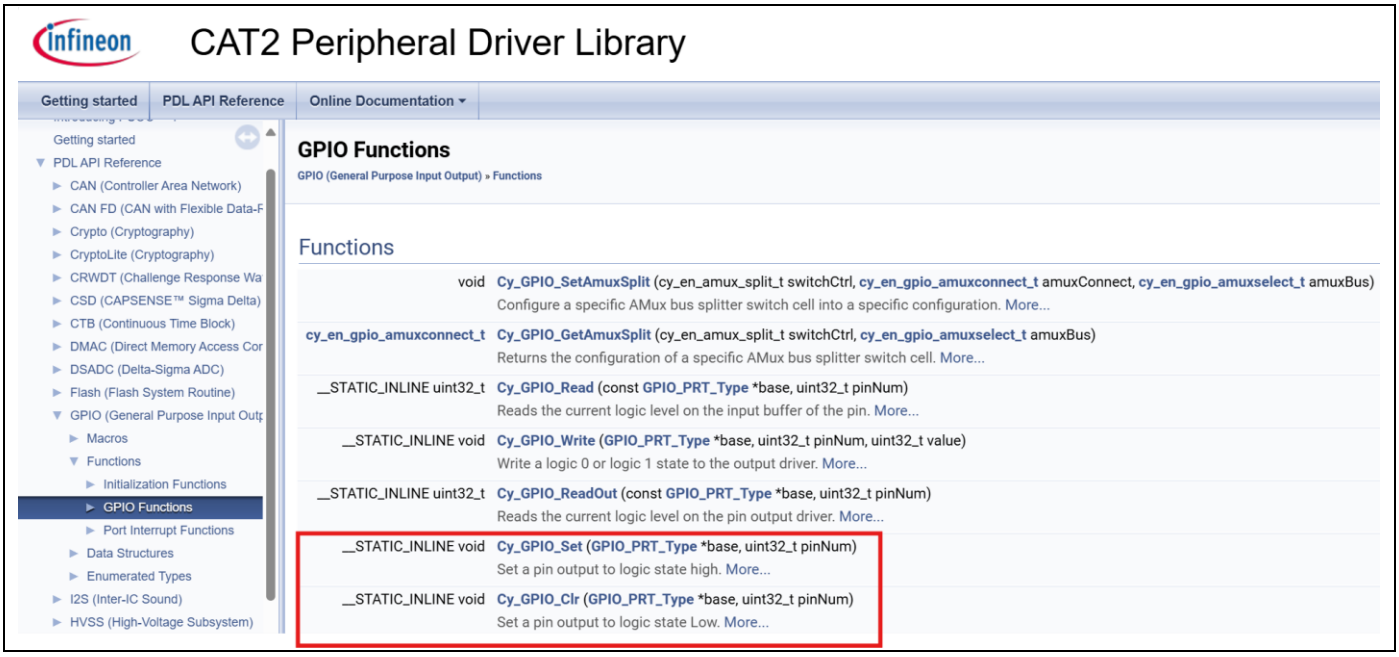



**Figure 15 PDL Documentation GPIO Functions**

7. For this lab, the **Cy_GPIO_Set** and **Cy_GPIO_Clr** functions will be used to set and clear the GPIO. Clicking on these functions will navigate to more details on how to use them.

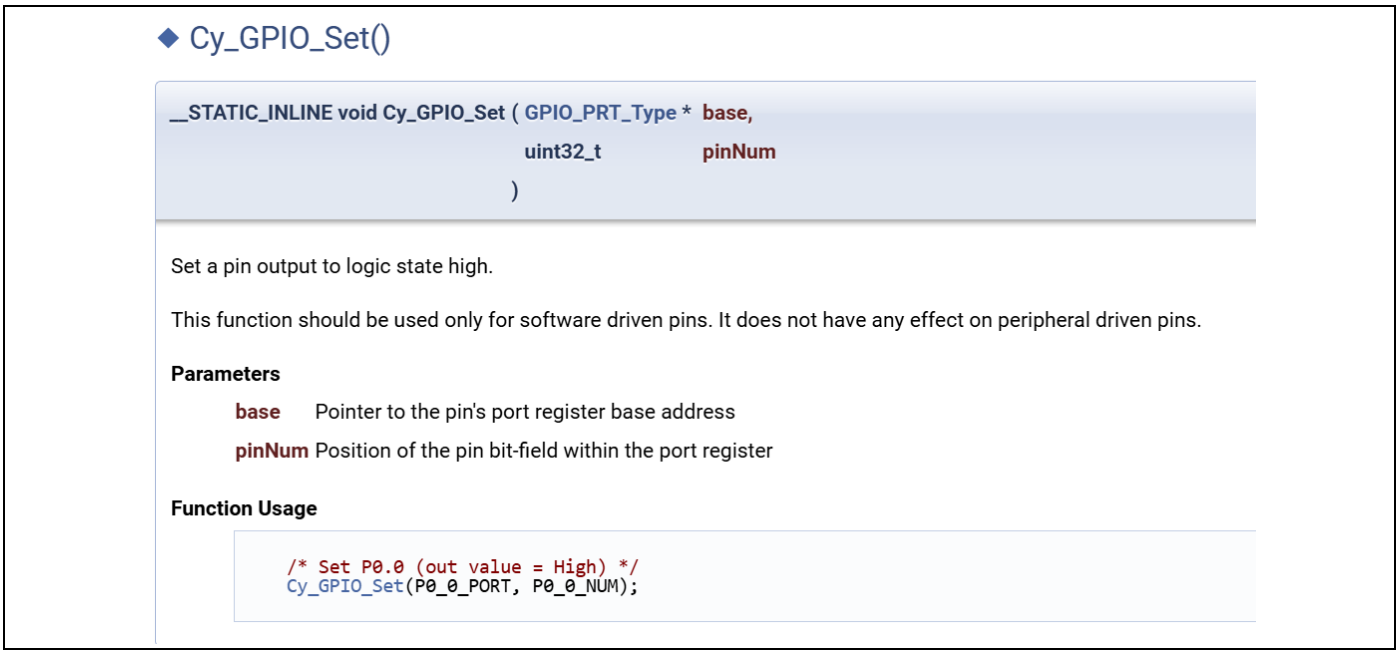



**Figure 16 Cy_GPIO_Set API Documentation**

8. Let's copy this line of code, navigate back to the Eclipse IDE and expand the **Empty_PSOC4_App** project. Then open the main.c file.

## Empty project creation and GPIO control

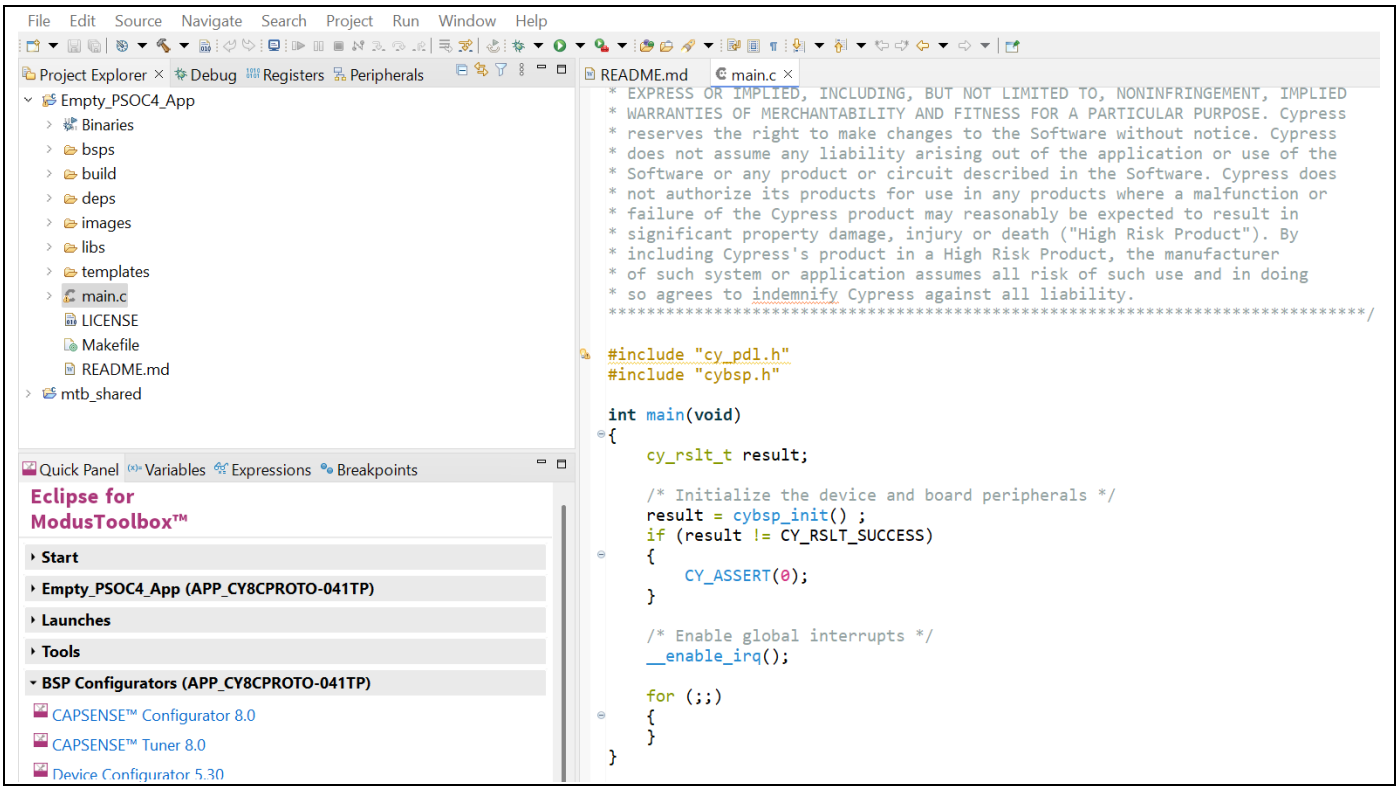



**Figure 17 Eclipse IDE view with main.c open for empty project**

9. Paste the **Cy_GPIO_Set** and **Cy_GPIO_Clr** APIs within the main.c for loop.

## Empty project creation and GPIO control

10. Navigate back to the device configurator and find the alias for the LED to be controlled in the pins tab on the left panel. The alias will be in the **Name(s)** column of the table. When using the device configurator to enable a GPIO a few behind the scenes items will occur.

    a. Definitions for the GPIO will be created which can be used in user code. These definitions will include items like the **_PORT**, **_PIN**, **_PORT_NUM**, **_DRIVEMODE**, **_INIT_DRIVESTATE**, and **_HSIOM** (this is used to select the High-Speed IO Matrix routing). These can be found in the code preview window on the right side of the device configurator.

    b. The GPIO will be initialized according to the settings selected in the Device Configurator. This will occur in the **cybsp_init** function.

    c. If a HAL is enabled, the GPIO usage will be reserved. HAL will not be covered in this lab manual.

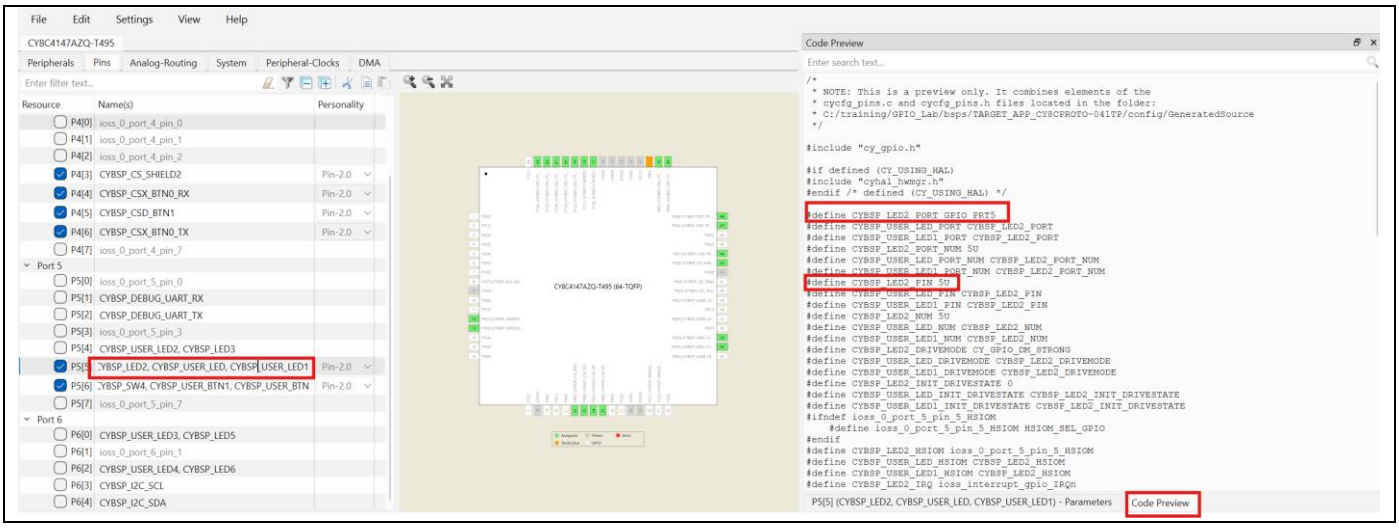



**Figure 18 Code preview for alias names of components**

11. The **Cy_GPIO_Set** and **Cy_GPIO_Clr** APIs require the pin **_PORT** and **_PIN** definitions. Use the alias with the added suffixes as inputs to those APIs. These macros can be found in the device configurator code preview window.

## Empty project creation and GPIO control

12. A delay is needed so that the GPIO setting and clearing will blink the LED at a rate that is observable. This can be quickly accomplished by using a blocking delay function **Cy_Syslib_Delay**. This API is in the Syslib PDL, which can be found in the CAT2 Peripheral Driver Library documentation.

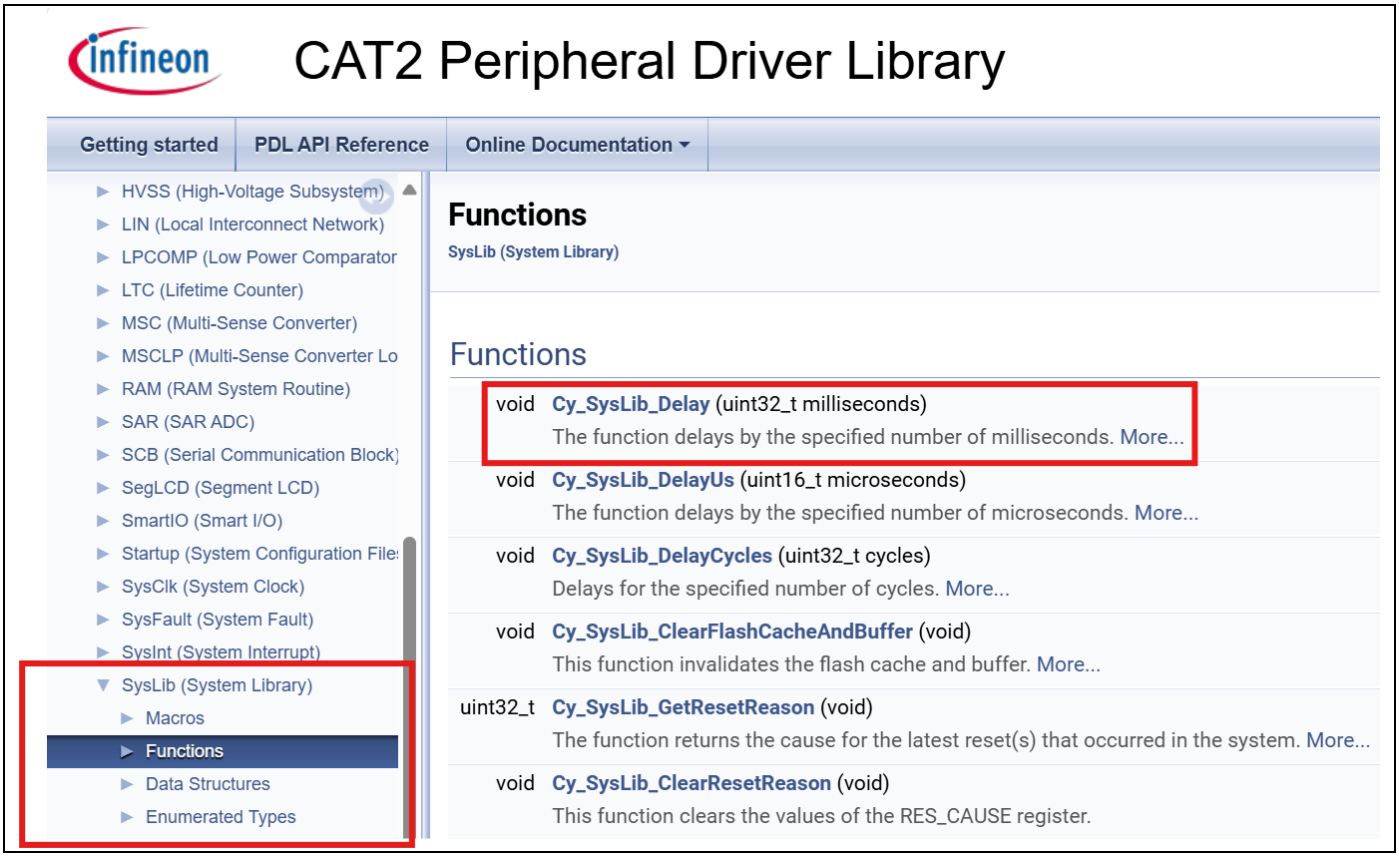



**Figure 19 Syslib PDL Documentation**

13. Update the main.c for loop to switch the LED on and off every second by adding the code in **BOLD**

```
for (;;)
{
        Cy_GPIO_Set(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
        Cy_SysLib_Delay(1000);
        Cy_GPIO_Clr(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
        Cy_SysLib_Delay(1000);
}
```

**Figure 20 For loop to blink an LED**

## Empty project creation and GPIO control

14. Program the device by navigating to the Eclipse IDE quick panel, and pressing the **Empty_PSOCx_App Program** button under the **Launches** tab and observe the LED blinking

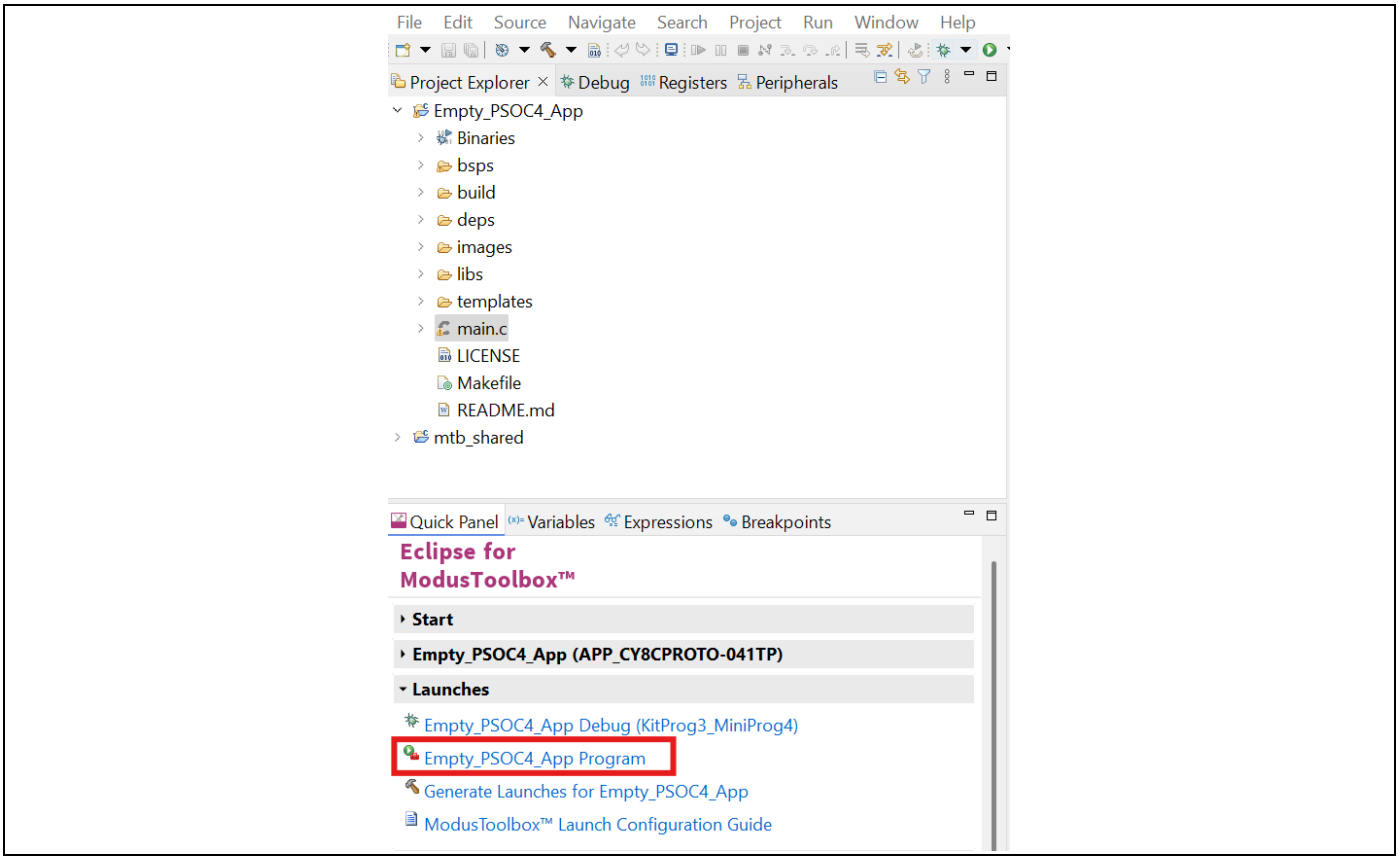



**Figure 21 Programming the device from the Eclipse IDE Quick Panel Launches tab**

## 3.5 GPIO control through only the Peripheral Driver Library (PDL)

An alternative to using the device configurator to setup a GPIO for application use is to use the middleware libraries for setup and control. Using the steps found in GPIO control through the device configurator, the PDL documentation can found. In this documentation the APIs to configure and control GPIOs can be found.

All GPIOs not configured by the device configurator follow the device default configuration. This is typically high impedance analog drive mode, CMOS input buffer, fast slew rate, and no internal connections. This means that to use a GPIO as an output, the only setting that must be changed is the drive mode.

1. Navigate to the corresponding evaluation kit user guide to find the pins used for the LEDs
2. Set the drive mode to strong output

```
#define NEW_LED_PORT    GPIO_PRT5
#define NEW_LED_PIN                (4u)

int main(void)
{
        cy_rslt_t result;

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

        Cy_GPIO_SetDrivemode(NEW_LED_PORT, NEW_LED_PIN, CY_GPIO_DM_STRONG);
```

**Figure 22 Using middleware APIs to set the drive mode for a GPIO**

3. Control the GPIO output with the _Set and _Clr APIs

```
for (;;)
{
        Cy_GPIO_Set(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
        Cy_GPIO_Clr(NEW_LED_PORT, NEW_LED_PIN);
        Cy_SysLib_Delay(1000);
        Cy_GPIO_Clr(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
        Cy_GPIO_Set(NEW_LED_PORT, NEW_LED_PIN);
        Cy_SysLib_Delay(1000);
}
```

**Figure 23 Using middleware APIs to set the drive mode for a GPIO**

4. Program the device and observe the two LEDs blinking

## 3.6 Reading a GPIO input

When configuring a GPIO as an input there are a few things to consider.

- What is driving the signal into the microcontroller?
    - When configured for digital input, a GPIO is a high z input, meaning that if not pulled or driven to a potential (VDD or GND), the input will generally be at some value in between. This means that the microcontroller will read a 1 (high) or 0 (low) at random.
    - If the driving circuit is an open drain circuit, then the opposite direction of the open drain should be pulled to a potential. For example, if a circuit is an open drain - drives low, then a pull-up should be added either externally, or the resistive pull-up drive mode should be used so that an internal pull-up resistor is enabled.
- As noted in the description of PSOC™ GPIOs here, the input threshold of a digital input is configurable and when connecting the microcontroller to a circuit that requires TTL compliant inputs, the input threshold should be set to LVTTL
- As the input to a GPIO is transitioning from high to low or low to high, it may pass the threshold multiple times

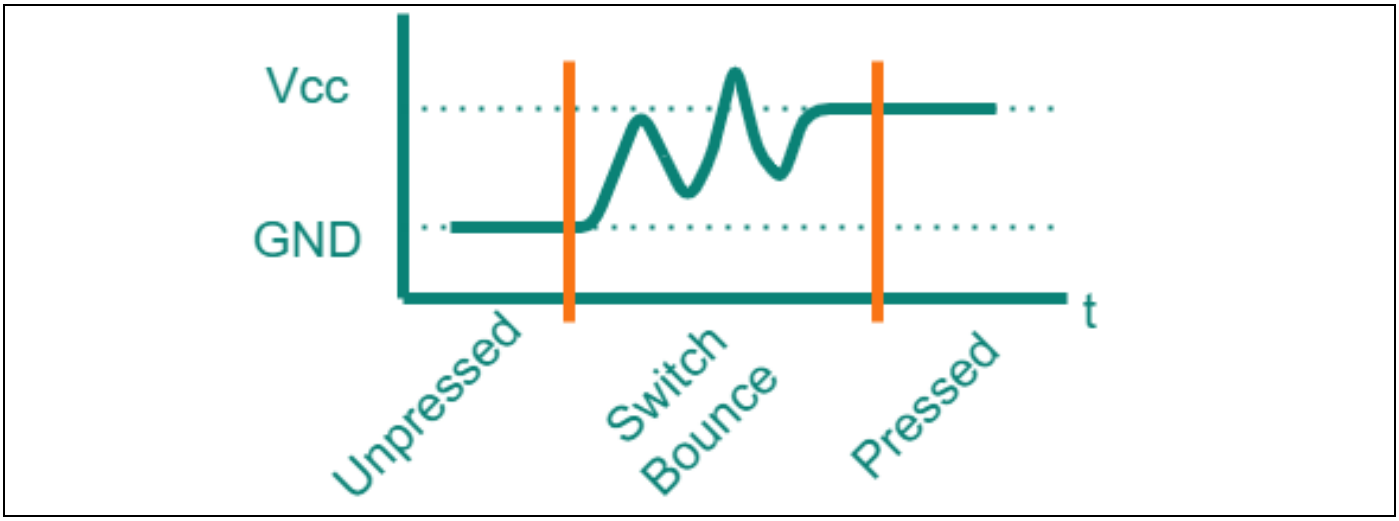



**Figure 24 Switch bounce**

    - Depending on the implementation, a debounce algorithm may need to be implemented in software.

## Empty project creation and GPIO control

1. Navigate to the Device Configurator to configure a GPIO for digital input by changing the drive mode to **Resistive Pull-Up, Input buffer on** and ensure that the initial drive state is **High (1)**

    a. Note that typically for evaluation kit BSPs, the user button will be labeled as "**CYBSP_USER_BTNx**"

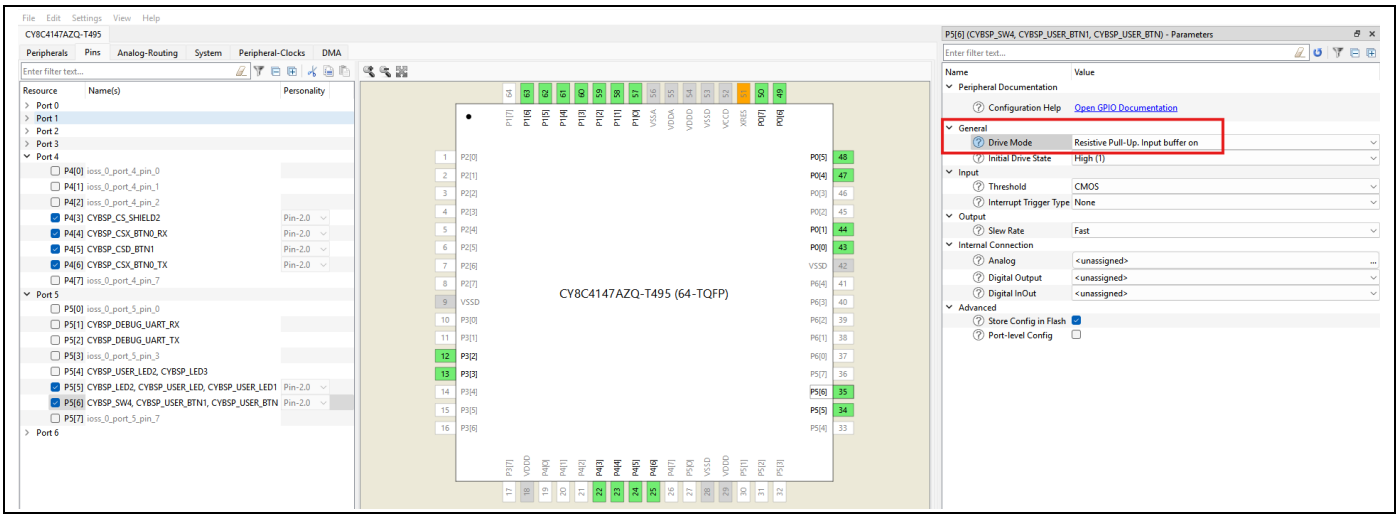



**Figure 25 Configuring a GPIO for digital input in the Device Configurator**

Note: The PSOC™ 4000T Multi-Sense Kit (CY8CPROTO-040T-MS), the LED2 and User Button pins are multiplexed through J6. This means that the code for controlling the LED2 will need to be removed.

2. Save and close the Device Configurator
3. Find the "**CY_GPIO_Read"** API in the PDL documentation, noting that the input to the function requires the GPIO port base address and the pin number
4. Add the if statement into the main function for loop to poll the GPIO and only toggle the GPIOs while the button is not pressed

```
for (;;)
{
        if (1UL == Cy_GPIO_Read(CYBSP_USER_BTN1_PORT, CYBSP_USER_BTN1_PIN))
        {
                Cy_GPIO_Set(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
                Cy_GPIO_Clr(NEW_LED_PORT, NEW_LED_PIN);
                Cy_SysLib_Delay(1000);
                Cy_GPIO_Clr(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
                Cy_GPIO_Set(NEW_LED_PORT, NEW_LED_PIN);
                Cy_SysLib_Delay(1000);
        }
        else
        {
                Cy_GPIO_Clr(NEW_LED_PORT, NEW_LED_PIN);
                Cy_GPIO_Clr(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
        }
}
```

**Figure 26 Reading a GPIO input**

5. Program the device and observe the two LEDs blinking, then press and hold the user button and observe the LEDs get set to off

Note that on the PSOC™ 4100S Max kit (CY8CKIT-041S-MAX), the LEDs are active low, meaning that driving the pin low causes the LED to turn on. This means that using the code found for this lab will cause the LED behavior to be inverted from the other kits.

## 3.7 Interrupt on a GPIO

Using a GPIO as an input has considerations like debounce, threshold settings, and static level control when based on the type of source driving the signal. When using a GPIO input as an interrupt, some other items should be considered.

- PSOC™ GPIOs can be used to detect the rising, falling, or both edges of a signal as it transitions

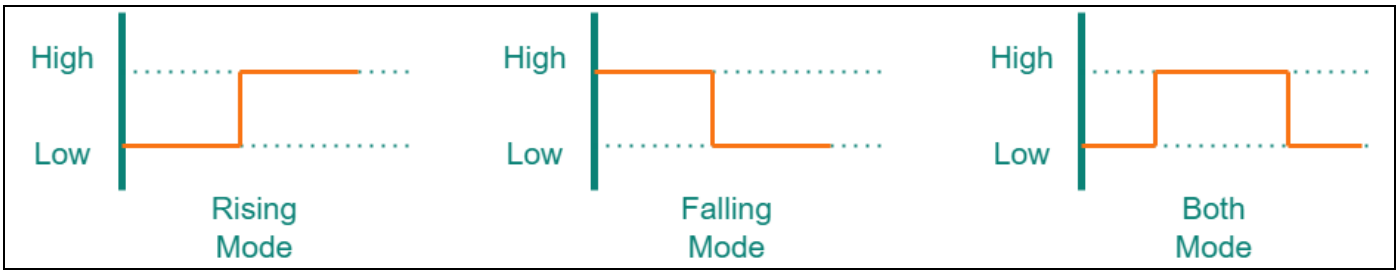



**Figure 27 GPIO edges**

  - Debouncing this signal may be required as the edge may be detected multiple times as the signal transitions from low to high or high to low

- Interrupt priority should also be considered
  - The interrupt could be indicating a very high or a very low priority event. The ARM Nested Vectored interrupt controller on PSOC™ MCUs will service high priority interrupts before low priority interrupts. Also, if a low priority interrupt is being serviced when a high priority interrupt occurs, the NVIC will suspend the low priority ISR and transfer control to the higher priority ISR

1. Navigate to the Device Configurator to configure a GPIO for a rising edge interrupt
   a. Some PSOC™ BSPs will have multiple buttons. If they do and they are enabled by default, the interrupt trigger type for the other buttons may be enabled. Please disable the other button interrupts

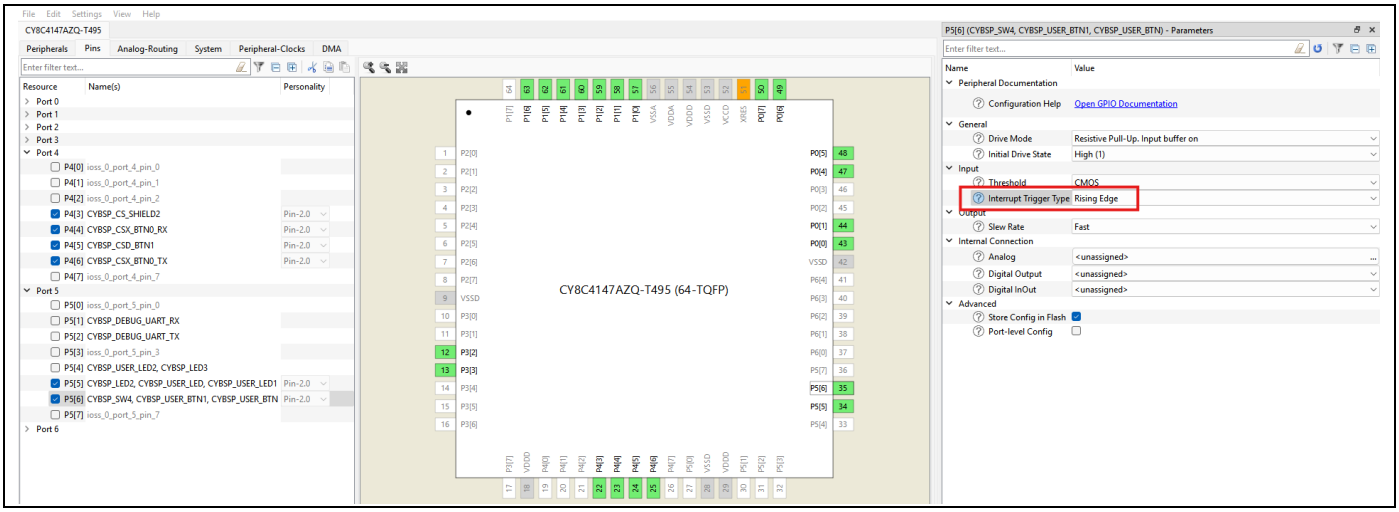



**Figure 28 Rising edge interrupt**

2. Save and close the Device Configurator

## Empty project creation and GPIO control

3. Navigate to the **SysInt** PDL documentation

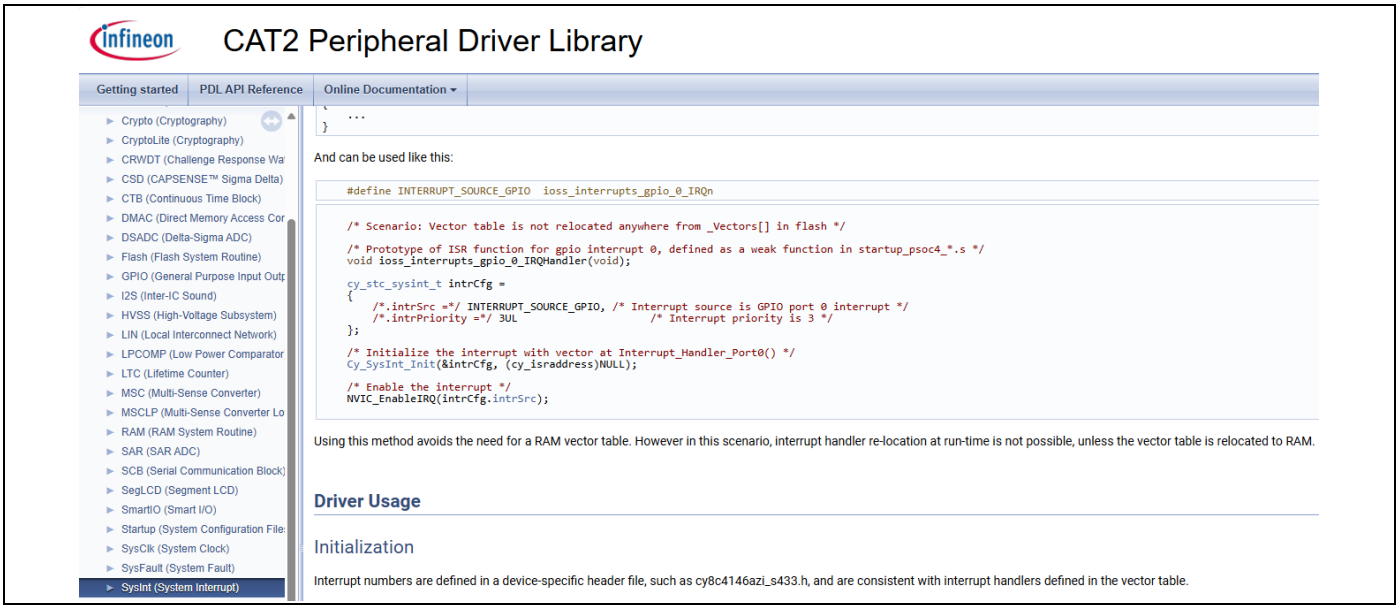



**Figure 29 SysInt PDL Documentation**

4. Before we initialize the interrupt, we need to create an interrupt service routine (ISR)

```
static bool enable_blink = false;

void button_isr(void)
{
        enable_blink ^= 1;
        Cy_GPIO_ClearInterrupt(CYBSP_USER_BTN1_PORT, CYBSP_USER_BTN1_PIN);
}
```

**Figure 30 Button interrupt service routine**

This ISR toggles a Boolean value, then clears the interrupt for the corresponding GPIO.

5. Initialize the SysInt for the corresponding GPIO

```
cy_stc_sysint_t intrCfg =
{
        /*.intrSrc =*/ CYBSP_USER_BTN1_IRQ,
        /*.intrPriority =*/ 3UL
};
Cy_SysInt_Init(&intrCfg, button_isr);

/* Enable the interrupt */
NVIC_EnableIRQ(intrCfg.intrSrc);
```

**Figure 31 Initialize the SysInt for the give GPIO**

Note: For CAT1 devices like PSOC6 and PSOC™ Edge, an interrupt mask may be needed for the interrupt to work properly. Please see the PDL documentation and look for **Cy_GPIO_SetInterruptMask**.

6. Update the main function for loop to use the **enable_blink** Boolean value

```
for (;;)
{
        if (1UL == enable_blink)
        {
                Cy_GPIO_Set(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
                Cy_GPIO_Clr(NEW_LED_PORT, NEW_LED_PIN);
                Cy_SysLib_Delay(1000);
                Cy_GPIO_Clr(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
                Cy_GPIO_Clr(NEW_LED_PORT, NEW_LED_PIN);
                Cy_SysLib_Delay(1000);
        }
        else
        {
                Cy_GPIO_Clr(NEW_LED_PORT, NEW_LED_PIN);
                Cy_GPIO_Clr(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
        }
}
```

**Figure 32 Updates to the main function for loop**

7. Program the device and note that the LEDs are not blinking, then press the button and observe the LEDs blinking

## 3.8 Conclusion

The final state of this lab should enable 3 GPIOs, two should be set to outputs for LED control, and the third as an input used to trigger an interrupt. The interrupt on the user button is used to control whether or not the LEDs are blinking.

# 4 Serial Communication Block with UART

## 4.1 Objective

The objective of this lab is to demonstrate how to enable a serial communication block, SCB, using the device configurator and the peripheral driver library. The retarget-IO middleware will also be enabled to showcase easier debugging with standard C libraries like printf.

## 4.2 Description

The serial communication block on PSOC™ devices is a highly configurable digital communication interface. It can be used to enable a universal asynchronous receiver/transmitter interface. This lab will cover setup, sending data, receiving data with a polling and an interrupt driven method, and the retarget-IO middleware.

- Adding a UART SCB
- Receiving data from UART
- Enabling interrupts for UART
- Enabling retarget-IO middleware

The Universal Asynchronous Receiver/Transmitter (UART) protocol is an asynchronous serial interface protocol. UART communication is typically point-to-point. The UART interface consists of two signals, the transmitter output (TX) and the receiver input (RX). The serial communication blocks on PSOC™ devices can be used as a UART controller, supporting multiple protocols, data frame sizes, programmable number of stop bits, parity support, programmable oversampling, start skipping, and hardware flow control (CTS and RTS).

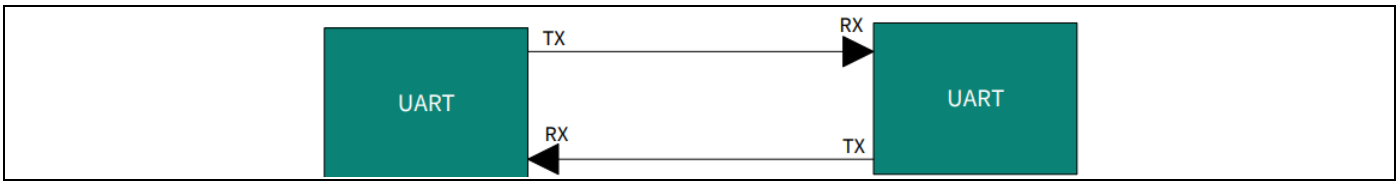



**Figure 33 UART example**

A typical UART transfer consists of a start bit followed by multiple data bits, optionally followed by a parity bit and finally completed by one or more stop bits. The start and stop bits indicate the start and end of data transmission. The parity bit is sent by the transmitter and is used by the receiver to detect single bit errors. Because the interface does not have a clock (asynchronous), the transmitter and receiver use their own clocks; thus, the transmitter and receiver need to agree on the baud rate.

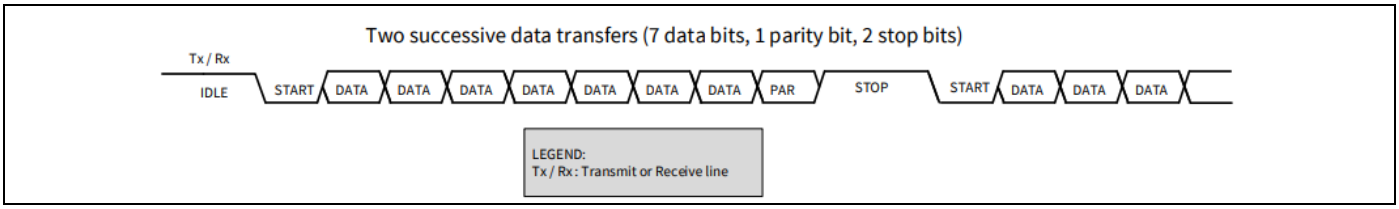



**Figure 34 UART, standard protocol example**

Most PSOC™ SCBs can support alternative UART modes like Smartcard reader, IrDA, and LIN. Those modes will not be covered in this lab.

## 4.3 Adding a UART SCB

1. The project created in the first exercise can be used, or the instructions shown there can be used to create a new project.
2. Once the project has been opened, navigate to the Eclipse IDE Quick Panel and open the Device Configurator. Navigate to the **Peripherals** tab and enable the SCB that is associated with the boards UART port. Typically for Infineon PSOC™ evaluation kit, the alias name will be pre-filled as "**CYBSP_UART**". If there is not an alias pre-filled, please see the user manual for your evaluation kit and determine which pins are connected to the KitProg for UART, then determine which SCB is associated with those pins.

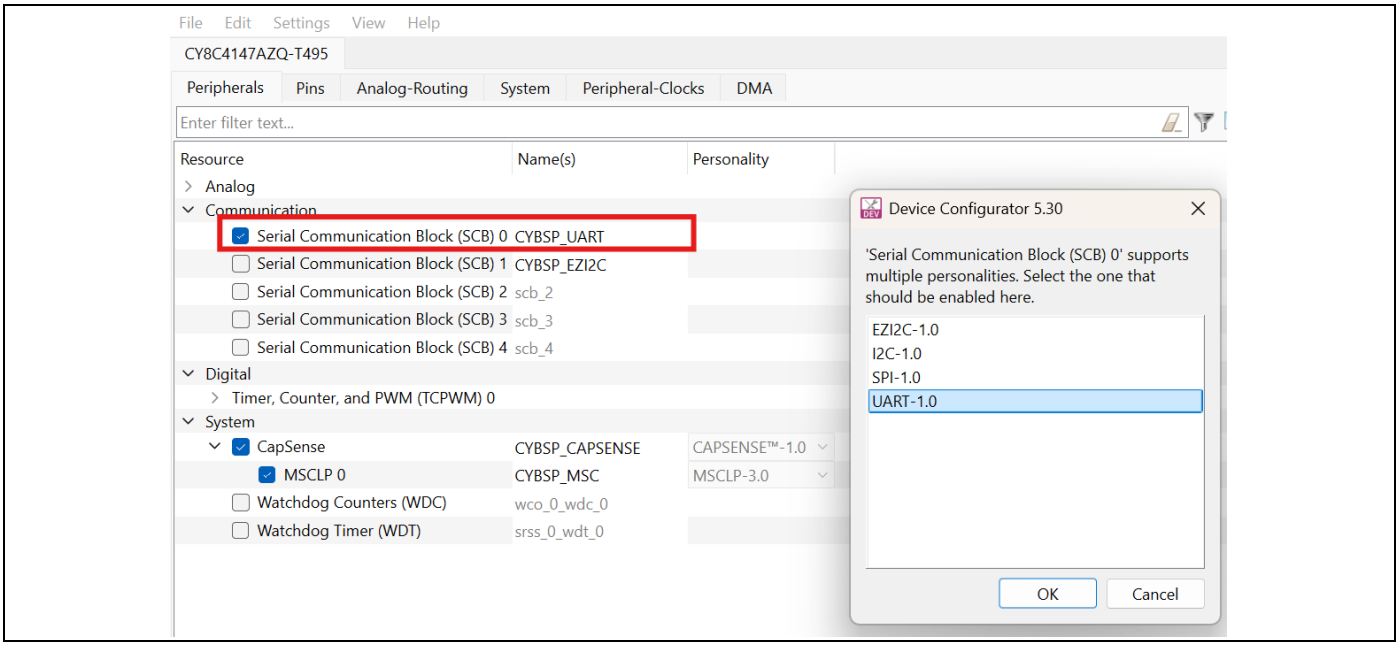



**Figure 35 Enable SCB as a UART interface in the Device Configurator**

## Serial Communication Block with UART

3. The settings can be configured in the parameters window on the right side of the device configurator.

    a. Standard mode, with a baud rate of 115200, LSB first bit order, 8 bits of data, no parity, and 1 stop bit.

    b. A clock will need to be supplied. PSOC™ microcontrollers have many configurable peripheral clocks. You can pick any unused clock divider, and the configurator will set it up for the chosen baud rate. Confirm that the **Baud Rate Accuracy %** is below 2%.

    c. Select the appropriate GPIOs for RX and TX. The configurator will configure these correctly if were not used previously. If they were enabled previously in the pins tab, then proceed to the next step to configure them correctly.

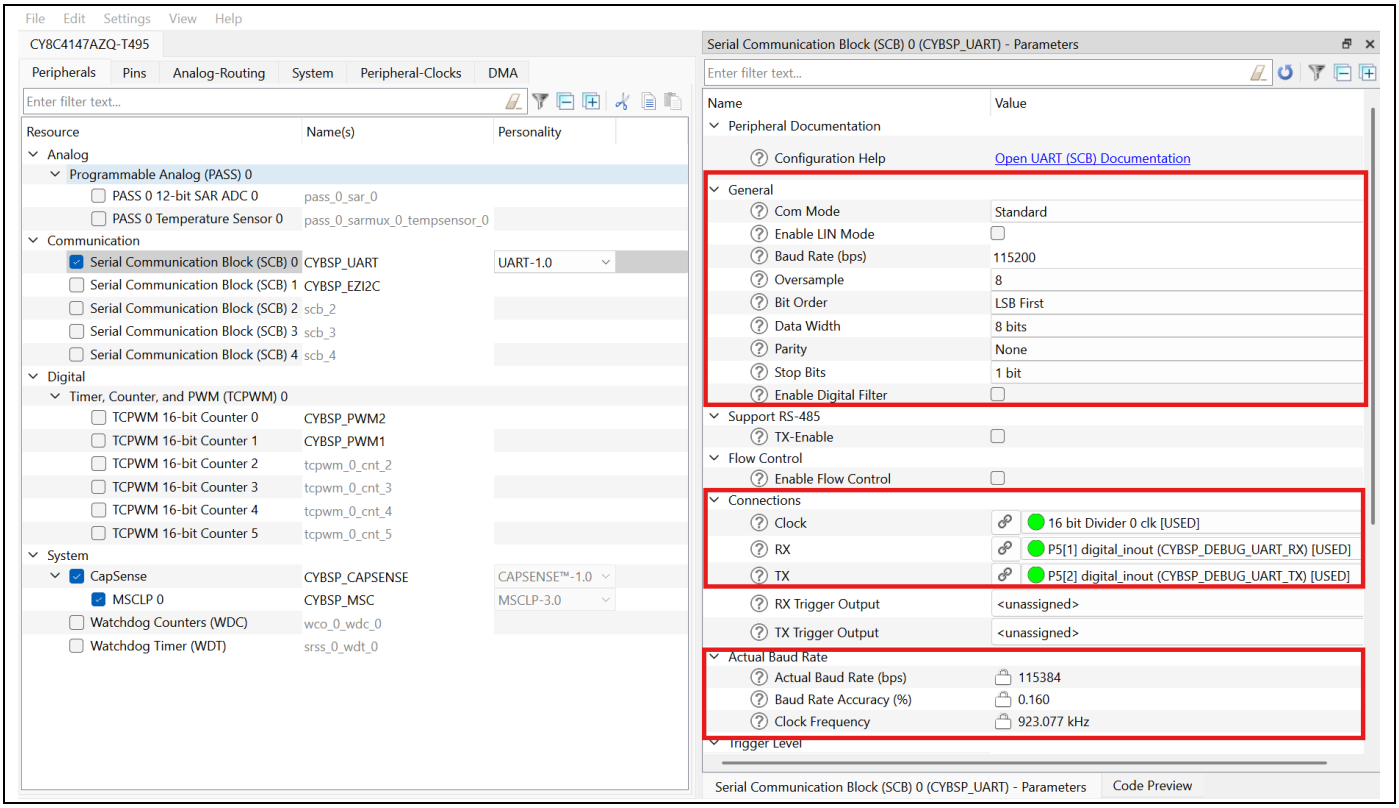



**Figure 36 Configure the SCB settings for Debug UART**

## Serial Communication Block with UART

4. Navigate to the **Pins** tab and ensure that the **RX** pin is set to **Digital High-Z. Input buffer on** drive mode and the **TX** pin is set to **Strong Drive. Input buffer off**

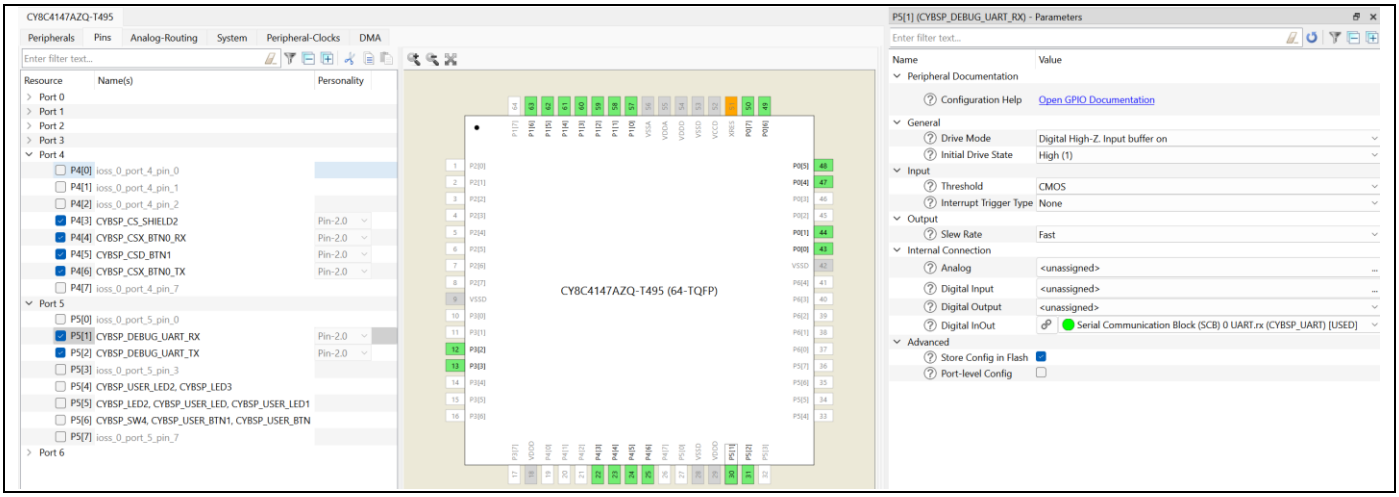



**Figure 37 Configure the RX GPIO drive mode**

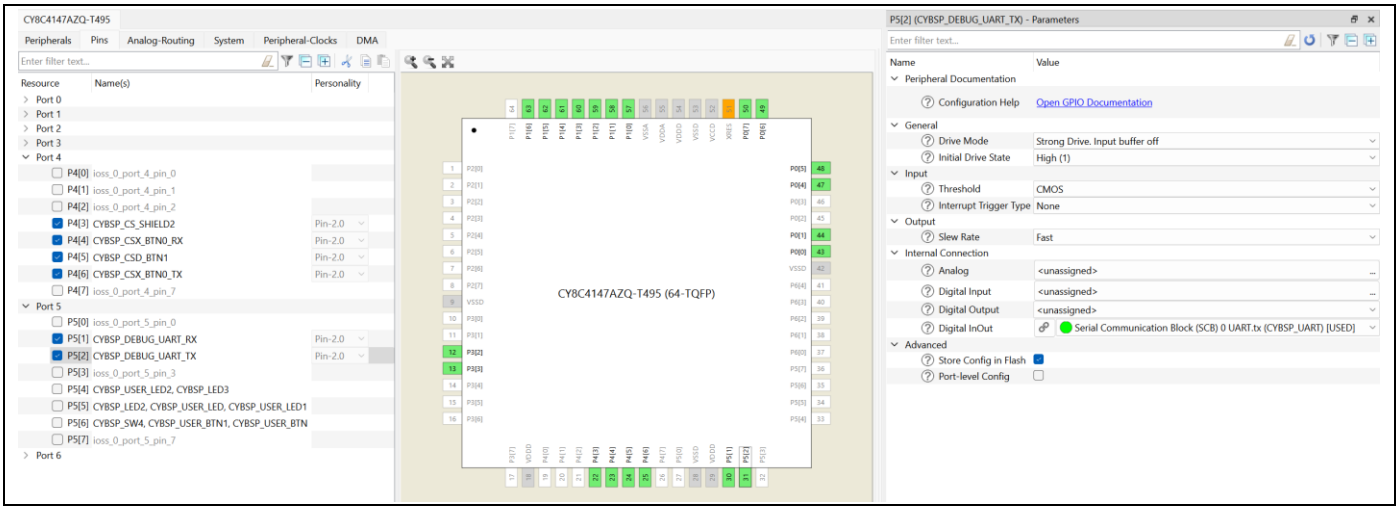



**Figure 38 Configure the TX GPIO drive mode**

5. Save the device configurator, but keep it open so that the code preview for the UART SCB can be viewed

## Serial Communication Block with UART

6. Navigate to the PDL documentation for the SCB in UART mode. Find the **Cy_SCB_UART_Init**, **Cy_SCB_UART_Enable**, and **Cy_SCB_UART_PutString** APIs.

Note that the PDL documentation shows how to create the config structure, but the device configurator generates the structure automatically when it is saved. To get started, only the **Cy_SCB_UART_Init**, **Cy_SCB_UART_Enable**, and **Cy_SCB_UART_PutString** APIs will be needed as the device configurator will generate and invoke the other needed setup code including the GPIO initialization and the peripheral clock initialization.

7. Add the required code to enable the SCB and send the first message.

```
Cy_SCB_UART_Init(CYBSP_UART_HW, &CYBSP_UART_config, NULL);

/* Enable global interrupts */
__enable_irq();

Cy_SCB_UART_Enable(CYBSP_UART_HW);
Cy_SCB_UART_PutString(CYBSP_UART_HW, "Hello PSOC\r\n");
```

**Figure 39 PDL Documentation for SCB in UART mode**

8. In Eclipse IDE, a serial terminal window can be opened by navigating to the **Terminal** tab in the output window at the bottom of the IDE, then pressing the **open a terminal** button

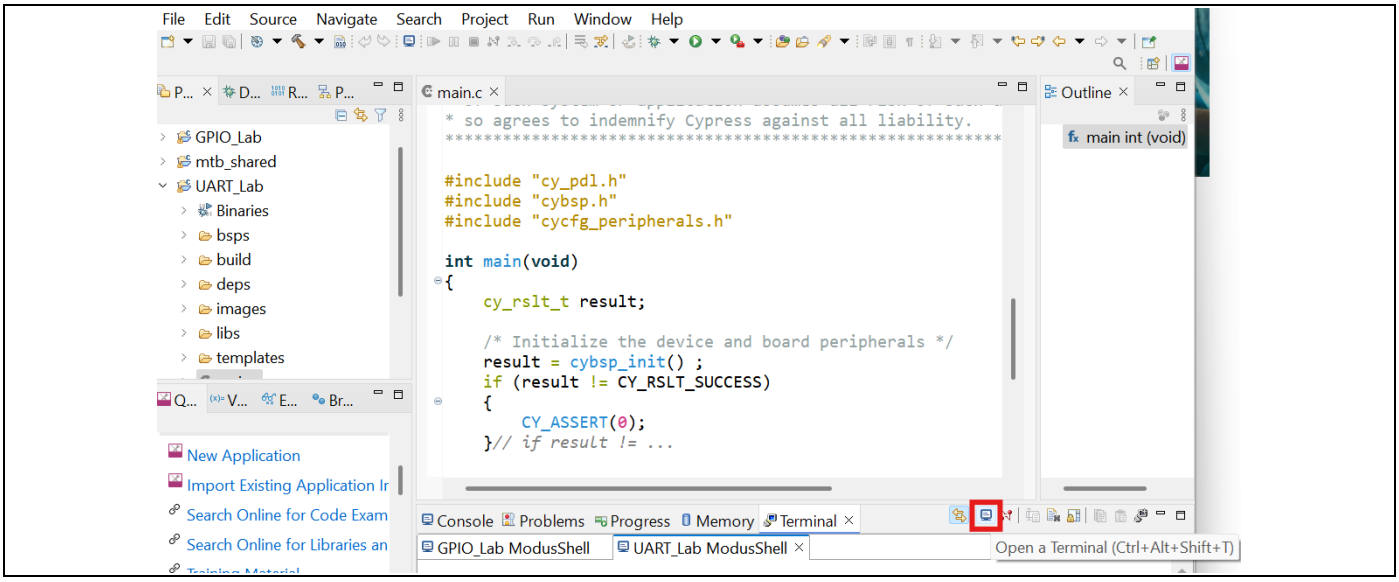



**Figure 40 Opening a new serial terminal in Eclipse IDE**

9. Select the serial port corresponding with your evaluation kit and configure the terminal

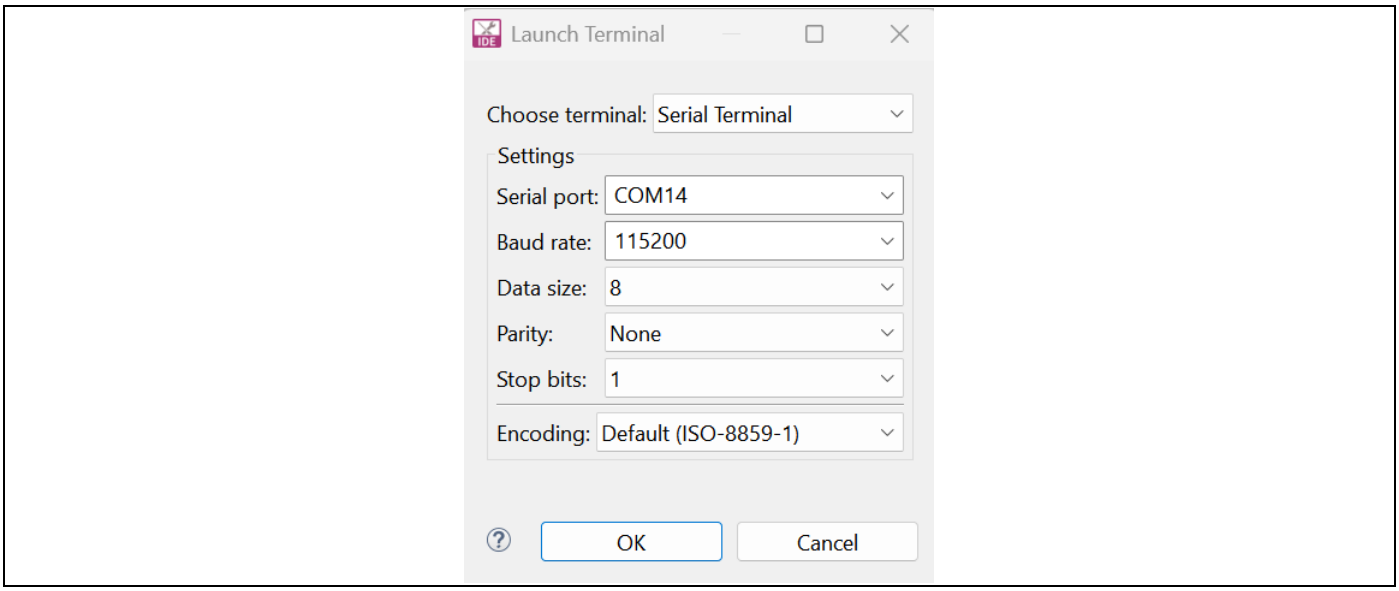



**Figure 41 Configuring the serial terminal option**

10. Program the device

Note that on the PSOC™ 4000T multi-sense kit (CY8CKIT-040T-MS), the SW2 must be in the "**UART**" position.

11. Navigate to the new terminal and observe the output

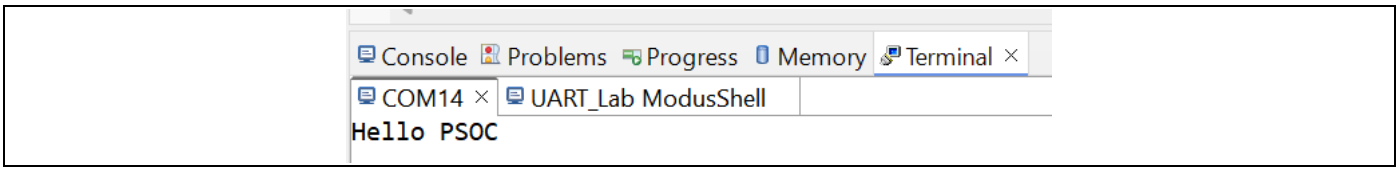



**Figure 42 Output of program to terminal for SCB UART PDL put string**

## 4.4 Receiving data from UART

Most terminals will not echo data typed into them by default. This means that when sending characters to a device using a serial terminal, the user won't get instant feedback that what they have sent has been received by the device. Let's read data from the UART RX FIFO and print it back out to the terminal so that the user can see what they have typed in real time and also see that the device has received the character.

1. Navigate back to the SCB UART PDL documentation and find the **Cy_SCB_UART_Get**, **Cy_SCB_UART_GetNumInRxFifo**, and **Cy_SCB_UART_ClearRxFifo** APIs.

    a. The **Cy_SCB_UART_Get** API is a non-blocking function to retrieve data from the SCBs FIFO. If the FIFO is empty, then the **Get** API will return a value of 0xFFFFFFFF. For this reason, it is best to check how many bytes are in the FIFO by invoking the **Cy_SCB_UART_GetNumInRxFifo** API before calling the **Get** API. Each character can be echoed, with a check for carriage returns to add a new line command. Once the FIFO has been processed, clear the RX FIFO with **Cy_SCB_UART_ClearRxFifo**.

2. Update the main for loop in the main function to poll the FIFO status, read the data, then echo the data back

```
uint32_t char_received;
for (;;)
{
        uint32_t data_count = Cy_SCB_UART_GetNumInRxFifo(CYBSP_UART_HW);

        for (uint32_t i = 0; i < data_count; i++)
        {
                char_received = (uint32_t)Cy_SCB_UART_Get(CYBSP_UART_HW);

                Cy_SCB_UART_Put(CYBSP_UART_HW, char_received);

                /* If the char_received is a carriage return "\r", then add in a new line "\n". */
                if (13 == (char) char_received)
                {
                        Cy_SCB_UART_Put(CYBSP_UART_HW, 10);
                }
        }

        if (0UL < data_count)
        {
                Cy_SCB_UART_ClearRxFifo(CYBSP_UART_HW);
        }
}
```

**Figure 43 Updates to the main function for loop to echo received characters**

3. Program the device and observe the output. As you enter new characters, they will be echo' d back on the terminal

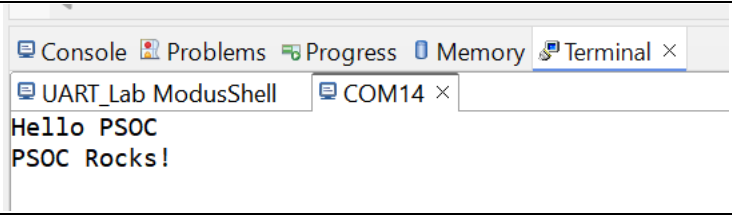



**Figure 44 Output of program to terminal for polled echoes**

## 4.5 Enabling interrupts for UART

A more efficient approach is to leverage UART interrupts to handle character reception. This eliminates the need to poll for data and ensures that characters are processed as soon as they are received.

1. Create a UART ISR to read a character from the RX FIFO and put it back on the TX FIFO

```
void uart_isr(void)
{
        uint32_t interrupt_status = Cy_SCB_GetRxInterruptStatusMasked(CYBSP_UART_HW);

        if (interrupt_status & CY_SCB_RX_INTR_NOT_EMPTY)
        {
                char char_received = Cy_SCB_UART_Get(CYBSP_UART_HW);

                Cy_SCB_UART_Put(CYBSP_UART_HW, char_received);

                /* If the char_received is a carriage return "\r", then add in a new line "\n" */
                if (13 == char_received)
                 {
                        Cy_SCB_UART_Put(CYBSP_UART_HW, 10);
                }

                Cy_SCB_ClearRxInterrupt(CYBSP_UART_HW, CY_SCB_RX_INTR_NOT_EMPTY);
        }
}
```

**Figure 45 UART ISR**

Note that the ISR gets the interrupt status, then checks that the **RX_INTR_NOT_EMPTY** interrupt is set, then uses the **Cy_SCB_UART_Get** API to read a byte from the RX FIFO. Then it puts the read byte onto the TX FIFO using **Cy_SCB_UART_Put** API. It does a check to see if a carriage return was received and if so, it sends out a new line character. Then before leaving the ISR, it clears the interrupt.

2. Enable the interrupt in the main function

```
/* Enable global interrupts */
__enable_irq();

cy_stc_sysint_t uart_intr_cfg = {
        CYBSP_UART_IRQ,
        3U,
};

Cy_SysInt_Init(&uart_intr_cfg, uart_isr);
NVIC_EnableIRQ(uart_intr_cfg.intrSrc);

Cy_SCB_SetRxInterruptMask(CYBSP_UART_HW, CY_SCB_RX_INTR_NOT_EMPTY);

Cy_SCB_UART_Enable(CYBSP_UART_HW);
Cy_SCB_UART_PutString(CYBSP_UART_HW, "Echo with intterupts\r\n");
```

**Figure 46 Enabling the UART interrupt**

4. Remove all code from the main function for loop

5. Program the device and observe the output. As you enter new characters, they will be echoed back on the terminal

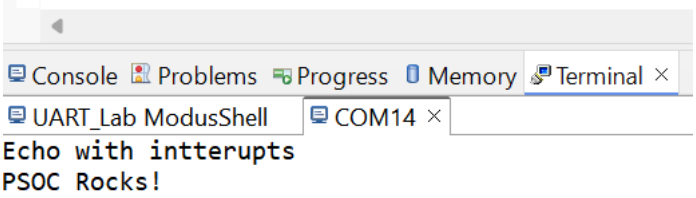



**Figure 47 Output of program to terminal for interrupt driven echoes**

## 4.6 Enabling retarget-IO middleware

Printing messages to a serial terminal emulator window on a computer is a common use case for UART (e.g. for printing debug messages). ModusToolbox™ provides a library called retarget-io to simplify the process by retargeting the standard inputs and outputs to the SCB UART target. This allows you to use standard C functions such as printf for printing messages over UART.

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

1. Open the library manager from the Eclipse IDE Quick Panel
2. Press the **Add Library** button

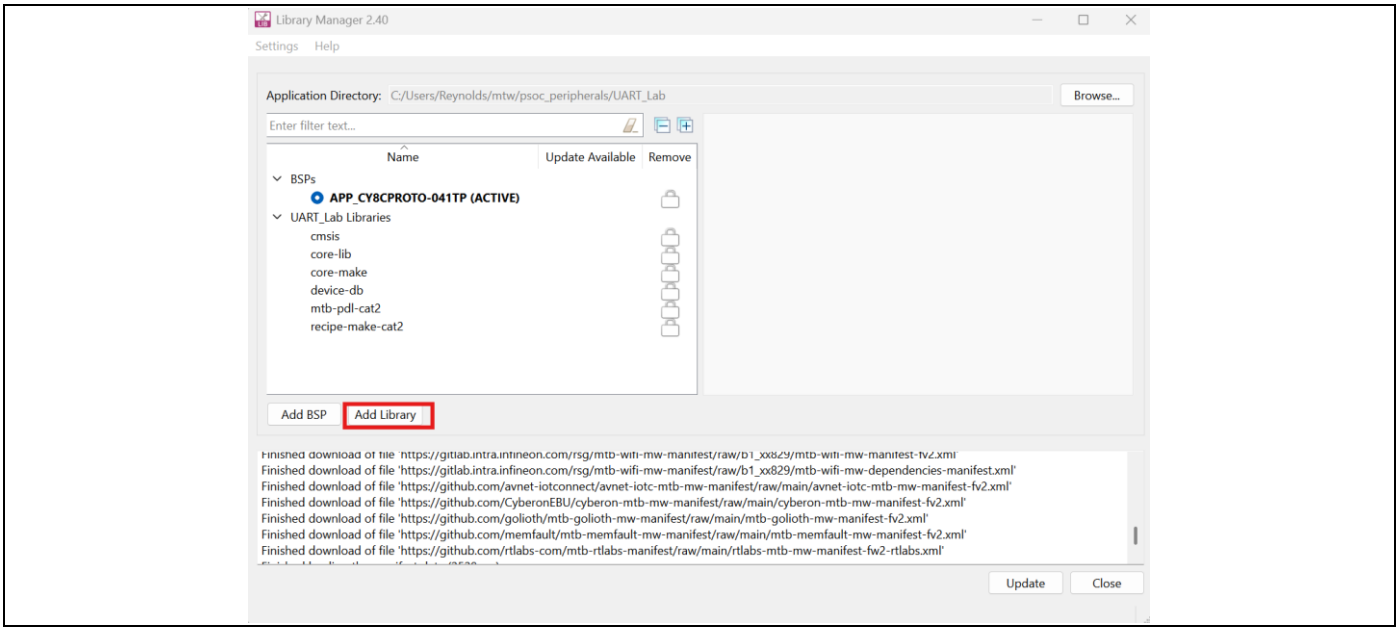



**Figure 48 Library Manager**

3. Use the search bar to find **retarget**, select the check box, then press **Ok**

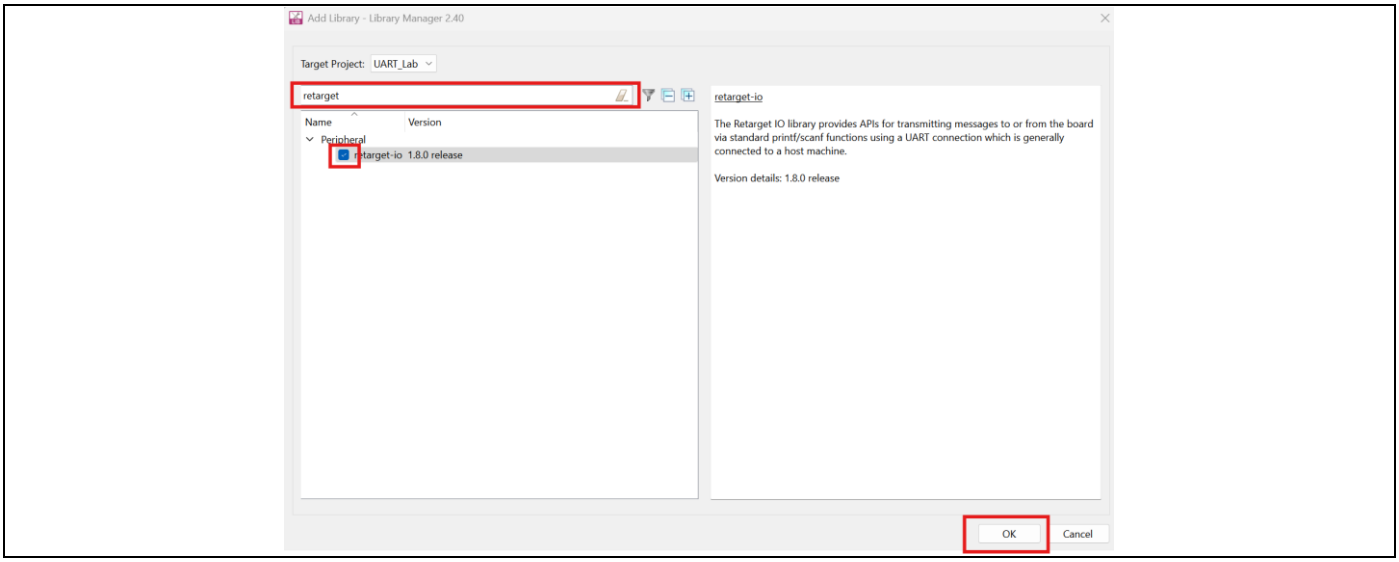



**Figure 49 Adding the retarget-IO library**

4. The library will be added to the project libraries. To update the project, press **Update** and wait for it to complete, then close the library manager
5. Navigate to the retarget-IO documentation, which is now populated in the projects Documentation in the Eclipse IDE quick panel


**Figure 50 Retarget-IO documentation for use with PDLs**

6. Remove all source code relating to the interrupts from the previous lab step, then add in the source code show here
    a. Please see the appendix for using the MTB-HAL for PSOC™ Edge devices

```
#include "cy_retarget_io.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);
        }

        Cy_SCB_UART_Init(CYBSP_UART_HW, &CYBSP_UART_config, NULL);

        /* Enable global interrupts */
        __enable_irq();

        Cy_SCB_UART_Enable(CYBSP_UART_HW);

        cy_retarget_io_init(CYBSP_UART_HW);

        printf("Hello from retarget-IO\r\n");
```

**Figure 51 Updated source code for retarget-IO use**

7. Program the device
8. Observe the new output in the terminal

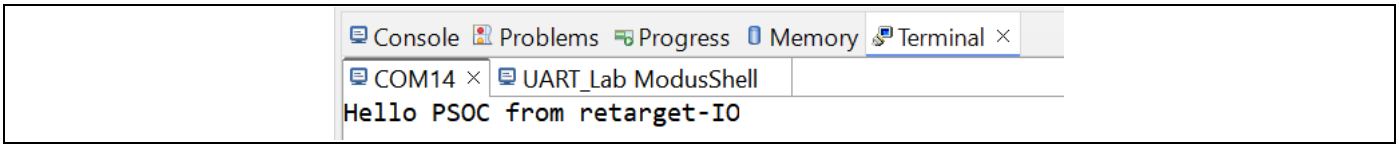



**Figure 52 Output of program to terminal**

## 4.7 Conclusion

This lab demonstrated how to enable a UART interface using the device configurator and the PDL libraries as well as how to enable the retarget-IO library for routing standard C printf functions to the UART interface for easier debug messaging.

# 5 Timer-Counter-PWMs

## 5.1 Objective

The objective of this lab is to demonstrate how to enable and configure the TCPWM blocks for different use cases from a simple timer to signal duty cycle measurement.

## 5.2 Description

The TCPWM or Timer - Counter - Pulse Width Modulator, is a highly configurable peripheral that can interface with other hardware blocks like GPIOs, intelligent analog blocks, DMA, SCBs, and SMART-I/O. PSOC™ MCUs typically come with multiple TCPWM blocks. Depending on the device, there may be 8-, 16-, or 32-bit TCPWMs. The labs will cover enabling and configuring the TCPWM block to accomplish some common application use cases for the block.

1. Timer-compare counter
2. PWM
3. Timer-capture for event counting
4. Timer-capture for event measuring

Please note that only the timer-compare counter and PWM labs steps are supported on the PSOC™ 4000T multi-sense kit (CY8CKIT-040T-MS) as that device only has two TCPWM counters. The TCPWM counters on that device support the functionality shown in the event counting and event measuring labs, but the steps described in these labs require 4 TCPWM counters.

## 5.3 Timer-compare counter

The timer functionality increments or decrements a counter between 0 and the value stored in the PERIOD register. When the counter is running, the count value stored in the COUNTER register is compared with the compare/capture register (CC). When the counter changes from a state in which COUNTER equals CC, the cc_match event is generated. This event can be used to generate an interrupt.

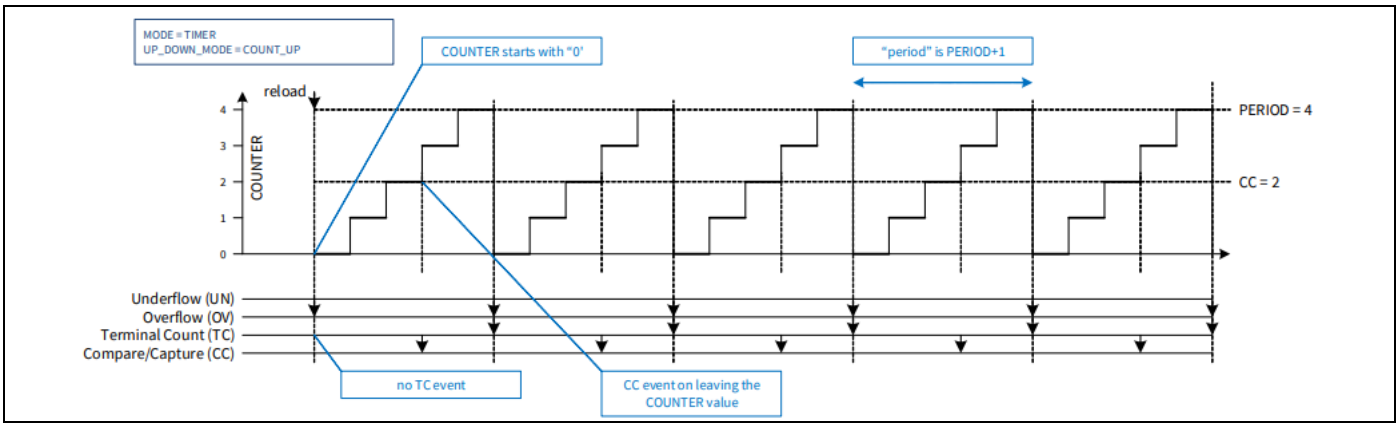



**Figure 53 Timer in up-counting, continuous mode**

The counter increments on every pulse of the input clock. When using continuous mode, the counter will increment until the count reaches the value in the PERIOD register. Once the counter reaches the value in the COMPARE register, an event is generated, which can be configured as an interrupt. By adjusting the input clock, PERIOD, and COMPARE registers, a timer can be generated that fires an interrupt on a set period. This can be used for tasks like time keeping, triggering time sensitive events, and more.

Let's create a timer that generates an interrupt every second. We will use this interrupt to toggle a GPIO in the application code.

1. The project created in the first exercise can be used, or the instructions shown there can be used to create a new project.

2. Once the project has been opened, navigate to the Eclipse IDE Quick Panel and open the Device Configurator. Navigate to the **Peripherals** tab and enable a **TCPWM 16-bit Counter** by toggling the checkbox, selecting the **Counter** personality, then pressing OK.

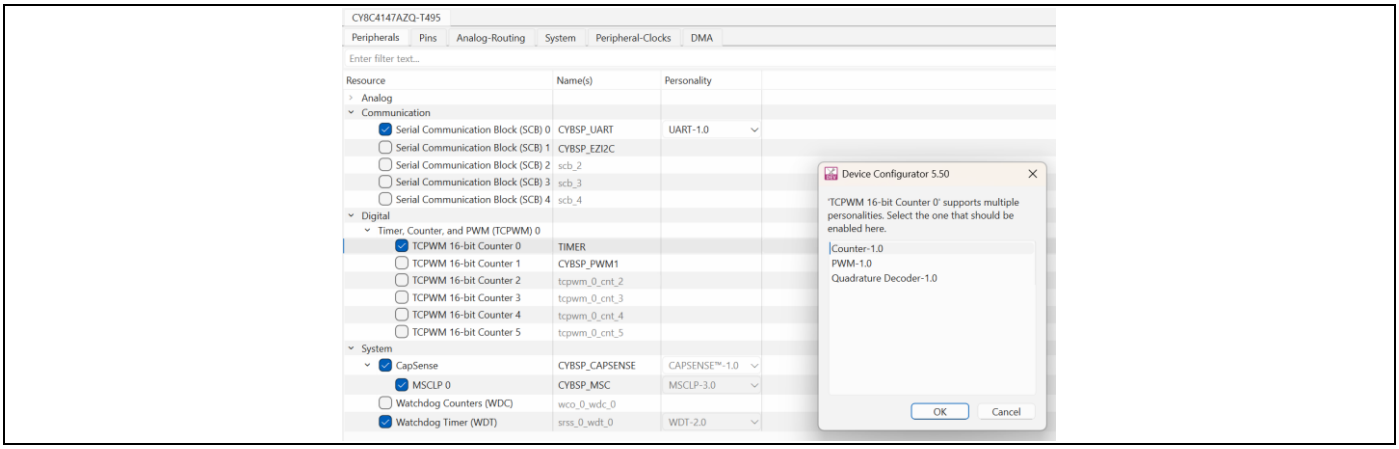



**Figure 54 Enable TCPWM 16-bit Counter in the Device Configurator**

3. You may rename the alias of the block to "TIMER" to ensure that the following code matches the alias

4. Configure the TCPWM for **Compare**, set the **Period** and **Compare** values, select **Compare & Capture** as the interrupt source, and select a **Clock Signal**

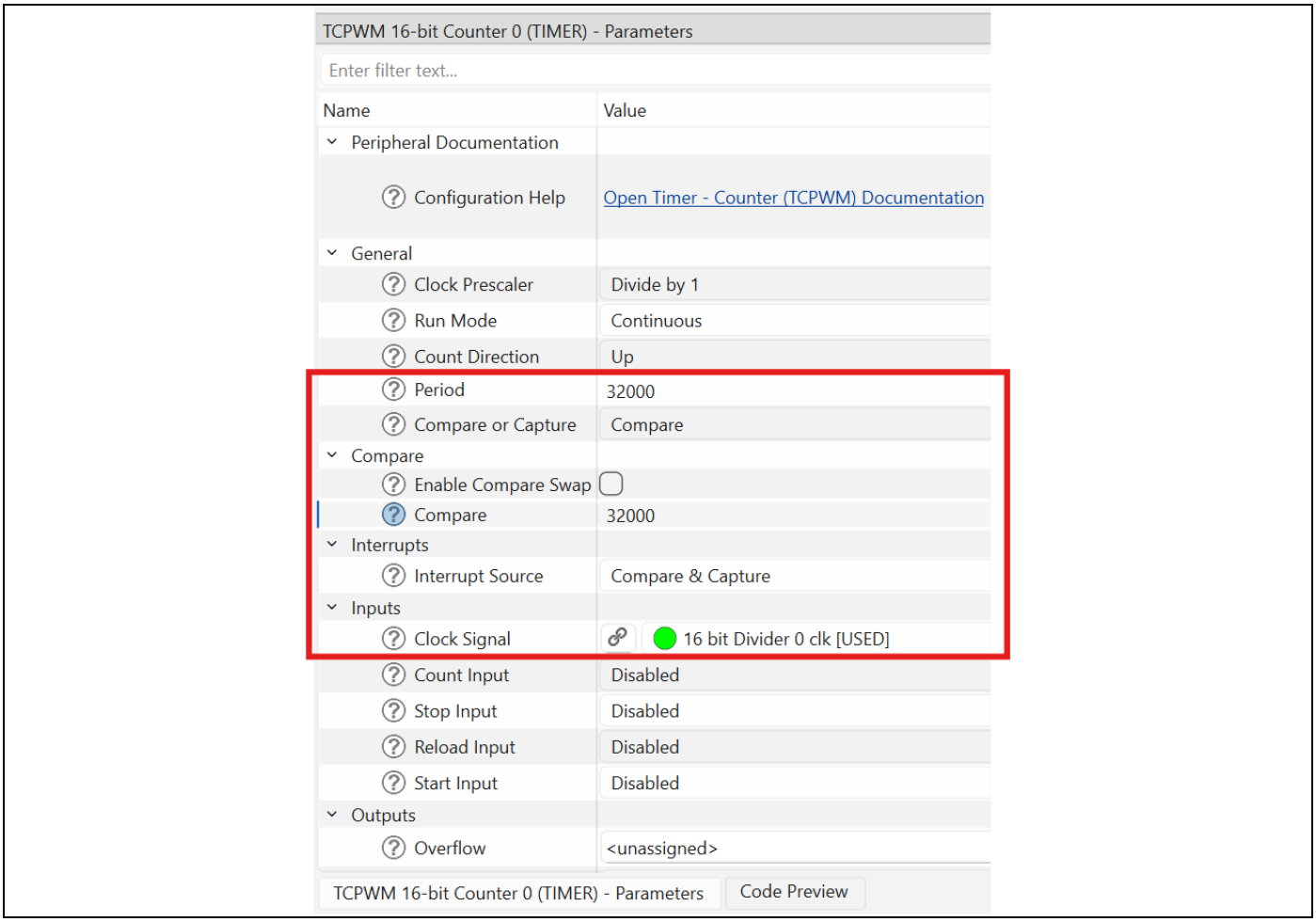



**Figure 55 Configure the TCPWM 16-bit Counter in the Device Configurator as a Compare Timer**

5. The goal for this lab step is to create a Timer that triggers every second. To accomplish this, the peripheral clock input must allow for the compare interrupt to trigger every second. This means that the clock signal must be set to 32KHz. To do this, navigate to the **Peripheral-Clocks** tab and open the clock signal selected for the TCPWM instance and adjust it to 32 KHz

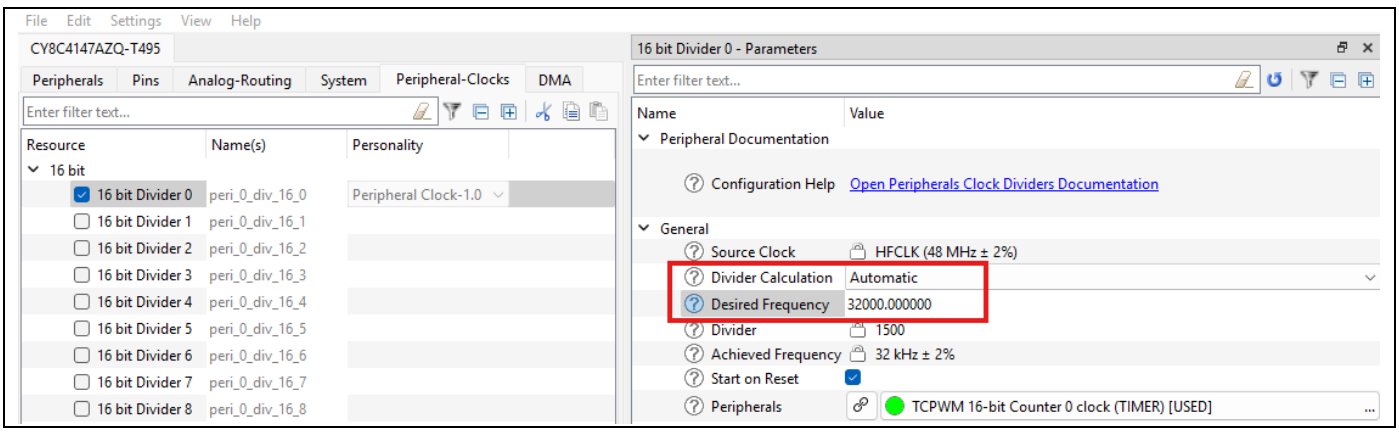



**Figure 56 Configure the peripheral clock to meet the timing requirements of the timer**

6. Save the device configurator

7. Navigate back to the **Peripherals** tab, click on the TCPWM that you enabled, then open the documentation link found at the top of the Parameters tab
8. Navigate to the **Functions** module and click on the **Cy_TCPWM_Counter_Init** function and observe the **Function Usage**
9. The device configurator has created the config structure, an alias for the **TCPWM_Type** base, the **NUM**, and **MASK** definitions. Copy the functions shown here into your main function before the **enable_irq** function, but after the **cybsp_init** function and modify the inputs to match the generated source from the device configurator

```
if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(TIMER_HW, TIMER_NUM, &TIMER_config))
{
        /* Handle possible errors */
}
/* Enable the initialized counter */
Cy_TCPWM_Counter_Enable(TIMER_HW, TIMER_NUM);

/* Then start the counter */
Cy_TCPWM_TriggerStart(TIMER_HW, TIMER_MASK);
```

**Figure 57 Enabling the timer in application code for PSOC™ 4 and PSOC™ 6**

```
if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(TIMER_HW, TIMER_NUM, &TIMER_config))
{
        /* Handle possible errors */
}
/* Enable the initialized counter */
Cy_TCPWM_Counter_Enable(TIMER_HW, TIMER_NUM);

/* Then start the counter */
Cy_TCPWM_TriggerStart_Single(TIMER_HW, TIMER_NUM);
```

**Figure 58 Enabling the timer in application code for PSOC™ Edge**

10. In the TCPWM configuration we enabled compare and capture interrupts, so let's add the source code to enable an ISR for the interrupt. Navigate back to the PDL documentation and find the **SysInt** libraries. We will need to setup the config structure for the interrupt, pass the ISR, and enable the interrupt.

```
cy_stc_sysint_t timer_intr_cfg =
{
        /*.intrSrc =*/ TIMER_IRQ,
        /*.intrPriority =*/ 3UL
};

Cy_SysInt_Init(&timer_intr_cfg, timer_isr);

/* Enable the interrupt */
NVIC_EnableIRQ(timer_intr_cfg.intrSrc);
```

**Figure 59 TCPWM Interrupt Initialization**

11. The ISR function must clear the interrupt source. The TCPWM PDL has an API to get and clear interrupts for the block

```
void timer_isr(void)
{
        Cy_TCPWM_ClearInterrupt(TIMER_HW, TIMER_NUM, Cy_TCPWM_GetInterruptStatus(TIMER_HW, TIMER_NUM));
}
```

**Figure 60 TCPWM ISR function**

12. The last step is to enable a GPIO that is connected to an LED, the steps in the GPIO lab for GPIO control can be used to configure the pin, then the **Cy_GPIO_Inv** API can be used to toggle the GPIO. Add that to the TCPWM ISR

```
void timer_isr(void)
{
        Cy_GPIO_Inv(CYBSP_USER_LED1_PORT, CYBSP_USER_LED1_PIN);
        Cy_TCPWM_ClearInterrupt(TIMER_HW, TIMER_NUM, Cy_TCPWM_GetInterruptStatus(TIMER_HW, TIMER_NUM));
}
```

**Figure 61 TCPWM ISR function with GPIO Invert API**

13. Program the device and observe the LED blinking every second

## 5.4 PWM

The PWM can output left, right, center, or asymmetrically aligned PWM. The PWM signal is generated by incrementing or decrementing a counter between 0 and PERIOD, and comparing the counter value COUNTER with CCx. When COUNTER equals CCx, the ccx_match event is generated. The pulse-width modulated signal is then generated by using the ccx_match event along with overflow and underflow events. The ccx_match event used for PWM generation is dictated by the compare swap feature, when the compare swap value is set to 0, the cc0_match event is used, when it is set to 1, the cc1_match event is used. Two pulse-width modulated signals “line_out” and “line_compl_out” are output from the PWM.

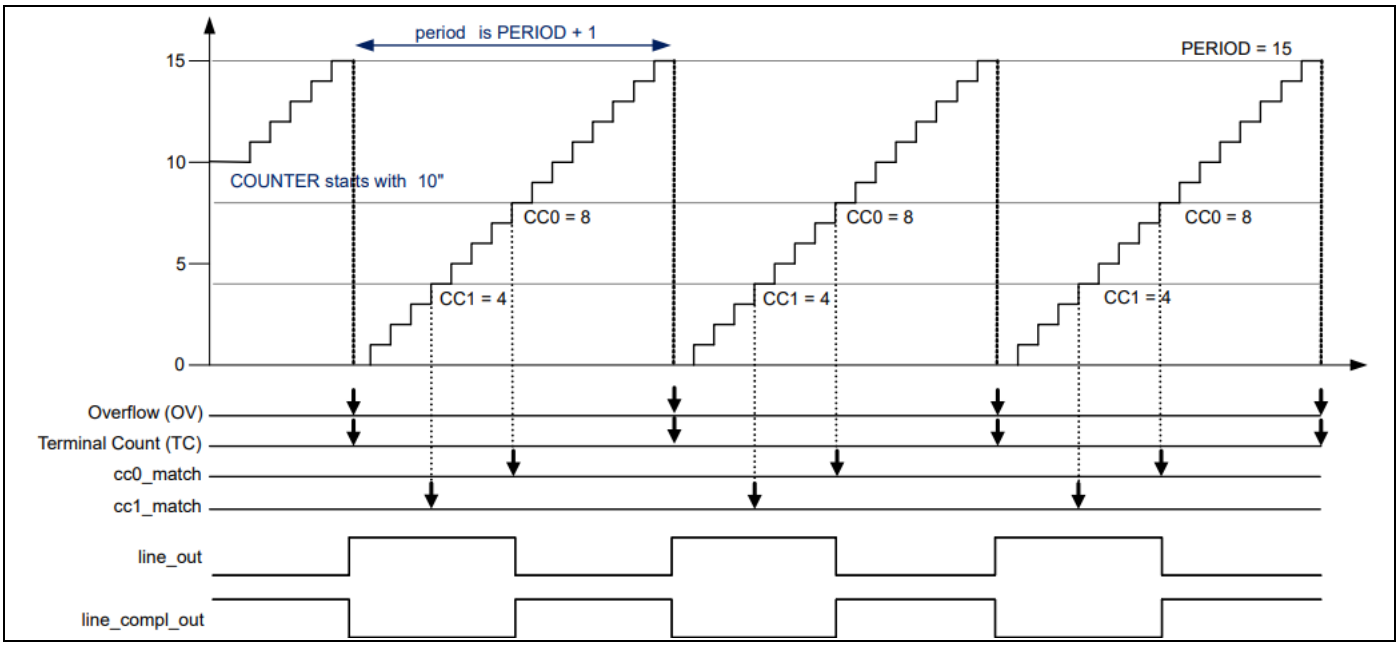



**Figure 62 TCPWM used for PWM generation on line_out**

By changing the PERIOD and COMPARE (CCx) registers, the frequency and duty cycle of the PWM can be adjusted.

## Timer-Counter-PWMs

In this portion of the lab, we will add another TCPWM counter to generate a PWM on a GPIO connected to an LED on the evaluation kit.

1. Navigate to the pins tab of the device configurator
2. Find a user LED GPIO and enable it, then connect the internal **Digital Output** connection to a TCPWM counter

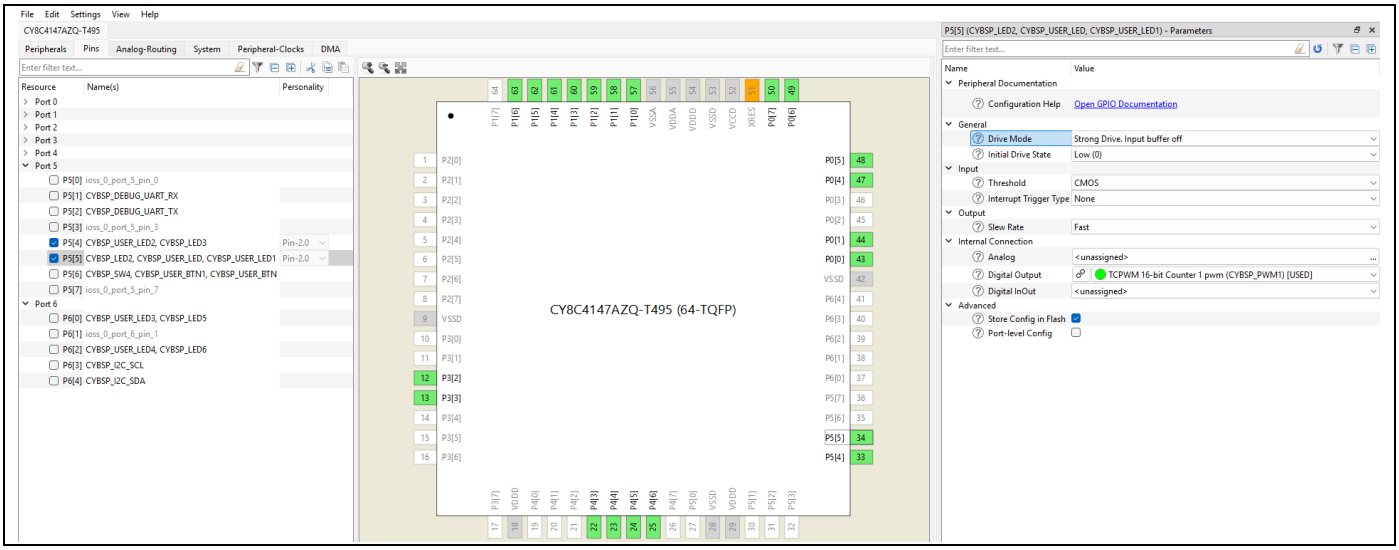



**Figure 63 GPIO configuration for use with TCPWM block as PWM_line**

3. Navigate to the **Peripherals** tab, set the alias of the TCPWM counter associated with the GPIO that was just enabled to **PWM** and enable it by toggling the check box, selecting the **PWM** personality, then pressing OK.

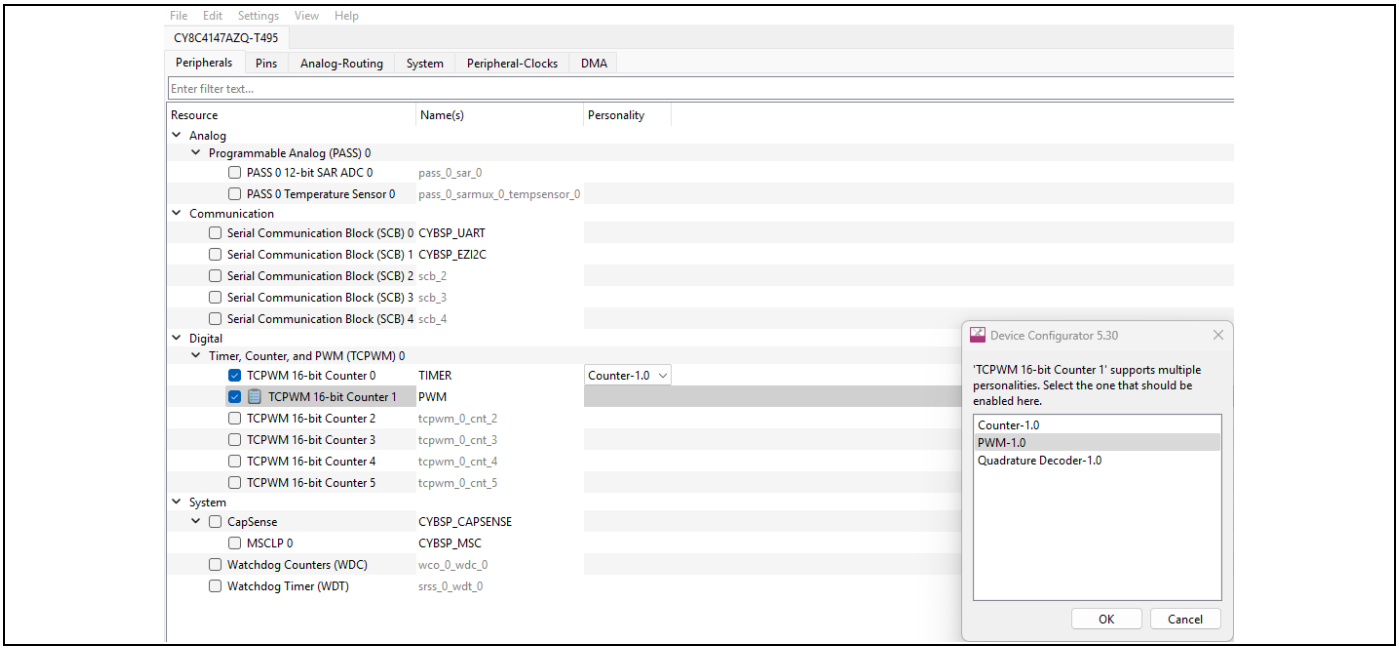



**Figure 64 Enabling the TCPWM counter for the associated GPIO**

## Timer-Counter-PWMs

4. Configure the TCPWM to generate a 5 Hz PWM with a 50% duty cycle. Using the same peripheral clock divider as the previously enabled TIMER, which is running at 32 KHz. Set the period to 6400 and set the compare half of that to achieve the 50% duty cycle. The frequency of the PWM is 32 KHz divided by the period (6400), which is 5 Hz.

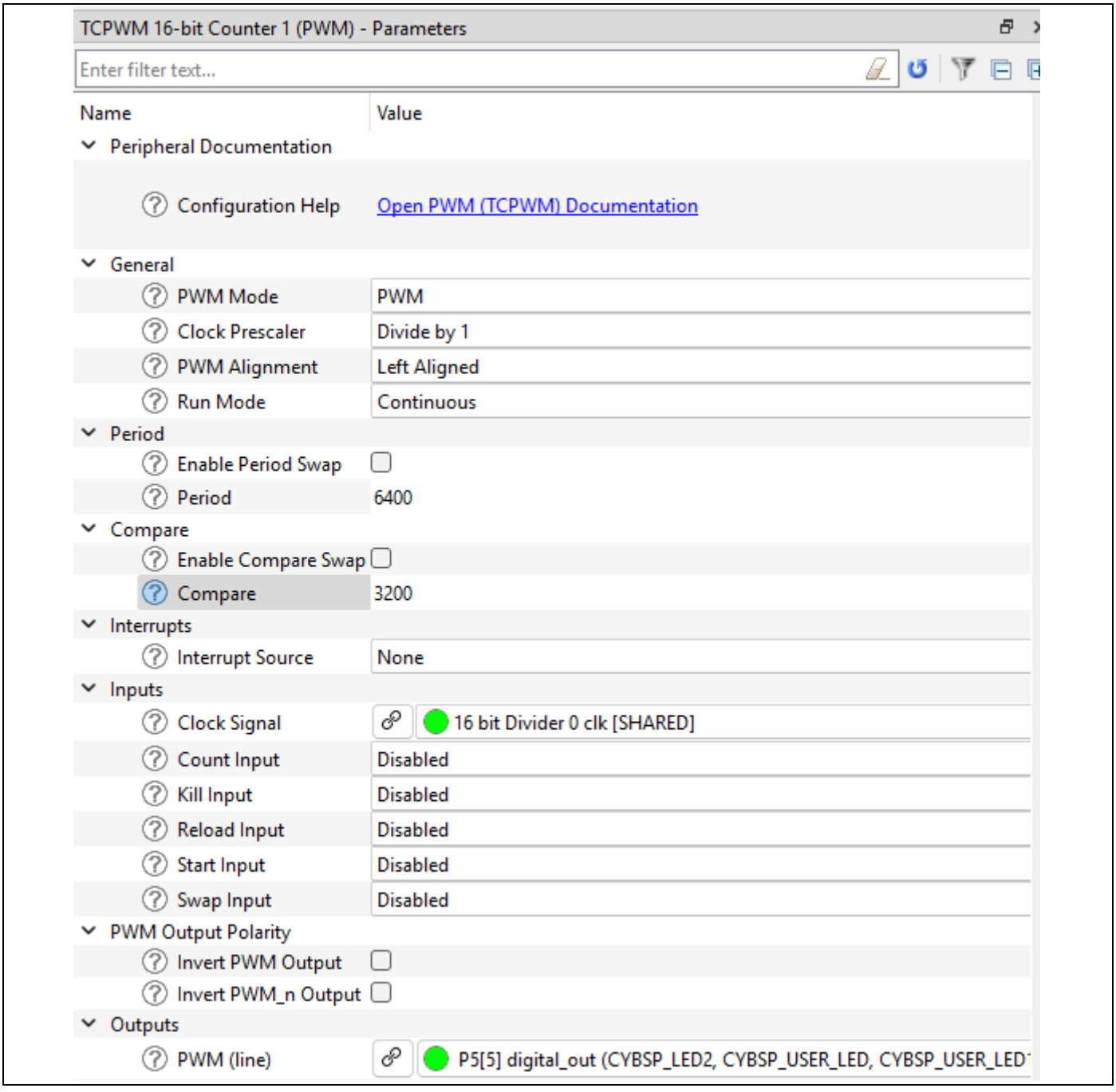



**Figure 65 TCPWM configuration for 5 Hz PWM with a 50% duty cycle**

Note that on some evaluation kits, the LEDs used may need to be modified from the instructions in the lab manual. For example, the PSOC™ 4100S Max kit (CY8CKIT-041S-MAX) is configured such that LED2 is associated with the TCPWM counter 6. Also, some kits will have LEDs associated with TCWPM PWM_N lines. This will also work, but the duty cycle will be inverted.

5. Save the device configurator settings

6. Open the **PWM (TCPWM) Documentation** and find the **Cy_TCPWM_PWM_INIT** API documentation
7. We will use the functions to initialize, enable, and start the PWM. Note that in step three, the alias of the TCPWM counter was set to PWM. Add the code into the main function before the **enable_irq** function call

```
if (CY_TCPWM_SUCCESS != Cy_TCPWM_PWM_Init(PWM_HW, PWM_NUM, &PWM_config))
{
        /* Handle possible errors */
}

/* Enable the initialized counter */
Cy_TCPWM_PWM_Enable(PWM_HW, PWM_NUM);

/* Then start the counter */
Cy_TCPWM_TriggerReloadOrIndex(PWM_HW, PWM_MASK);
```

**Figure 66 TCPWM PWM initialization in user code for PSOC™ 4 and PSOC™ 6**

```
if (CY_TCPWM_SUCCESS != Cy_TCPWM_PWM_Init(PWM_HW, PWM_NUM, &PWM_config))
{
        /* Handle possible errors */
}

/* Enable the initialized counter */
Cy_TCPWM_PWM_Enable(PWM_HW, PWM_NUM);

/* Then start the counter */
Cy_TCPWM_TriggerStart_Single(PWM_HW, PWM_NUM);
```

**Figure 67 TCPWM PWM initialization in user code for PSOC™ Edge**

8. Build and program the device. Observe the first LED blinking every second as the timer enabled is still running, and the second LED is blinking at a 5 Hz rate
9. You can now adjust the period or duty cycle of the PWM using the **Cy_SetPeriod0** and **Cy_SetCompare0** APIs

## 5.5 Timer-capture for event counting

When the count input is configured as rising/falling the count value is changed on each prescaled clk_counter edge in which an edge is detected on the count input. Using this capability the number of pulses on a GPIO can be counted. If this count is measured over a set period of time, the frequency of the pulses can be measured.

In the following steps, we will add another TCPWM that monitors the pulses of the PWM (Falling Edge) with the count input.

1. Navigate to the **Peripherals** tab of the device configurator
2. Set the alias for another TCPWM counter to **EVENT_COUNTER** and enable it by toggling the checkbox, then selecting **Counter** as the personality and pressing OK.

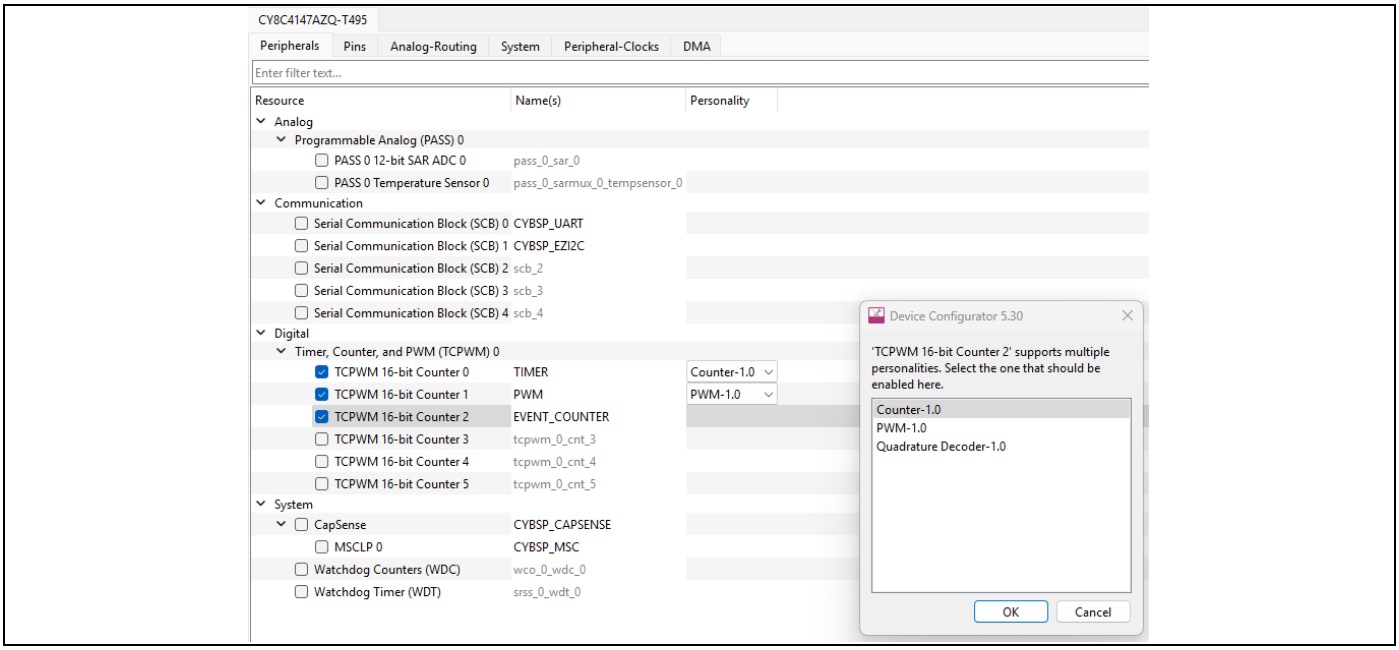
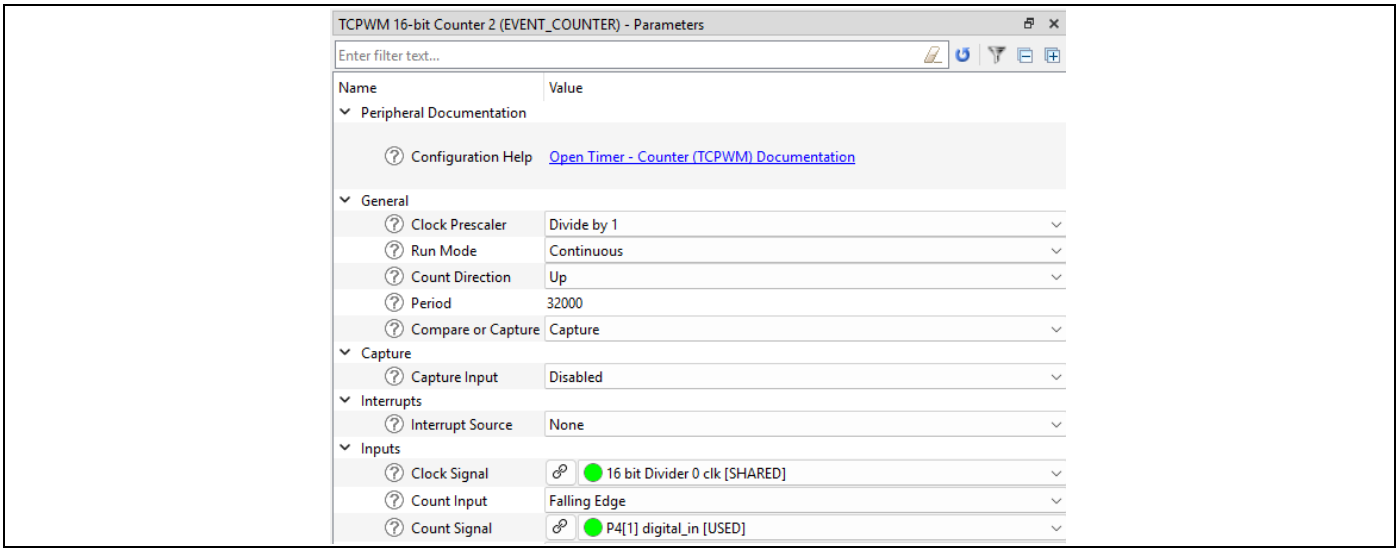



**Figure 68 Enabling the TCPWM PWM event counter**

3. Using the same peripheral clock as the timer and PWM and a period of 32000, the TCPWM counter will count events on a given GPIO and store them in the counter register. Configure the **Count Signal** for **Falling Edge** and select an available GPIO



## Timer-Counter-PWMs

**Figure 69 TCPWM PWM event counter configuration in the device configurator**

4. Configure the Count Signal GPIO, including setting the pin alias to **FEEDBACK_PIN**

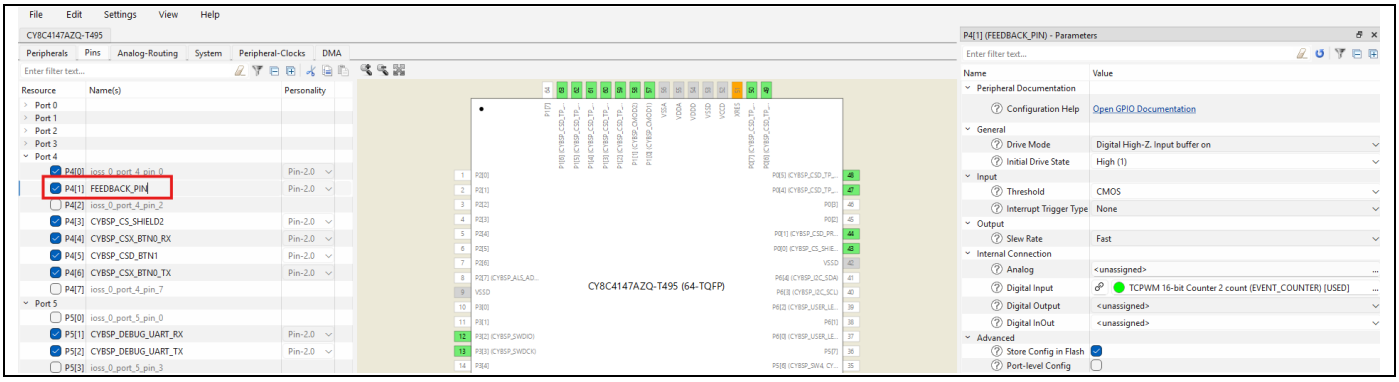



**Figure 70 Count signal GPIO configuration and alias**

5. Save the device configurator settings
6. Open the **Timer – Counter (TCPWM) Documentation** and navigate to **Common → Functions** and see the documentation for **Cy_TCWPM_Enable_Multiple** and **Cy_TCWPM_TriggerReloadOrIndex**. Notice that these APIs allow the user code to enable and start multiple TCPWM counters using the aliased masks. We will use these to enable and start all TCPWM counters for this project. These are only available for PSOC™ 4 and PSOC™ 6.
7. Add the initialization, enable, and start APIs for the new TCPWM block, aliased as **EVENT_COUNTER**

```
if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(EVENT_COUNTER_HW, EVENT_COUNTER_NUM,
&EVENT_COUNTER_config))
{
        /* Handle possible errors */
}

/* Enable the initialized counters */
Cy_TCPWM_Enable_Multiple(PWM_HW, TIMER_MASK | PWM_MASK | EVENT_COUNTER_MASK);
/* Then start the counters */
Cy_TCPWM_TriggerReloadOrIndex(PWM_HW, TIMER_MASK | PWM_MASK | EVENT_COUNTER_MASK);
```

**Figure 71 TCPWM EVENT_COUNTER initialization in user code for PSOC™ 4**

```
if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(EVENT_COUNTER_HW, EVENT_COUNTER_NUM,
&EVENT_COUNTER_config))
{
        /* Handle possible errors */
}

/* Enable the initialized counters */
Cy_TCPWM_Enable_Multiple(PWM_HW, TIMER_MASK | PWM_MASK | EVENT_COUNTER_MASK);
Cy_TCPWM_Enable_Multiple(EVENT_COUNTER_HW, EVENT_COUNTER_MASK);

/* Then start the counters */
Cy_TCPWM_TriggerReloadOrIndex(PWM_HW, TIMER_MASK | PWM_MASK | EVENT_COUNTER_MASK);
Cy_TCPWM_TriggerReloadOrIndex(EVENT_COUNTER_HW, EVENT_COUNTER_MASK);
```

**Figure 72 TCPWM EVENT_COUNTER initialization in user code for PSOC™ 6**

For PSOC™ 6, a second TCPWM block is used because the counter input needed is not available on the first TCPWM block. This means that when enabling multiple counters, the second block requires its own function call.

```
if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(EVENT_COUNTER_HW, EVENT_COUNTER_NUM,
&EVENT_COUNTER_config))
{
        /* Handle possible errors */
}

/* Enable the initialized counter */
Cy_TCPWM_Counter_Enable(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);

/* Then start the counter */
Cy_TCPWM_TriggerStart_Single(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);
```

**Figure 73 TCPWM EVENT_COUNTER initialization in user code for PSOC™ Edge**

8. Ensure that the previously used start/reload APIs are removed as they will be enabled and started with this new method

    a. Do this only for PSOC™ 4 and PSOC™ 6

9. As the LED GPIO may not be readily available on some evaluation kit headers without rework, another PWM output can be enabled on a pin available. See Table 1 to find the appropriate GPIO

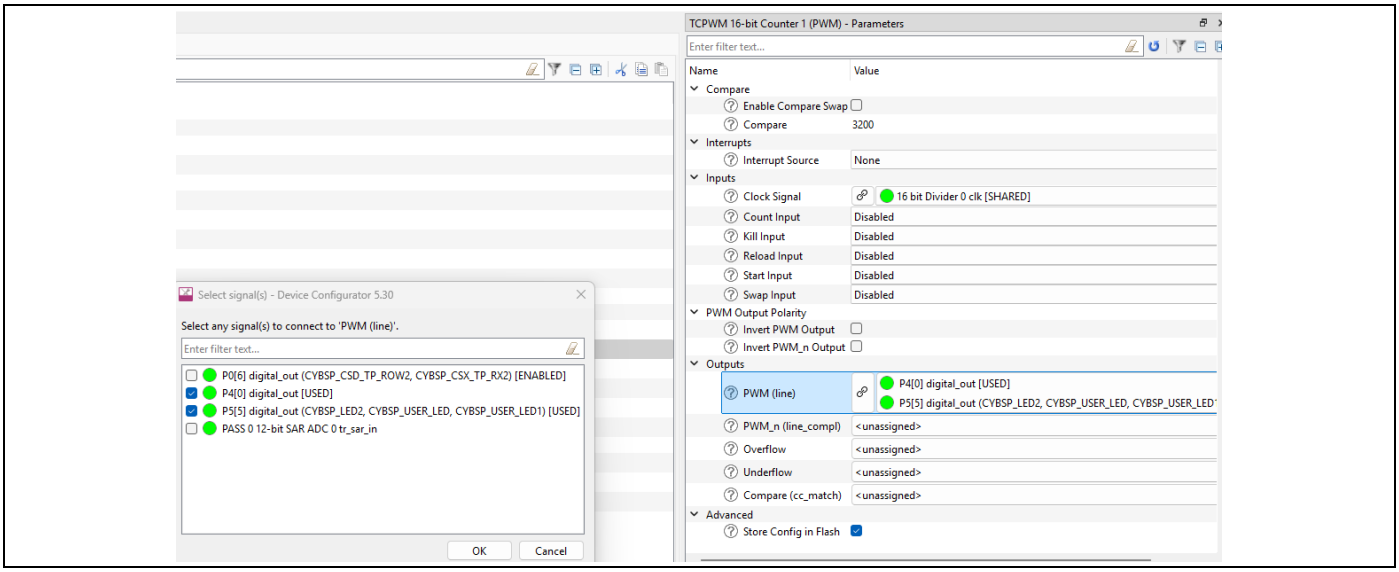



**Figure 74 Enable a second PWM output from the same TCPWM counter**

10. A jumper wire is needed to connect the PWM to be measured and the input to the capture. Connect either the LED GPIO or the second PWM GPIO to the capture GPIO using a jumper wire

**Table 1 PWM signal to capture input connection options**

| Evaluation Kit | LED GPIO | Second PWM GPIO | Counter GPIO |
|---|---|---|---|
| CY8CPROTO-041TP | P5.6 | P4.0 | P4.1 |
| CY8CKIT-041S-MAX | P6.4 | P1.4 | P1.2 |
| CY8CKIT-062S2-43012 | P11.1 | P10.7 | P9.0 |
| KIT_PSE84_EVAL | P16.6 | P13.6 | P11.1 |

11. Using the steps here, add a UART SCB to enable printing information to a terminal
    a. Note that it's best that the TCPWMs and the UART don't share a peripheral clock, as the UART peripheral uses the peripheral clock to configure the baud rate.
12. Add the stdio.h include to the main.c file

```
#include <stdio.h>
```

**Figure 75 Standard input output reference library**

13. Add a Boolean flag variable in the timer_isr to indicate that the timer has expired

```
volatile bool timer_flag = false;

void timer_isr(void)
{
        timer_flag = true;
        Cy_GPIO_Inv(CYBSP_USER_LED2_PORT, CYBSP_USER_LED2_PIN);
        Cy_TCPWM_ClearInterrupt(TIMER_HW, TIMER_NUM, Cy_TCPWM_GetInterruptStatus(TIMER_HW, TIMER_NUM));
}
```

**Figure 76 Timer flag variable in the timer_isr**

14. In the infinite for loop of the main function, check that this flag has been set, if it has, clear the flag, then read the EVENT_COUNTER count value and print it to the terminal

```
for (;;)
{
        if (true == timer_flag)
        {
                timer_flag = false;
                char print[8];
                uint32_t event_count = Cy_TCPWM_Counter_GetCounter(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);
                Cy_TCPWM_Counter_SetCounter(EVENT_COUNTER_HW, EVENT_COUNTER_NUM, 0);
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "**********\n\r");
                sprintf(print, "%u\r\n", event_count);
                Cy_SCB_UART_PutString(CYBSP_UART_HW, print);
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "**********\n\r");
        }
}
```

**Figure 77 Reading and printing the event counter**

15. Program the device, open a terminal and observe the output

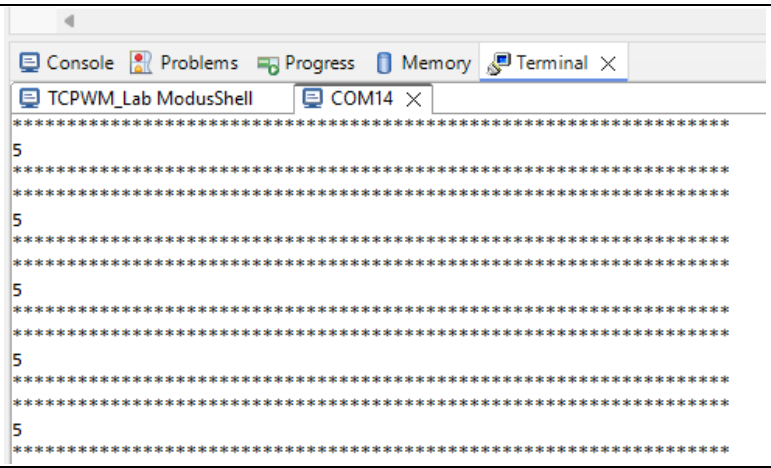



**Figure 78 Output of the counter**

## Timer-Counter-PWMs

16. The TCPWM counter is counting 5 falling edge events every second, this means that the signal is operating at a 5 Hz rate. We can use other PDL APIs to determine this frequency independent of clock and period changes that may be made to the TCPWM counter

    a. **Cy_TCPWM_Counter_GetPeriod**: This API returns the period of the TCPWM counter

    b. **Cy_SysClk_PeriphGetAssignedDivider**: This API returns the divider assigned to the peripheral. On PSOC™ 4 the format returned shows the divider type and number, where bits [7:6] = type, and bits[5:0] = divider number within that type. On PSOC™ 6 and PSOC™ Edge, the format returned shows the divider type and number, where bits [9:8] = type, and bits [7:0] = divider number within that type. It accepts an enumeration of possible peripheral clock assignments. The input enumeration is dependent on the device.

    c. **Cy_SysClk_PeriphGetFrequency**: This API returns the frequency of the peripheral clock

17. Let's add a function to print a float value

```
void print_float(float value)
{
        char print_float[20];
        int int_part = (int)value; /* Integer part */
        int frac_part = (int)((value - int_part) * 100); /* Fractional part (2 decimal places) */
        sprintf(print_float, "%d.%02d", int_part, frac_part);
        Cy_SCB_UART_PutString(CYBSP_UART_HW, print_float);
}
```

**Figure 79 Function to print a float value**

18. Update the for loop in the main function to calculate and print the signal frequency

```
/* Get the TCPWM counter period. */
uint32_t counter_period = Cy_TCPWM_Counter_GetPeriod(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);

/* Get the clock divider type and number. */
uint32_t clock_divider = Cy_SysClk_PeriphGetAssignedDivider(PCLK_TCPWM_CLOCKS1);
uint32_t divider_type = (clock_divider&PERI_PCLK_CTL_SEL_TYPE_Msk)>>PERI_PCLK_CTL_SEL_TYPE_Pos;
uint32_t divider_number = (clock_divider&PERI_PCLK_CTL_SEL_DIV_Msk)>>PERI_PCLK_CTL_SEL_DIV_Pos;

/* To divide the TCPWM clock by 1 requires the pre-scaler register to be set to 0, so add 1 to the pre-scaler. */
uint32_t prescaler = TIMER_config.clockPrescaler + 1U;

/* Determine the frequency of the clock driving the TCPWM counter. */
uint32_t clock_frequency = Cy_SysClk_PeriphGetFrequency(divider_type, divider_number)/prescaler;

/* Determine the period in seconds of the TCPWM counter. */
float counter_time = (1/(float)clock_frequency)*(float)counter_period;

for (;;)
{
        if (true == timer_flag)
        {
                timer_flag = false;
                uint32_t event_count = Cy_TCPWM_Counter_GetCounter(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);
                Cy_TCPWM_Counter_SetCounter(EVENT_COUNTER_HW, EVENT_COUNTER_NUM, 0);

                float signal_frequency = (float)event_count/counter_time;
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "*********************************************************************\n\r");
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Signal Frequency = ");
                print_float(signal_frequency);
                Cy_SCB_UART_PutString(CYBSP_UART_HW, " hz \r\n");
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "*********************************************************************\n\r");
        }
}
```

**Figure 80 Function to print a float value for PSOC™ 4**

```
/* Get the TCPWM counter period. */
uint32_t counter_period = Cy_TCPWM_Counter_GetPeriod(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);

/* Get the clock divider type and number. */
uint32_t clock_divider = Cy_SysClk_PeriphGetAssignedDivider(PCLK_TCPWM1_CLOCKS0);
uint32_t divider_type = (clock_divider&CY_PERI_CLOCK_CTL_TYPE_SEL_Msk)>>CY_PERI_CLOCK_CTL_TYPE_SEL_Pos;
uint32_t divider_number = (clock_divider&CY_PERI_CLOCK_CTL_DIV_SEL_Msk)>>  CY_PERI_CLOCK_CTL_DIV_SEL_Pos;

/* To divide the TCPWM clock by 1 requires the pre-scaler register to be set to 0, so add 1 to the pre-scaler. */
uint32_t prescaler = TIMER_config.clockPrescaler + 1U;

/* Determine the frequency of the clock driving the TCPWM counter. */
uint32_t clock_frequency = Cy_SysClk_PeriphGetFrequency(divider_type, divider_number)/prescaler;

/* Determine the period in seconds of the TCPWM counter. */
float counter_time = (1/(float)clock_frequency)*(float)counter_period;

for (;;)
{
        if (true == timer_flag)
        {
                timer_flag = false;
                uint32_t event_count = Cy_TCPWM_Counter_GetCounter(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);
                Cy_TCPWM_Counter_SetCounter(EVENT_COUNTER_HW, EVENT_COUNTER_NUM, 0);

                float signal_frequency = (float)event_count/counter_time;
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "*********************************************************************\n\r");
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Signal Frequency = ");
                print_float(signal_frequency);
                Cy_SCB_UART_PutString(CYBSP_UART_HW, " hz \r\n");
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "*********************************************************************\n\r");
        }
}
```

**Figure 81 Function to print a float value for PSOC™ 6**

```
/* Get the TCPWM counter period. */
uint32_t counter_period = Cy_TCPWM_Counter_GetPeriod(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);

/* Get the clock divider type and number. */
uint32_t clock_divider = Cy_SysClk_PeriphGetAssignedDivider(PCLK_TCPWM0_CLOCK_COUNTER_EN2);
uint32_t divider_type = (clock_divider&CY_PERI_CLOCK_CTL_TYPE_SEL_Msk)>>CY_PERI_CLOCK_CTL_TYPE_SEL_Pos;
uint32_t divider_number = (clock_divider&CY_PERI_CLOCK_CTL_DIV_SEL_Msk)>>  CY_PERI_CLOCK_CTL_DIV_SEL_Pos;

/* To divide the TCPWM clock by 1 requires the pre-scaler register to be set to 0, so add 1 to the pre-scaler. */
uint32_t prescaler = TIMER_config.clockPrescaler + 1U;

/* Determine the frequency of the clock driving the TCPWM counter. */
uint32_t clock_frequency = Cy_SysClk_PeriphGetFrequency(divider_type, divider_number)/prescaler;

/* Determine the period in seconds of the TCPWM counter. */
float counter_time = (1/(float)clock_frequency)*(float)counter_period;

for (;;)
{
        if (true == timer_flag)
        {
                timer_flag = false;
                uint32_t event_count = Cy_TCPWM_Counter_GetCounter(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);
                Cy_TCPWM_Counter_SetCounter(EVENT_COUNTER_HW, EVENT_COUNTER_NUM, 0);

                float signal_frequency = (float)event_count/counter_time;
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "**********\n\r");
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Signal Frequency = ");
                print_float(signal_frequency);
                Cy_SCB_UART_PutString(CYBSP_UART_HW, " hz \r\n");
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "**********\n\r");
        }
}
```

**Figure 82 Function to print a float value for PSOC™ Edge**

Note that the **PCLK_TCPWM_CLOCKSx** will vary based on the TCPWM block used for the event counter.

19. Program the device and observe the output in the terminal

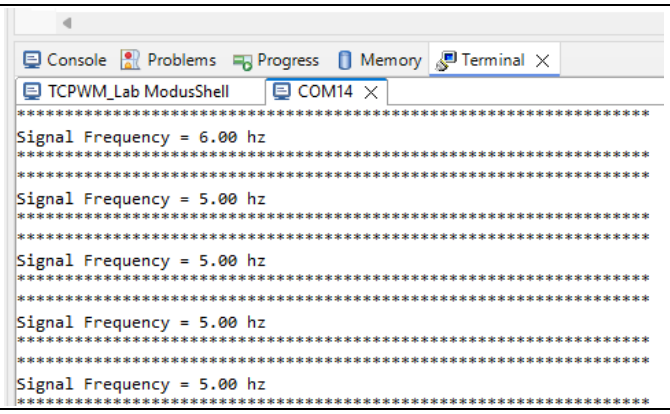



**Figure 83 Terminal output of the signal frequency**

## 5.6 Timer-capture for event measuring

The capture functionality increments/decrements a counter between 0 and PERIOD. When the capture event is activated the counter value COUNTER is copied to CC (and CC is copied to CC_BUFF). The capture event can be triggered through the capture trigger input or through a firmware write to command register.

In the figure below, the capture is set up for a rising edge event. When the capture occurs, the COUNTER is copied into the CC register. It will be copied into the CC_BUFF register on the next event.

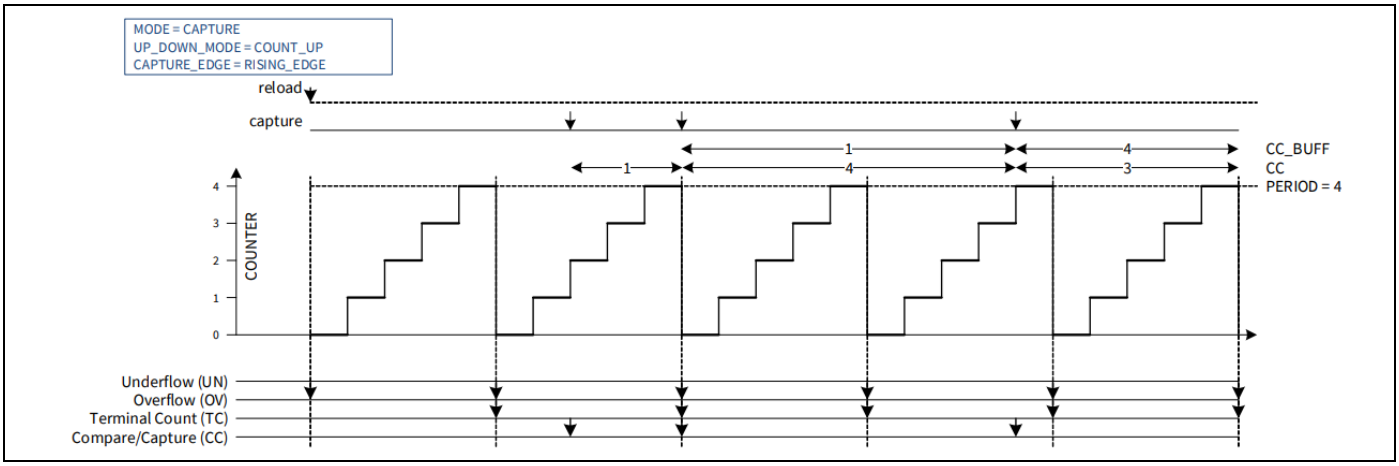



**Figure 84 Terminal output of the signal frequency**

In this portion of the lab, we will add a TCPWM capture to track how long a pulse is. This can be used to determine the duty cycle of a signal.

1. Navigate to the **Peripherals** tab of the device configurator
2. Set the alias for another TCPWM counter to **CAPTURE** and enable it by toggling the checkbox, selecting **Counter** as the personality, then pressing OK.

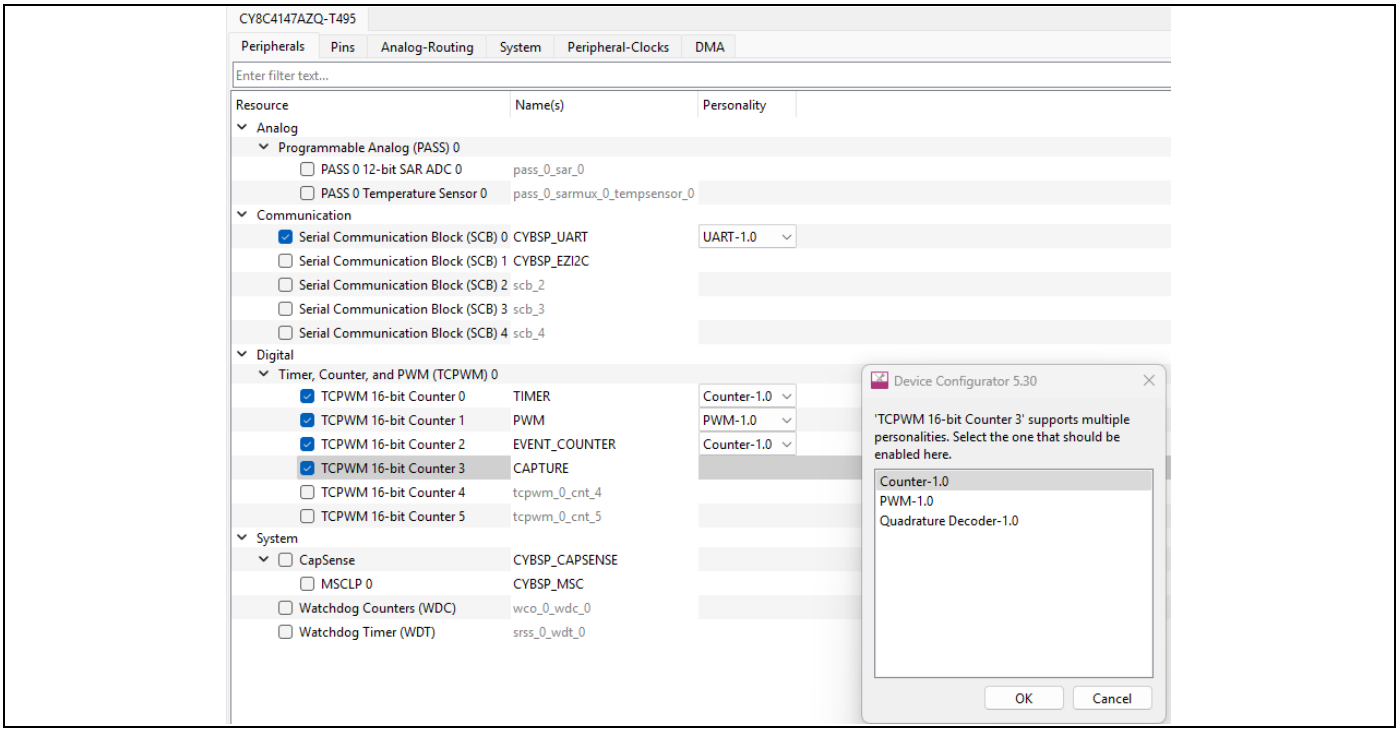



**Figure 85 Enabling the TCPWM capture in the device configurator**

3. Using the same peripheral clock as the previous TCPWM counters and a period of 32000, the TCPWM counter in capture mode can generate an interrupt on the capture of a signal edge

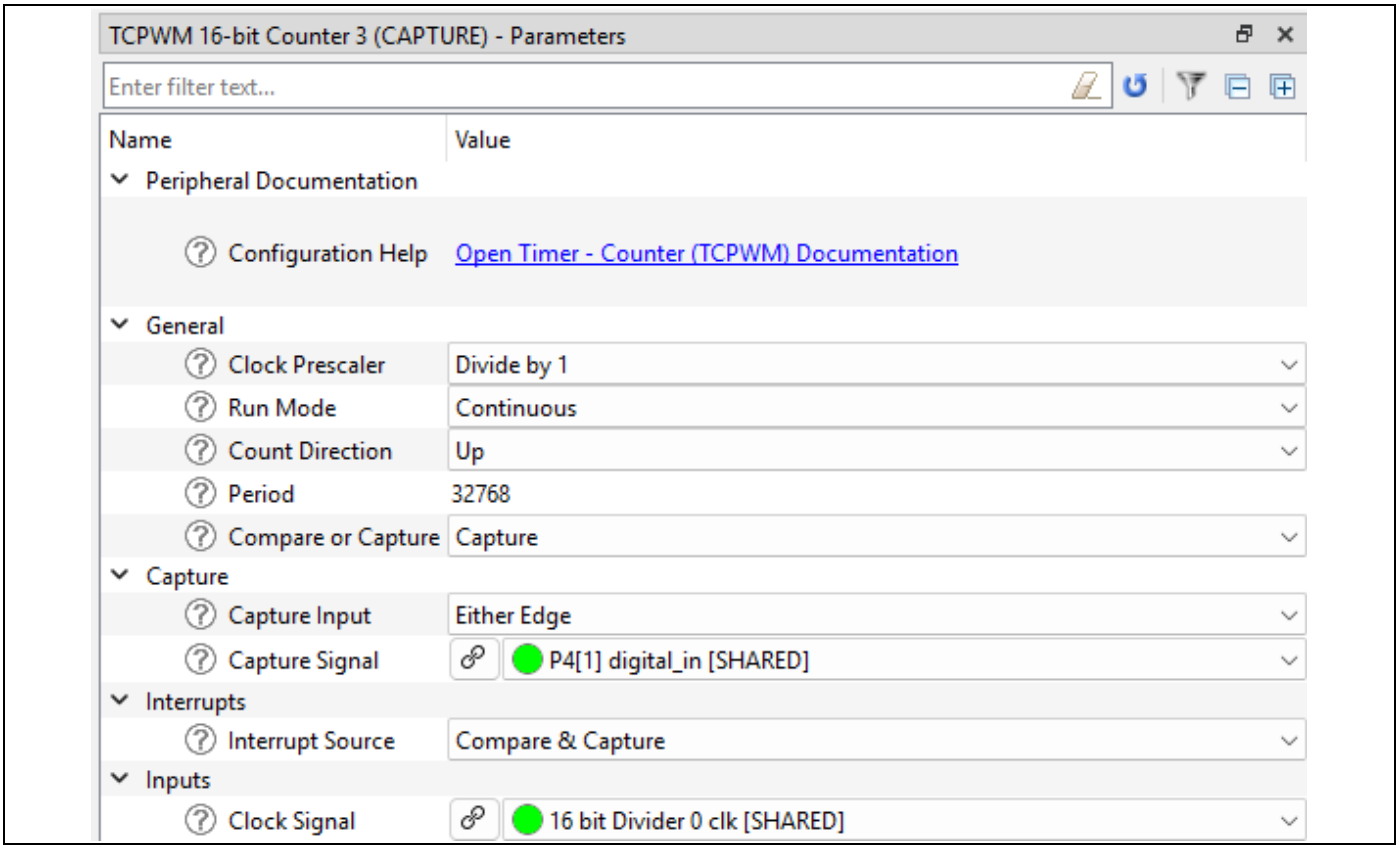



**Figure 86 TCPWM PWM capture configuration in the device configurator**

5. Save the device configurator settings
6. Initialize, enable, and start the TCPWM counter in the main function of the project

```
if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(CAPTURE_HW, CAPTURE_NUM, &CAPTURE_config))
{
        /* Handle possible errors */
}

/* Enable the initialized counters */
Cy_TCPWM_Enable_Multiple(PWM_HW, TIMER_MASK | PWM_MASK | EVENT_COUNTER_MASK | CAPTURE_MASK);
/* Then start the counters */
Cy_TCPWM_TriggerReloadOrIndex(PWM_HW, TIMER_MASK | PWM_MASK | EVENT_COUNTER_MASK | CAPTURE_MASK);
```

**Figure 87 Initialize the TCPWM capture counter in user code for PSOC™ 4**

```
if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(CAPTURE_HW, CAPTURE_NUM, &CAPTURE_config))
{
        /* Handle possible errors */
}

/* Enable the initialized counters */
Cy_TCPWM_Enable_Multiple(PWM_HW, TIMER_MASK | PWM_MASK);
Cy_TCPWM_Enable_Multiple(EVENT_COUNTER_HW, EVENT_COUNTER_MASK | CAPTURE_MASK);

/* Then start the counters */
Cy_TCPWM_TriggerReloadOrIndex(PWM_HW, TIMER_MASK | PWM_MASK );
Cy_TCPWM_TriggerReloadOrIndex(EVENT_COUNTER_HW, EVENT_COUNTER_MASK | CAPTURE_MASK);
```

**Figure 88 Initialize the TCPWM capture counter in user code for PSOC™ 6**

```
if (CY_TCPWM_SUCCESS != Cy_TCPWM_Counter_Init(CAPTURE_HW, CAPTURE_NUM, &CAPTURE_config))
{
        /* Handle possible errors */
}

/* Enable the CAPTURE counter */
Cy_TCPWM_Counter_Enable(CAPTURE_HW, CAPTURE_NUM);
/* Then start the counter */
Cy_TCPWM_TriggerStart_Single(CAPTURE_HW, CAPTURE_NUM);
```

**Figure 89 Initialize the TCPWM capture counter in user code for PSOC™ Edge**

7. Initialize a **SysInt** for the capture TCPWM

```
cy_stc_sysint_t capture_intr_cfg =
{
        /*.intrSrc =*/ CAPTURE_IRQ,
        /*.intrPriority =*/ 3UL
};

Cy_SysInt_Init(&capture_intr_cfg, capture_isr);

/* Enable the interrupt */
NVIC_EnableIRQ(capture_intr_cfg.intrSrc);
```

**Figure 90 Initialize the TCPWM capture counter in user code**

8. Add the ISR

```
volatile bool capture_flag = false;

void capture_isr(void)
{
        capture_flag = true;
        Cy_TCPWM_ClearInterrupt(CAPTURE_HW, CAPTURE_NUM, Cy_TCPWM_GetInterruptStatus(CAPTURE_HW,
                                                                                     CAPTURE_NUM));
}
```

**Figure 91 Capture Interrupt Service Routine**

8. Update the main function

   a. Add two new variables

```
int main(void)
{
        cy_rslt_t result;
        uint32_t capture_result_high = 0;
        uint32_t capture_result_low = 0;
```

**Figure 92 Capture variables**

   b. Service the **capture_flag** in the for loop of the main function

```
if (true == capture_flag)
{
        capture_flag = false;
        if (0UL == Cy_GPIO_Read(FEEDBACK_PIN_PORT, FEEDBACK_PIN_PIN))
        {
                capture_result_low = Cy_TCPWM_Counter_GetCaptureBuf(CAPTURE_HW, CAPTURE_NUM);
        }
        else {
                capture_result_high = Cy_TCPWM_Counter_GetCaptureBuf(CAPTURE_HW, CAPTURE_NUM);
        }
        Cy_TCPWM_Counter_SetCounter(CAPTURE_HW, CAPTURE_NUM, 0);
}
```

**Figure 93 Capture variables**

   c. Calculate and print out the duty cycle of the signal

```
for (;;)
{
        if (true == timer_flag)
        {
                timer_flag = false;
                uint32_t event_count = Cy_TCPWM_Counter_GetCounter(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);
                uint32_t counter_period = Cy_TCPWM_Counter_GetPeriod(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);
                uint32_t clock_divider = Cy_SysClk_PeriphGetAssignedDivider(PCLK_TCPWM_CLOCKS1);
                uint32_t clock_frequency = Cy_SysClk_PeriphGetFrequency((clock_divider>>6 & 0x03), (clock_divider&0x3F));
                float counter_time = (1/(float)clock_frequency)*(float)counter_period;
                float signal_frequency = (float)event_count/counter_time;

                Cy_SCB_UART_PutString(CYBSP_UART_HW, "**********************************************************************\n\r");
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Signal Frequency = ");
                print_float(signal_frequency);
                Cy_SCB_UART_PutString(CYBSP_UART_HW, " hz \r\n");

                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Signal Duty Cycle =  ");
                print_float(((float)capture_result_high * 100)/((float)capture_result_low + (float)capture_result_high-1));
                Cy_SCB_UART_PutString(CYBSP_UART_HW, " % \r\n");

                Cy_SCB_UART_PutString(CYBSP_UART_HW, "**********************************************************************\n\r");
        }
```

**Figure 94 Calculating and printing the captured duty cycle of the signal**

9. Program the device and observe the terminal output

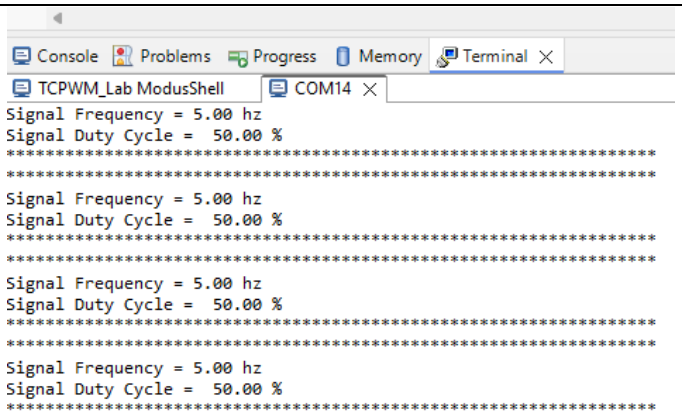



**Figure 95 Terminal output with duty cycle measurement added**

10. Now we can modify the PWM duty cycle in the user code

```
for (;;)
{
        if (true == timer_flag)
        {
                timer_flag = false;
                uint32_t event_count = Cy_TCPWM_Counter_GetCounter(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);
                uint32_t counter_period = Cy_TCPWM_Counter_GetPeriod(EVENT_COUNTER_HW, EVENT_COUNTER_NUM);
                uint32_t clock_divider = Cy_SysClk_PeriphGetAssignedDivider(PCLK_TCPWM_CLOCKS1);
                uint32_t clock_frequency = Cy_SysClk_PeriphGetFrequency((clock_divider>>6 & 0x03), (clock_divider&0x3F));
                float counter_time = (1/(float)clock_frequency)*(float)counter_period;
                float signal_frequency = (float)event_count/counter_time;

                Cy_SCB_UART_PutString(CYBSP_UART_HW, "**********************************************************************\n\r");
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Signal Frequency = ");
                print_float(signal_frequency);
                Cy_SCB_UART_PutString(CYBSP_UART_HW, " hz \r\n");

                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Signal Duty Cycle =  ");
                print_float(((float)capture_result_high * 100)/((float)capture_result_low + (float)capture_result_high-1));
                Cy_SCB_UART_PutString(CYBSP_UART_HW, " % \r\n");

                Cy_SCB_UART_PutString(CYBSP_UART_HW, "**********************************************************************\n\r");

                uint32_t pwm_period = Cy_TCPWM_PWM_GetPeriod0(PWM_HW, PWM_NUM);
                uint32_t pwm_compare = Cy_TCPWM_PWM_GetCompare0(PWM_HW, PWM_NUM) + 100;

                if (pwm_period > pwm_compare)
                {
                        Cy_TCPWM_PWM_SetCompare0(PWM_HW, PWM_NUM, pwm_compare+100);
                }
                else
                {
                        Cy_TCPWM_PWM_SetCompare0(PWM_HW, PWM_NUM, 100);
                }

        }
```

**Figure 96 Terminal output with duty cycle measurement added**

11. Program the device and observe the terminal output with the variable PWM duty cycle

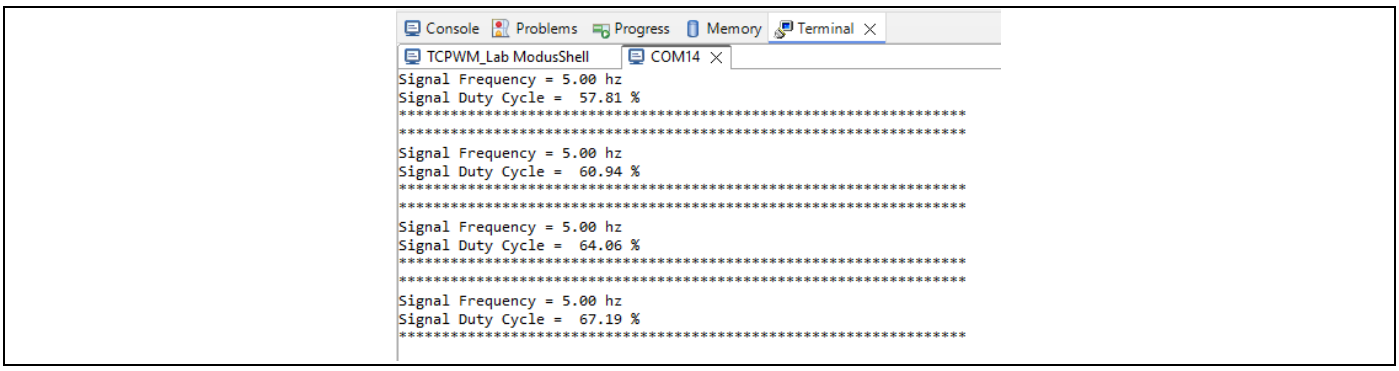



**Figure 97 Terminal output with a variable duty cycle**

## 5.7 Conclusion

This lab demonstrated how to enable and configure the TCPWM counter for different use cases using the ModusToolbox™ device configurator and the accompanying PDL library documentation. You should now be able to use the TCPWM counter peripheral to create simple timers, generate PWM signals, count and measure events.

# 6 Use the EZI2C driver to communicate with an I2C host

## 6.1 Objective

The objective of this lab is to communicate to an I2C host using the EZI2C driver for the PSOC™ serial communication block (SCB).

## 6.2 Description

The serial communication block can be configured as an I2C host or an I2C slave. Using the EZI2C slave peripheral driver, an application can be developed quickly. The trainee will learn how to enable the Systick timer to tick every 1 ms, configure a SCB for EZI2C slave mode, update the read buffer, and act upon data written.

- Enabling EZI2C slave peripheral
- Updating the EZI2C buffer with Systick up time
- Controlling an LED using an I2C write command

The standard I2C bus is a two-wire interface with a serial data (SDA) line and a serial clock (SCL) line. I2C devices are connected to these lines using open collector or open-drain output stages, with pull-up resistors (Rp). A simple master/slave relationship exists between devices. Masters and slaves can operate as either transmitter or receiver. Each slave device connected to the bus is software addressable by a unique 7-bit address.

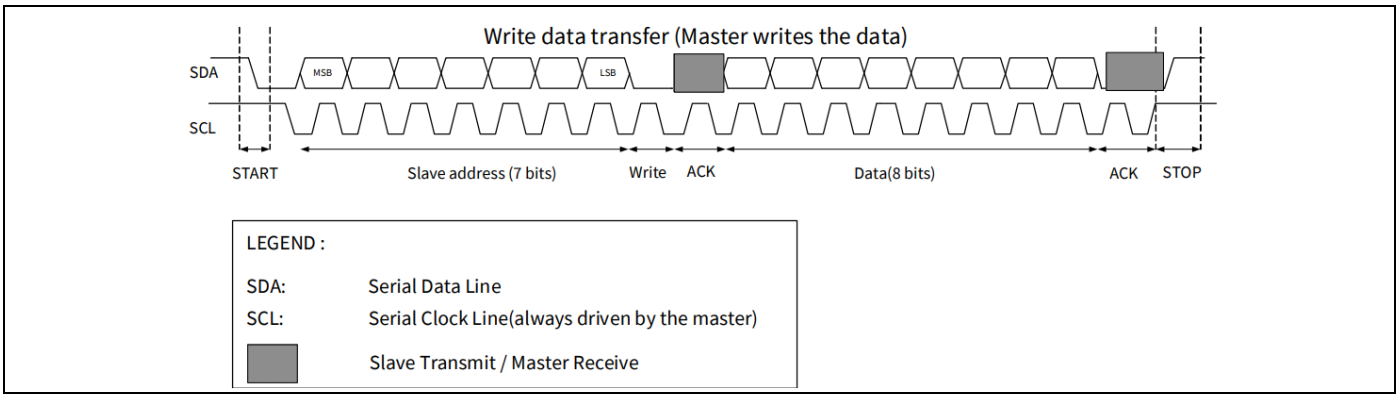



**Figure 98 Master write data transfer**

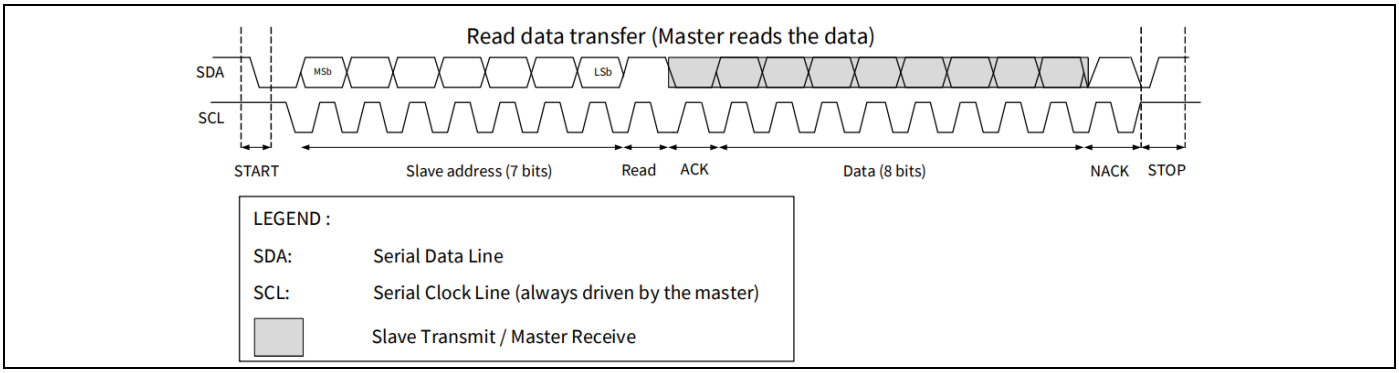



**Figure 99 Master read data transfer**

## Use the EZI2C driver to communicate with an I2C host

The EZI2C driver provided by Infineon adds a protocol on top of I2C 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 I2C.

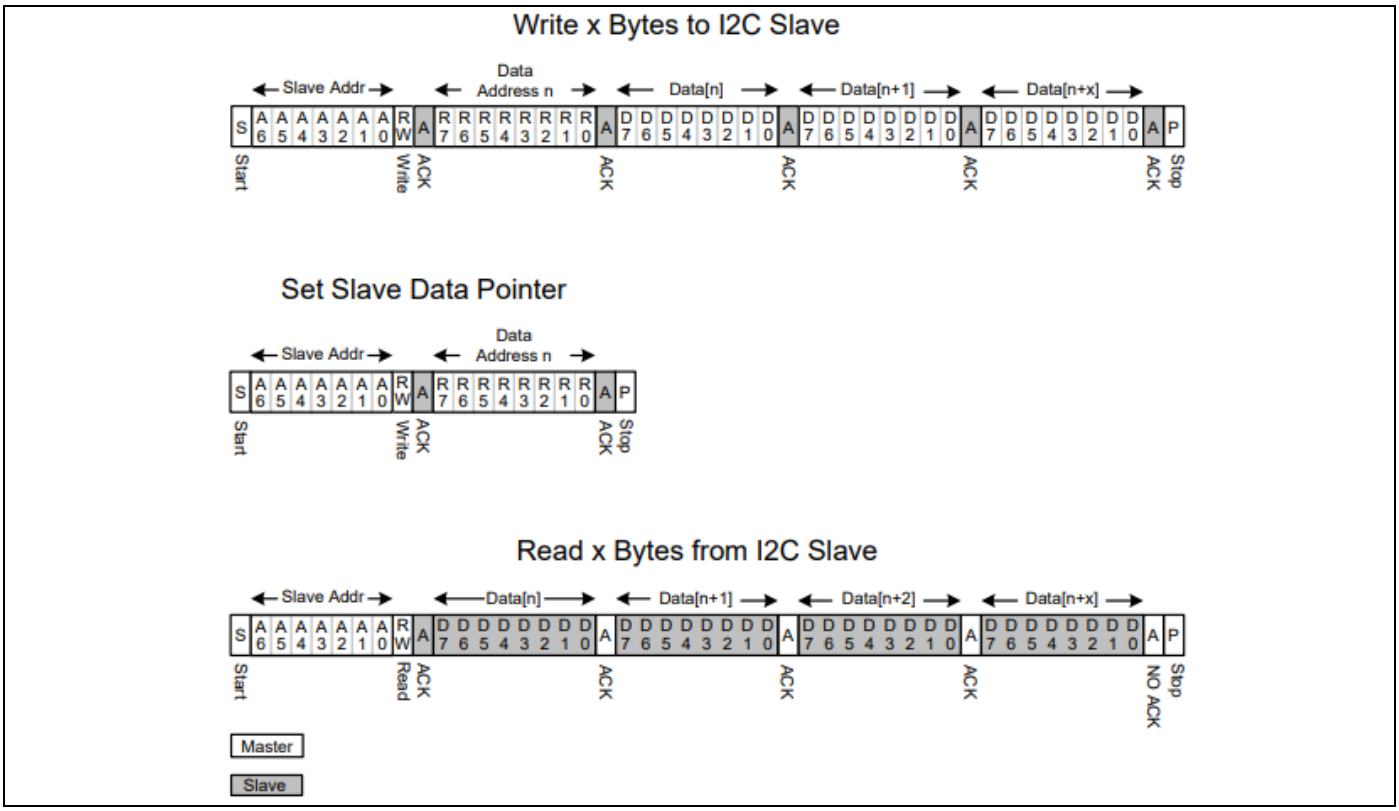



**Figure 100 EZI2C example with 8-bit sub-addressing**

## 6.3 Enabling EZI2C slave peripheral

1. The project created in the first exercise can be used, or the instructions shown there can be used to create a new project.
2. Once the project has been opened, navigate to the Eclipse IDE Quick Panel and open the Device Configurator. Navigate to the **Peripherals** tab and enable a SCB for **EZI2C** that is connected to the KitProg on the evaluation kit you are using

**Table 2 SCB associated with the KitProg**

| Evaluation Kit | SCB | SDA GPIO | SCL GPIO |
|---|---|---|---|
| CY8CPROTO-041TP | SCB1 | P6.4 | P6.3 |
| CY8CKIT-040T | SCB1 | P2.3 | P2.2 |
| CY8CKIT-041S-MAX | SCB0 | P1.1 | P1.0 |
| CY8CKIT-062S2-43012 | SCB3 | P6.1 | P6.0 |
| KIT_PSE84_EVAL | SCB0 | P8.1 | P8.0 |

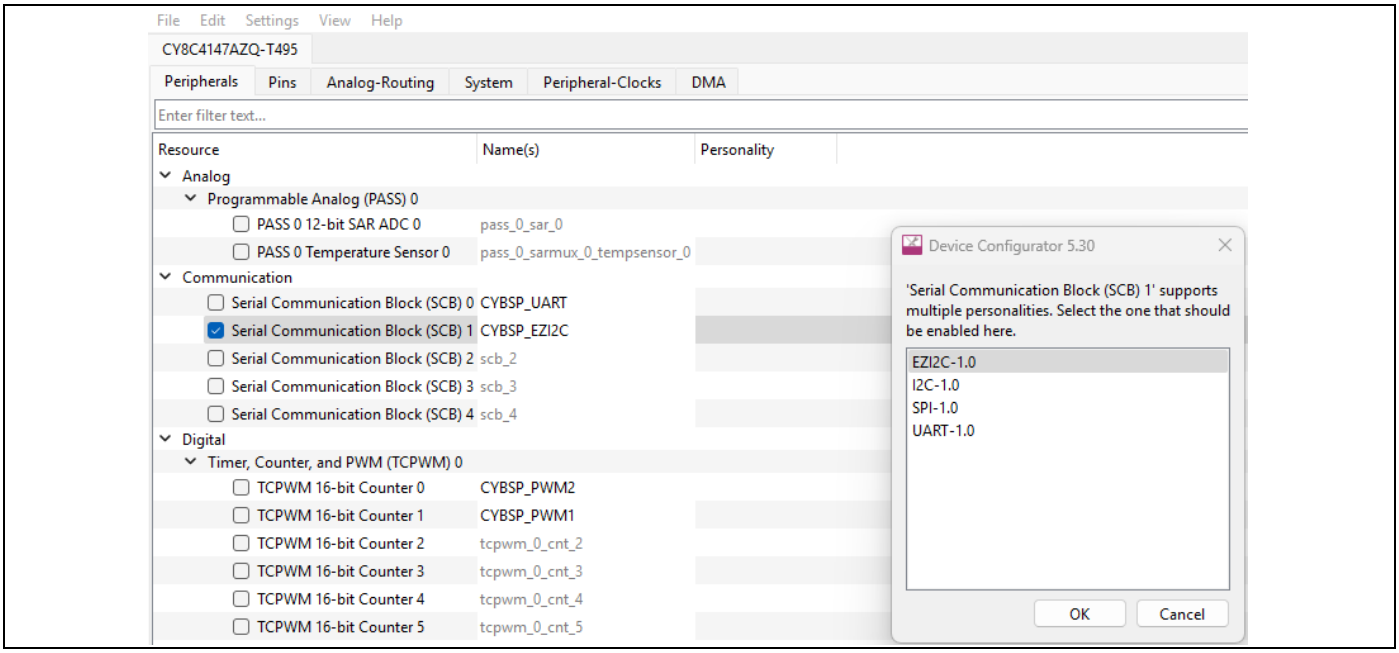



**Figure 101 Enabling a PSOC™ SCB for EZI2C mode**

## Use the EZI2C driver to communicate with an I2C host

3. Configure the clock signal to the peripheral, the SCL, and SDA pins

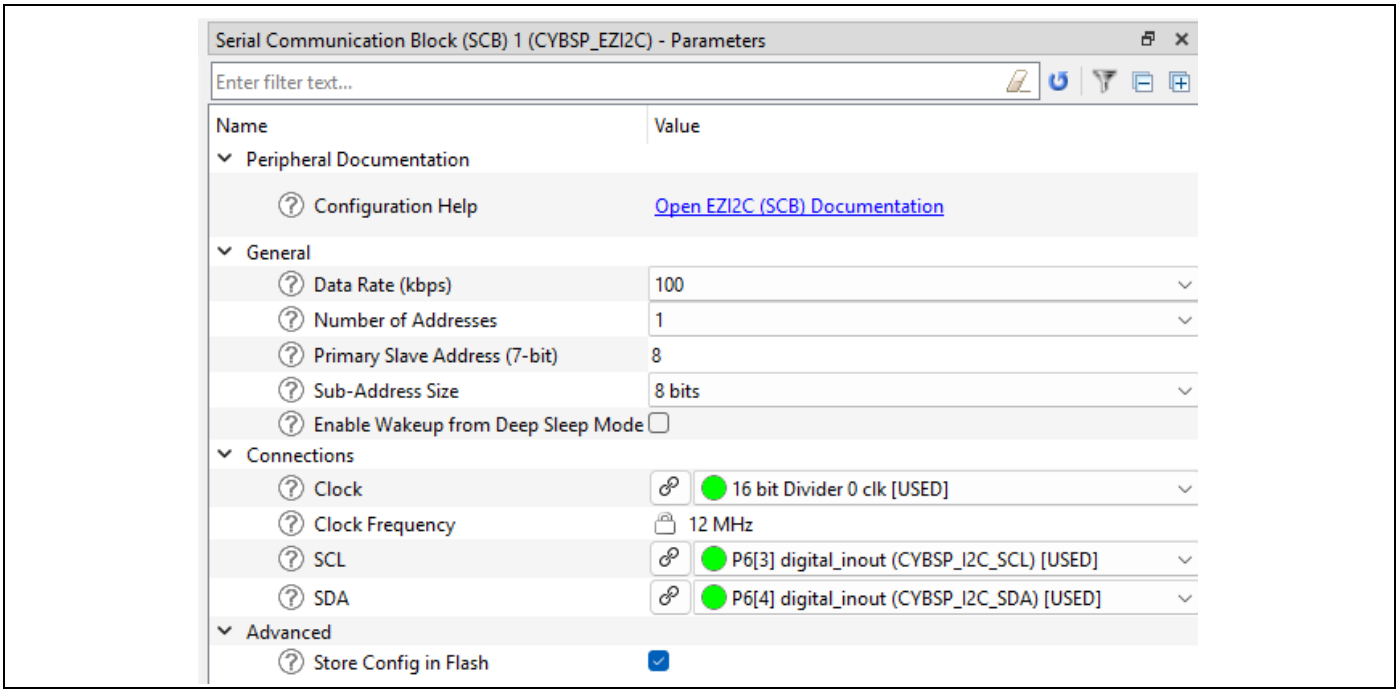



**Figure 102 Configuring the EZI2C parameters**

4. Save the device configurator
5. Open the **EZI2C (SCB) Documentation** and locate the documentation for APIs relating to initializing the interface, setting the buffer, initializing the interrupt, and enabling the interface

## Use the EZI2C driver to communicate with an I2C host

6. Let's add those APIs to the main function
    a. The device configurator generates the EZI2C configuration structure used with the **Cy_SCB_EZI2C_Init** API, so only the API needs to be copied into the main function after the cybsp_init function call and after the enable_irq function call
    b. Next, we will set the buffer for the peripheral to use for reads and writes initiated by the I2C master. We want the entire buffer to be readable and writeable, so the **Cy_SCB_EZI2C_SetBuffer1** input **rwBoundary** will be the same as the buffer size
    c. The **SysInt** interrupt must be configured with the accompanying ISR
    d. Lastly, the SCB must be enabled with the **Cy_SCB_EZI2C_Enable** API

```
#define BUFFER_SIZE (8UL)
uint8_t buffer[BUFFER_SIZE];
cy_stc_scb_ezi2c_context_t ezI2cContext;


void EZI2C_Isr(void)
{
	Cy_SCB_EZI2C_Interrupt(CYBSP_EZI2C_HW, &ezI2cContext);
}

int main(void)
{
	cy_rslt_t result;

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

	/* Configure EZI2C slave */
	(void) Cy_SCB_EZI2C_Init(CYBSP_EZI2C_HW, &CYBSP_EZI2C_config, &ezI2cContext);

	Cy_SCB_EZI2C_SetBuffer1(CYBSP_EZI2C_HW, buffer, BUFFER_SIZE, BUFFER_SIZE, &ezI2cContext);

	/* Populate configuration structure */
	const cy_stc_sysint_t ezI2cIntrConfig =
	{
		.intrSrc      = CYBSP_EZI2C_IRQ,
		.intrPriority = 3U,
	};

	/* Hook interrupt service routine and enable interrupt */
	(void) Cy_SysInt_Init(&ezI2cIntrConfig, &EZI2C_Isr);
	NVIC_EnableIRQ(ezI2cIntrConfig.intrSrc);

	/* Enable EZI2C to operate */
	Cy_SCB_EZI2C_Enable(CYBSP_EZI2C_HW);

	/* Enable global interrupts */
	__enable_irq();

	for (;;)
	{
	}
}
```

**Figure 103 Main function with EZI2C enabled**

7. Program the device

## Use the EZI2C driver to communicate with an I2C host

8. Using the Windows search bar, look for **Bridge Control Panel**. This tool comes with ModusToolbox™ and allows your PC to utilize the I2C, UART, or SPI bridge on the KitProg that is available on the evaluation kit

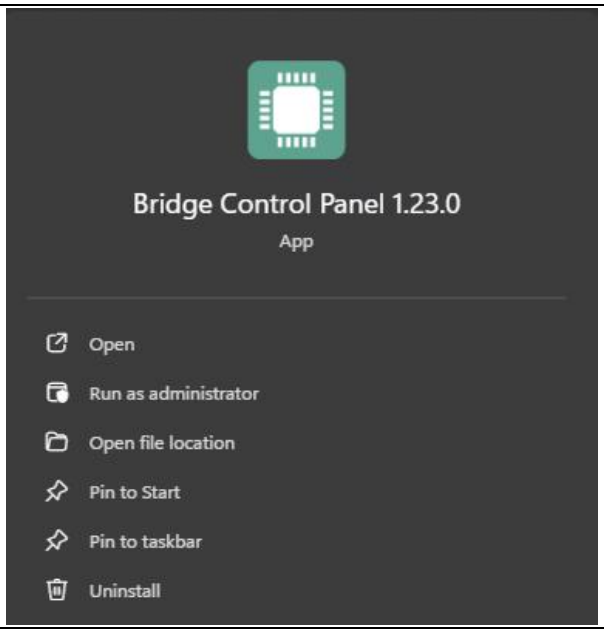



**Figure 104 Bridge Control Panel on Windows**

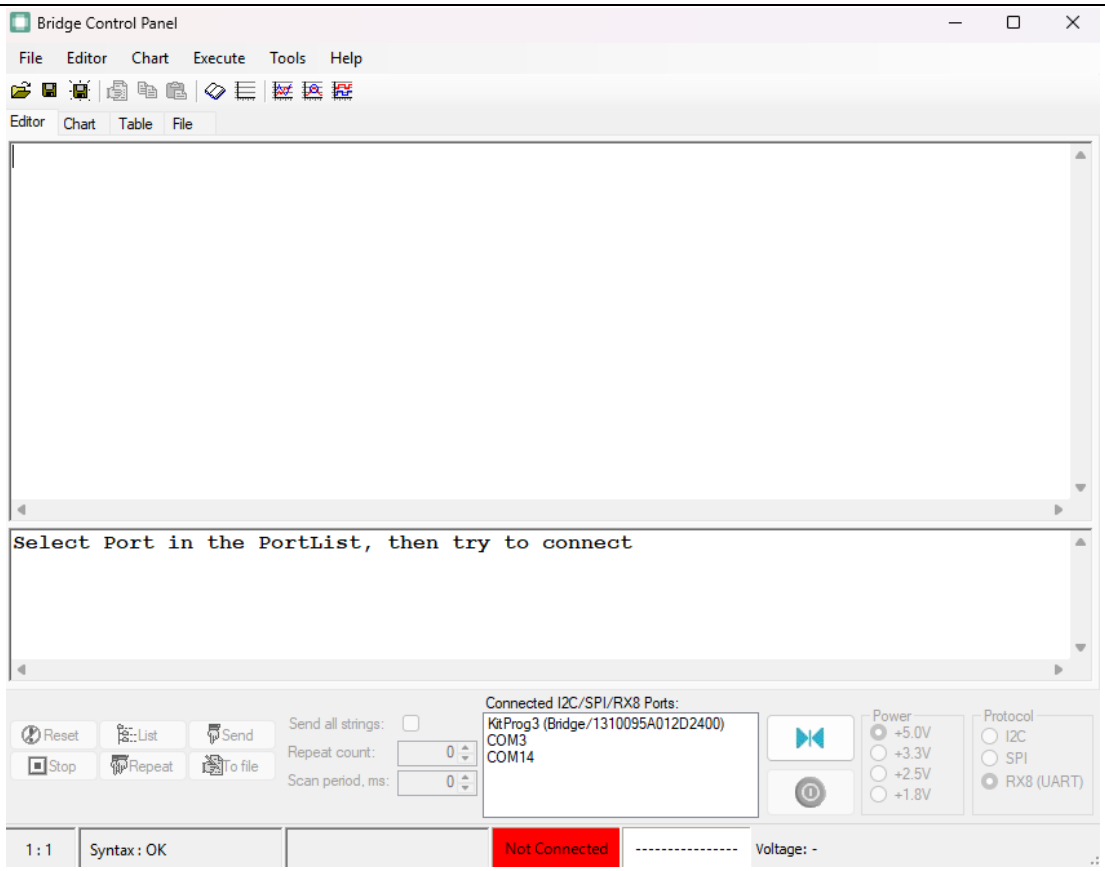



**Figure 105 Bridge Control Panel**

More details on how to use the Bridge Control Panel can be found in the **Help** tab.

## Use the EZI2C driver to communicate with an I2C host

9. To read from the device, start with the “r” character, then the slave address “08”, then each “x” corresponds to a byte read, then end the transaction with a stop using the “p” character

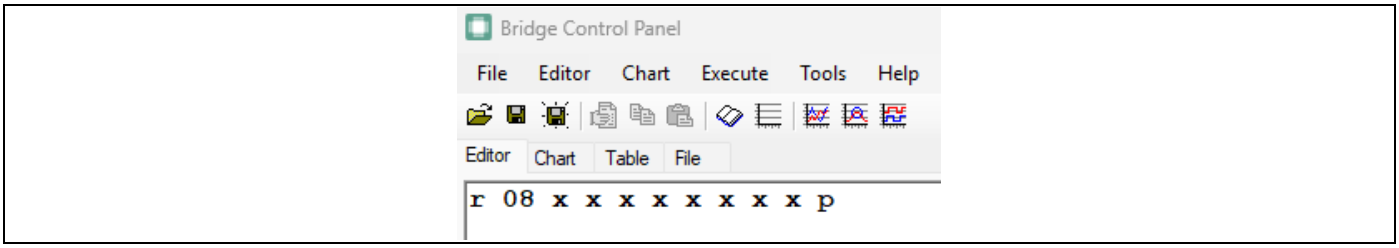



**Figure 106 Bridge Control Panel read 8 bytes command**

10. Connect to the device using the connect button in the bottom right of the application

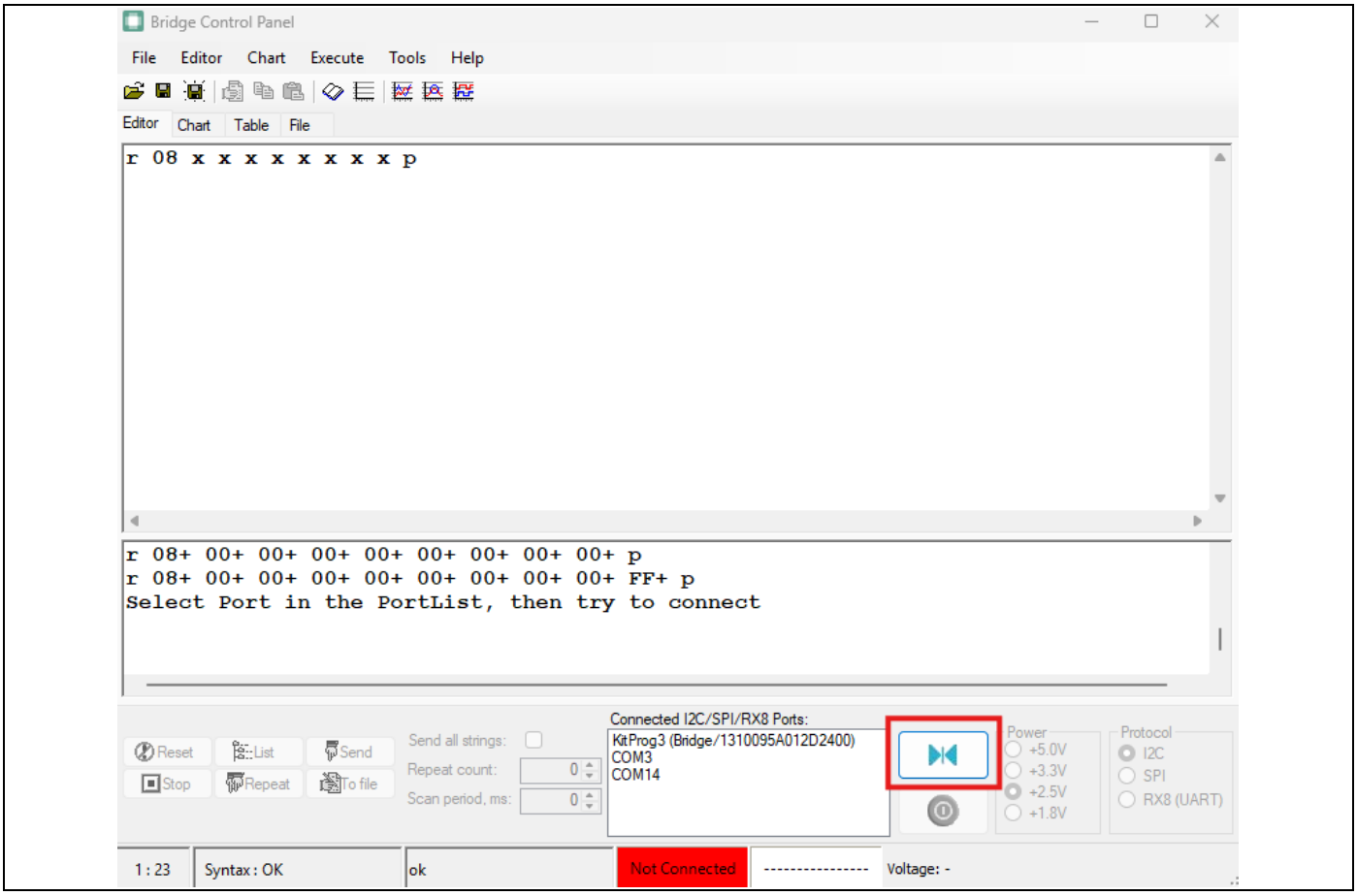



**Figure 107 Bridge Control Panel connect to KitProg**

## Use the EZI2C driver to communicate with an I2C host

11. Once it has connected, press the **Send** button and observer the successful transaction

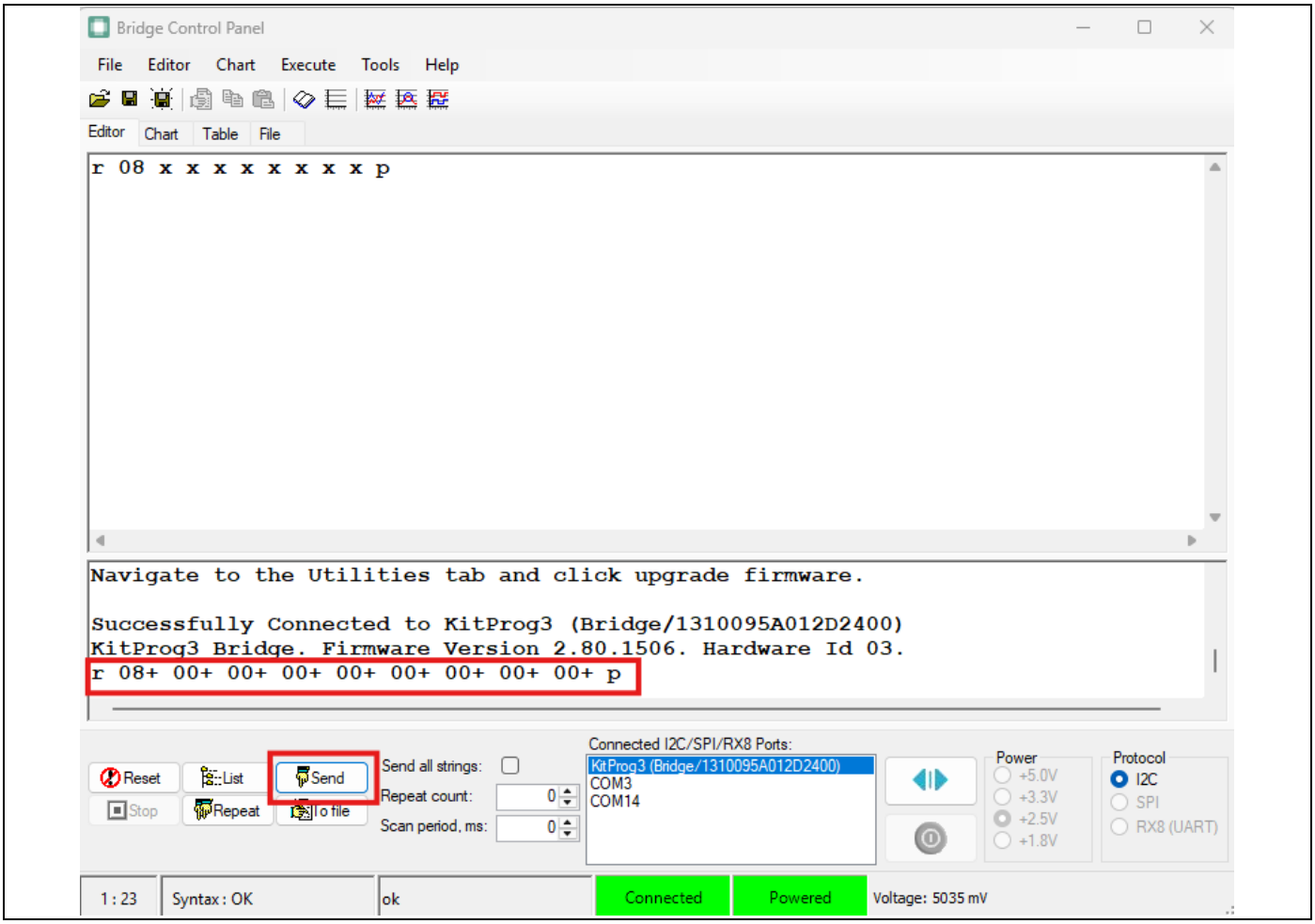



**Figure 108 Bridge Control Panel send command to KitProg**

12. Ensure that you disconnect from the KitProg before trying to program the device

## 6.4 Updating the EZI2C buffer with Systick up time

1. Navigate to the Peripheral Driver Library: Main Page from the Eclipse IDE Quick Panel, then open the PDL API Reference, and navigate to the **Systick (ARM System Timer)** page
2. This reference code can be copied into the main function and modified to use the IMO. The IMO clock frequency can be found by navigating to the device configurators **System** tab
3. Alternatively, the device configurator can be used to generate the initialization code from the **System** tab. Enable the SYSTICK, select the clock source and reload value. The reload value should be the source clock frequency divided by 1000 to achieve a tick interrupt every 1 ms
4. If the LFCLK is not enabled, enable the LFCLK and the source for the LFCLK.

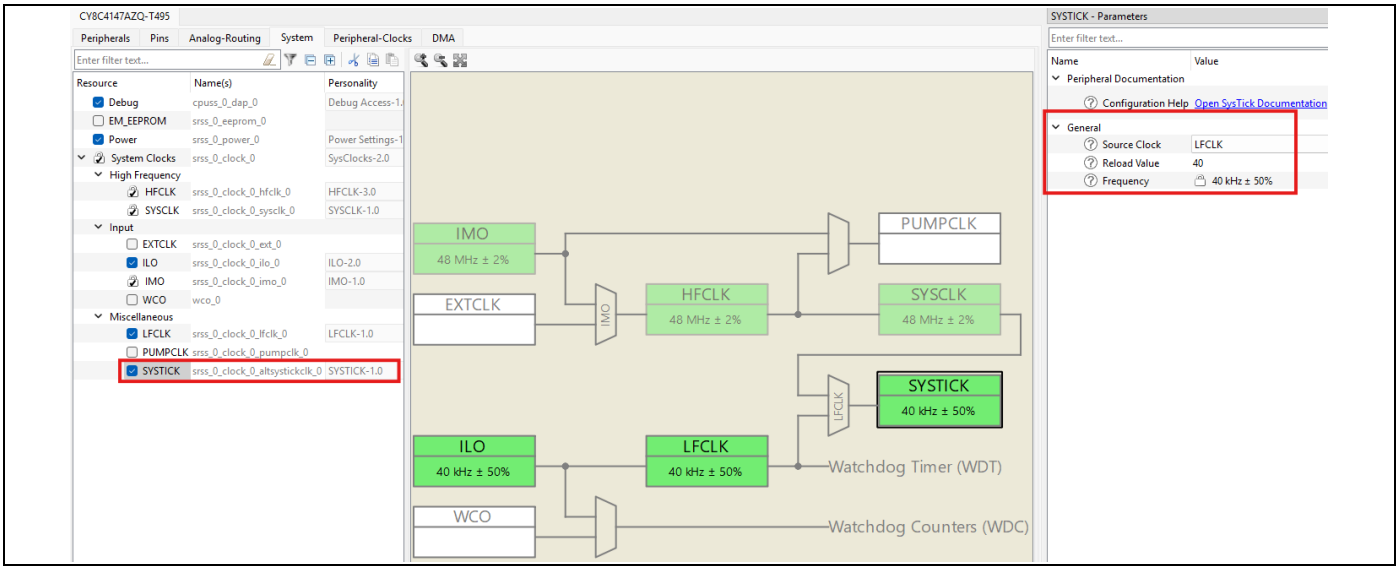



**Figure 109 Systick configuration in the device configurator**

## Use the EZI2C driver to communicate with an I2C host

5. The device configurator will generate the initialization code, but not the ISR code. That must be added to the user code in the main function

```
volatile uint32_t tick = 0;

void EZI2C_Isr(void)
{
        Cy_SCB_EZI2C_Interrupt(CYBSP_EZI2C_HW, &ezI2cContext);
}

void SysTick_Callback(void)
{
        tick++;
}

int main(void)
{
        cy_rslt_t result;

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

        /* Register one of the SysTick callbacks */
        Cy_SysTick_SetCallback(0UL, &SysTick_Callback);
```

**Figure 110 Systick ISR in user application code**

If using the PSOC™ Edge for this lab, please add in the API to initialize and enable the Systick timer for the CM33_NS.

```
Cy_SysTick_Init(CY_SYSTICK_CLOCK_SOURCE_CLK_LF, ((uint32_t)(((1000)/1000000.0)*32768)));
```

**Figure 111 Systick initialization for the PSOC™ Edge CM33 NS core**

6. Add the tick value to the buffer

```
uint32_t old_tick = tick;

for (;;)
{
        if (tick != old_tick)
        {
                old_tick = tick;
                buffer[0] = ((tick>>24) & 0xFF);
                buffer[1] = ((tick>>16) & 0xFF);
                buffer[2] = ((tick>>8) & 0xFF);
                buffer[3] = (tick & 0xFF);
        }
}
```

**Figure 112 Updating EZI2C buffer with tick value**

7. Program the device
8. Using the bridge control panel the buffer can be read periodically and mapped to a variable

## Use the EZI2C driver to communicate with an I2C host

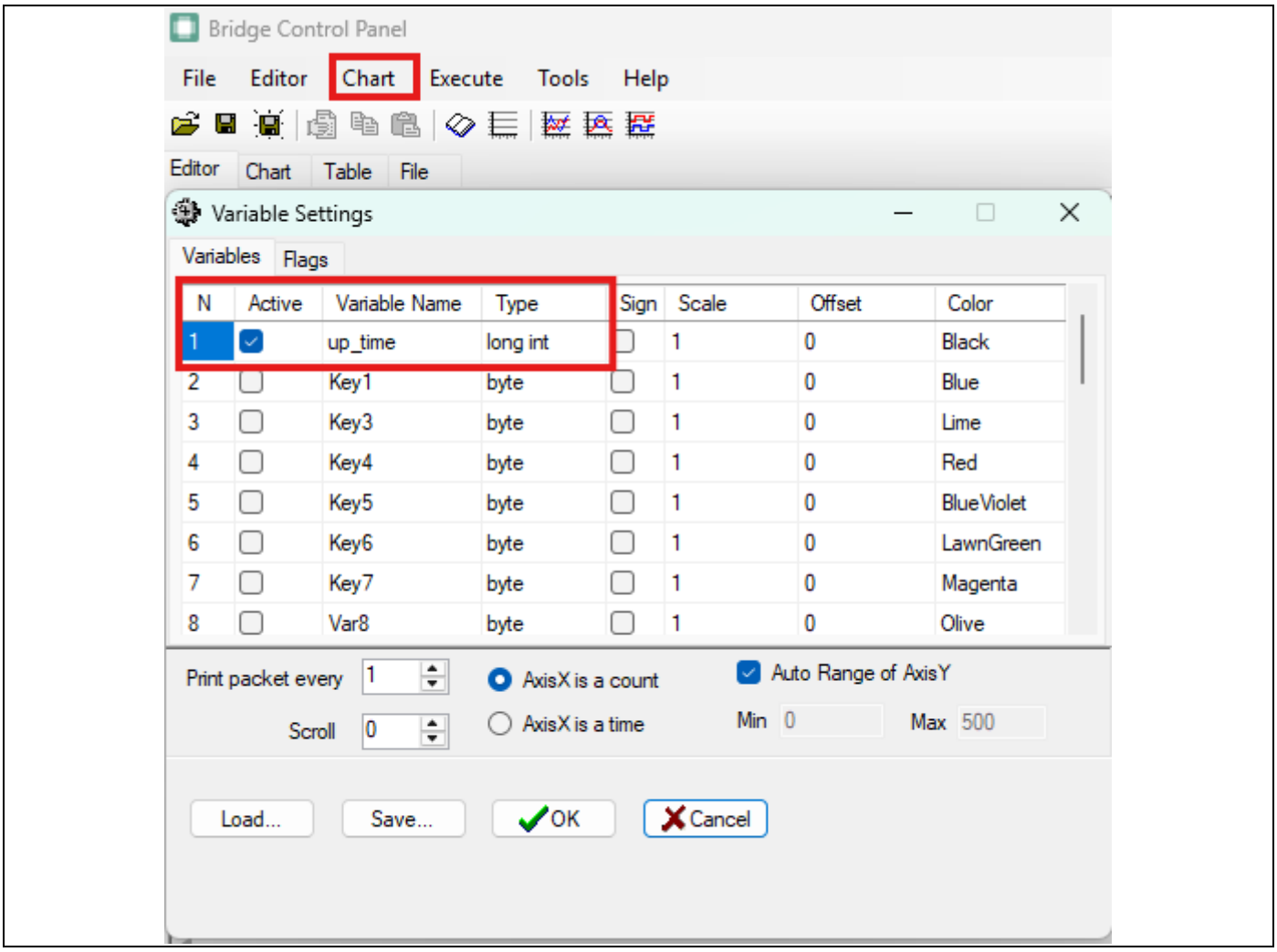



**Figure 113 Adding a chart variable**

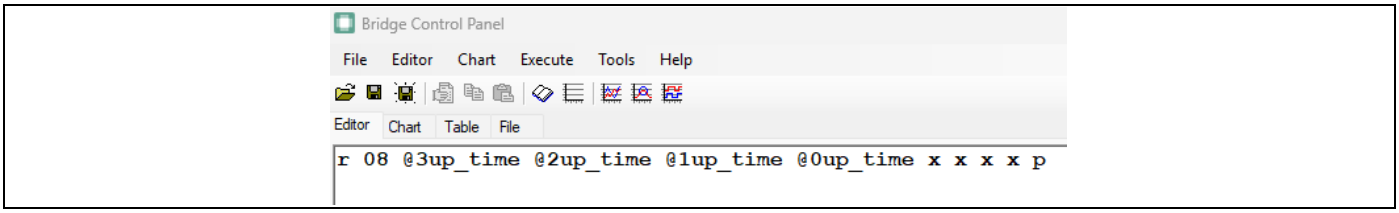



**Figure 114 Configuring the read to map the bytes to the chart variable**

## Use the EZI2C driver to communicate with an I2C host

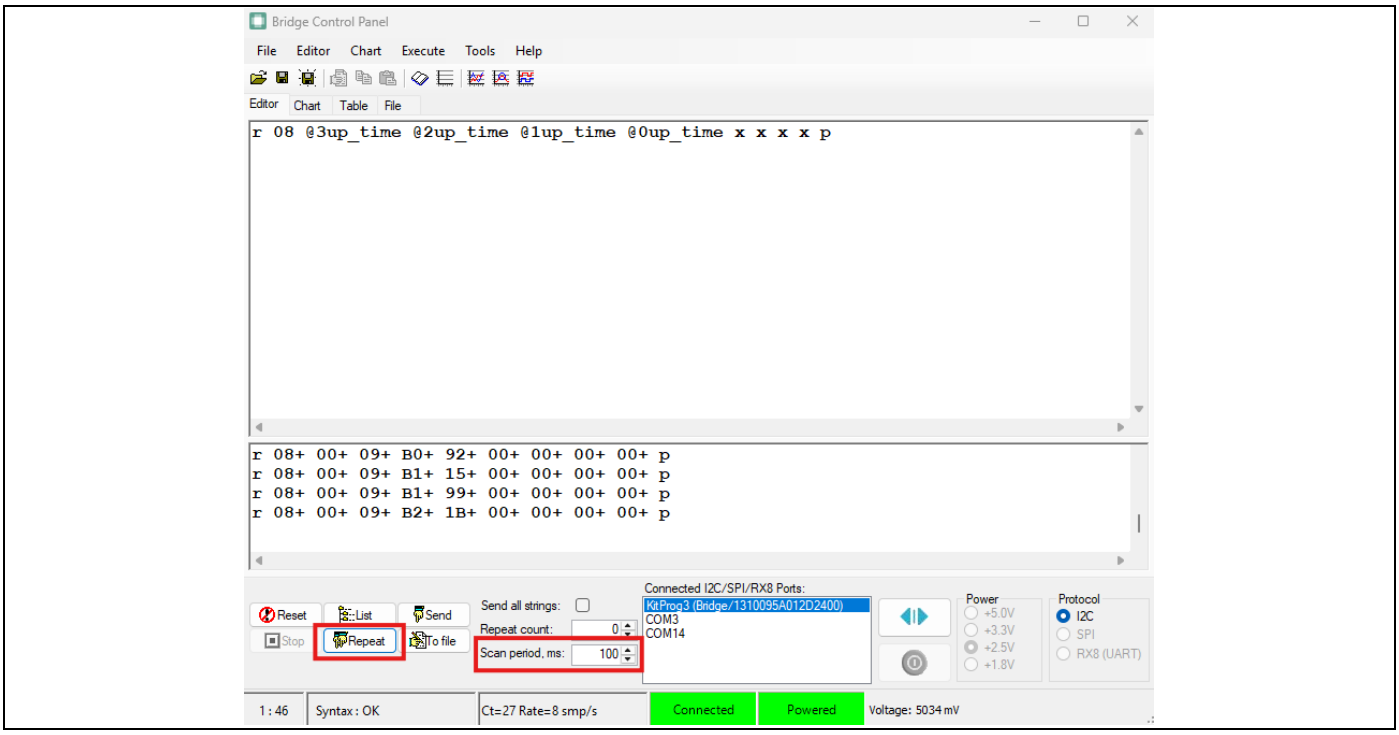



**Figure 115 Repeating the command every 100 ms**

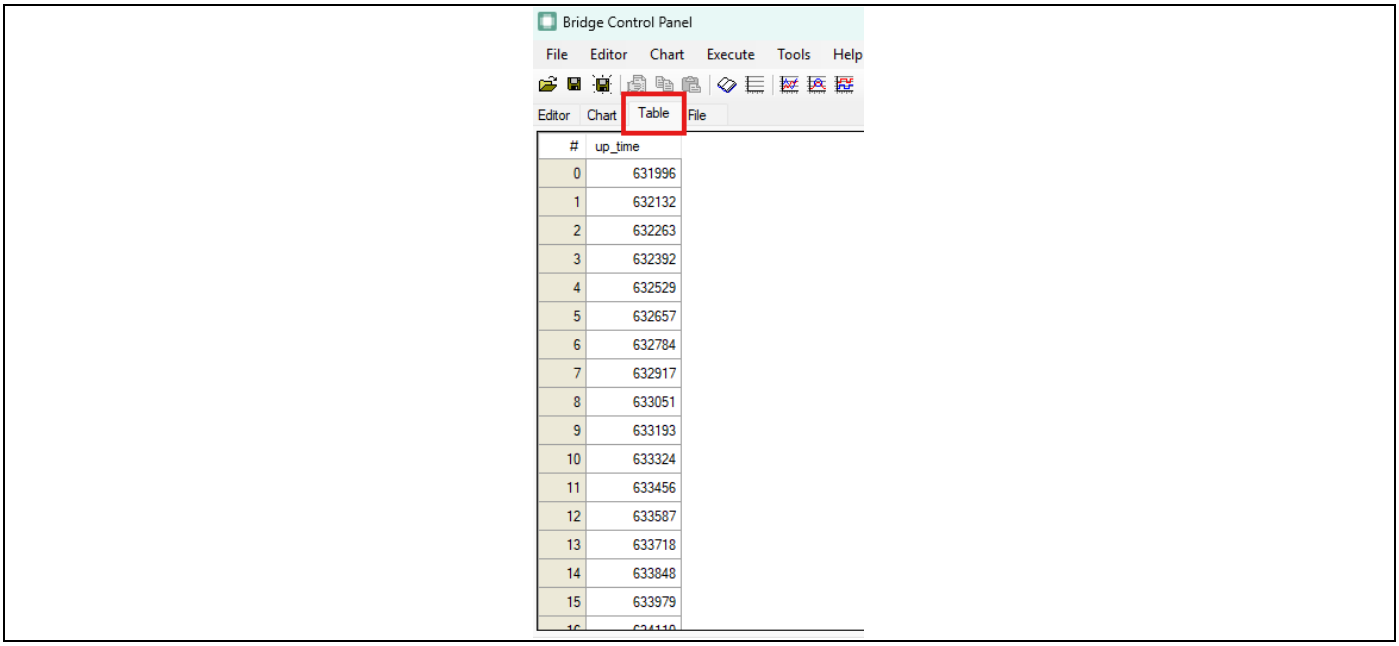



**Figure 116 Table of the variable being read every 100 ms**

## 6.5 Controlling an LED using an I2C write command

The EZI2C driver emulates a typical EEPROM interface with configurable sub-address size (8 or 16 bits). This means that when writing to the device, the steps are I2C start, write address, write sub-address, write data, I2C stop.

1. Enable a User LED as shown in the GPIO lab here
2. We will use the **Cy_SCB_EZI2C_GetActivity** API to determine if a write transaction has occurred on the interface. This API will need to be polled in the user application code. If a write transaction has occurred, check if the 4th byte in the buffer corresponds to a predetermined **TURN_LED_OFF** or **TURN_LED_ON** value, then set the GPIO accordingly

```
#define TURN_LED_OFF    (0x5A)
#define TURN_LED_ON     (0xA5)
```

**Figure 117 Define the LED commands**

```
for (;;)
{
        if (tick != old_tick)
        {
                old_tick = tick;
                buffer[0] = ((tick>>24) & 0xFF);
                buffer[1] = ((tick>>16) & 0xFF);
                buffer[2] = ((tick>>8) & 0xFF);
                buffer[3] = (tick & 0xFF);
        }
        if (CY_SCB_EZI2C_STATUS_WRITE1 ==
                (CY_SCB_EZI2C_STATUS_WRITE1 & Cy_SCB_EZI2C_GetActivity(CYBSP_EZI2C_HW, &ezi2cContext)))
        {
                /* If a write to the first buffer occurred */
                if (TURN_LED_OFF == buffer[4])
                {
                        Cy_GPIO_Clr(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
                }
                else if (TURN_LED_ON == buffer[4])
                {
                        Cy_GPIO_Set(CYBSP_LED2_PORT, CYBSP_LED2_PIN);
                }
        }

}
```

**Figure 118 Controlling an LED based on the 4th byte in the I2C buffer**

3. Program the device, then open the bridge control panel and reconnect to the KitProg

## Use the EZI2C driver to communicate with an I2C host

4. Send 0xA5 to the 4th byte to turn the LED on

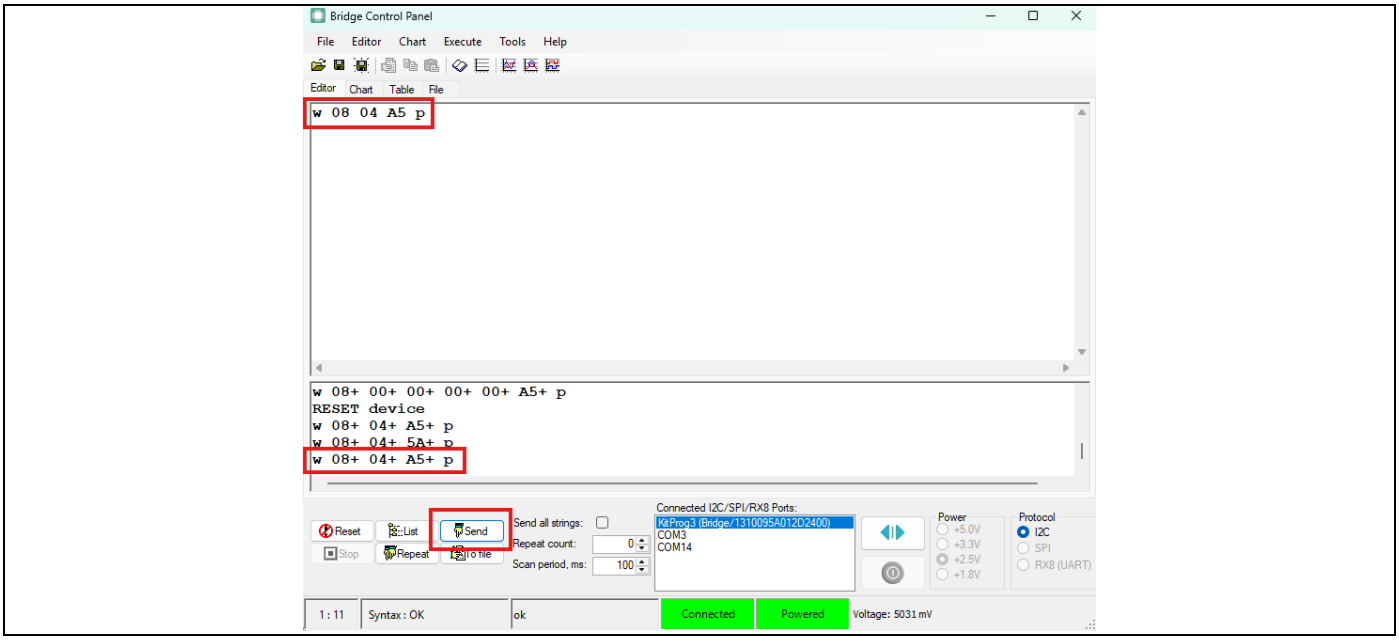



**Figure 119 Writing the command to turn the LED on**

Note that the command structure is a “w” for write, then the address “08”, then the sub-address “04”, the LED command “A5”, and the stop bit “p”.

5. Send 0x5A to the 4th byte to turn the LED off

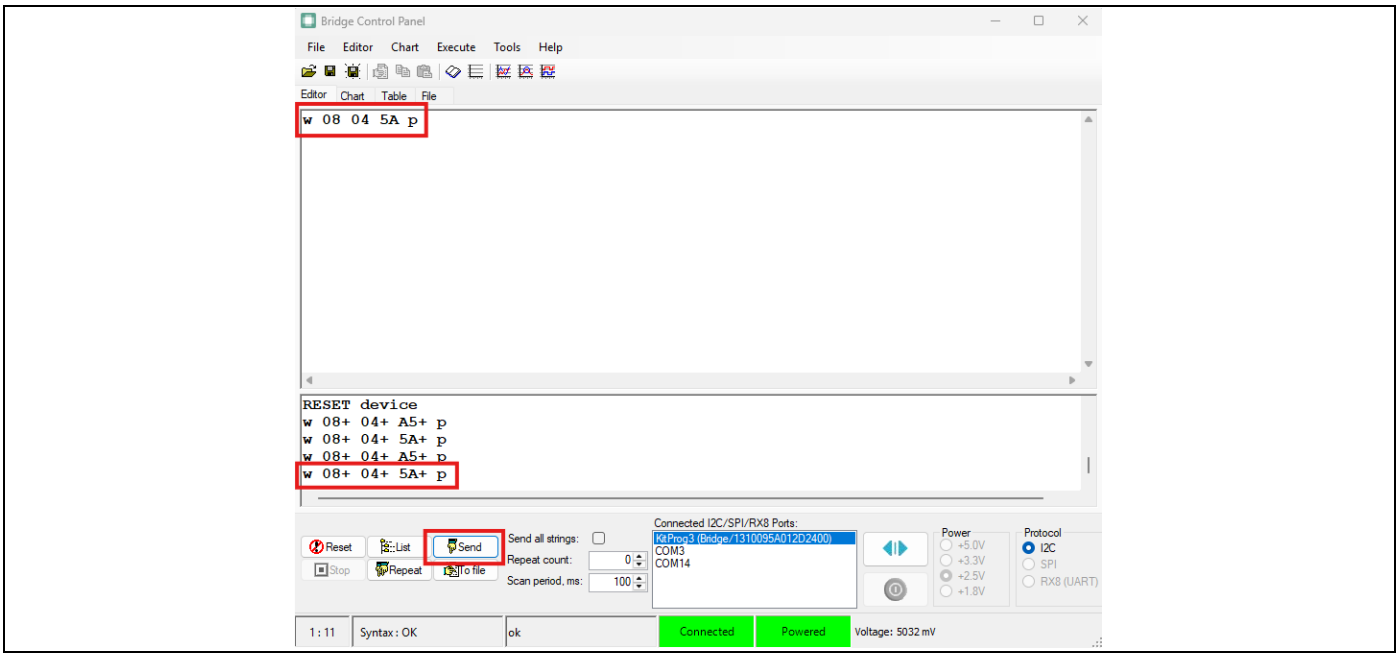



**Figure 120 Writing the command to turn the LED off**

## 6.6 Conclusion

This lab demonstrates how to use the EZI2C driver to quickly setup an I2C slave on a PSOC™ SCB. You should now be able to configure and enable the SCB, use the Systick Timer, read and write data to the device using the Bridge Control Panel.

# 7 Serial Communication Block with SPI

## 7.1 Objective

The objective of this lab is to demonstrate how to use ModusToolbox™ and the PSOC™ peripheral driver library to communicate from one MCU to another using a serial communication block configured for SPI.

## 7.2 Description

The serial communication block (SCB) can be configured as a SPI master or slave. Using the high-level SPI driver, the communication portion of an application can be developed quickly, allowing the user to focus on the rest of the application. The trainee will learn how to setup a PSOC™ device to operate as a SPI master and another PSOC™ device as a SPI slave.

The lab steps utilized a CY8CKIT-041S-MAX as the SPI master and a CY8CKIT-062S2-43012 as the SPI slave.

- SPI master
- SPI slave
- SPI slave with callbacks

The SPI protocol is a synchronous serial interface protocol. Devices operate in either master or slave mode. The master initiates the data transfer. The SCB supports single-master-multiple-slaves topology for SPI. Multiple slaves are supported with individual slave select lines.

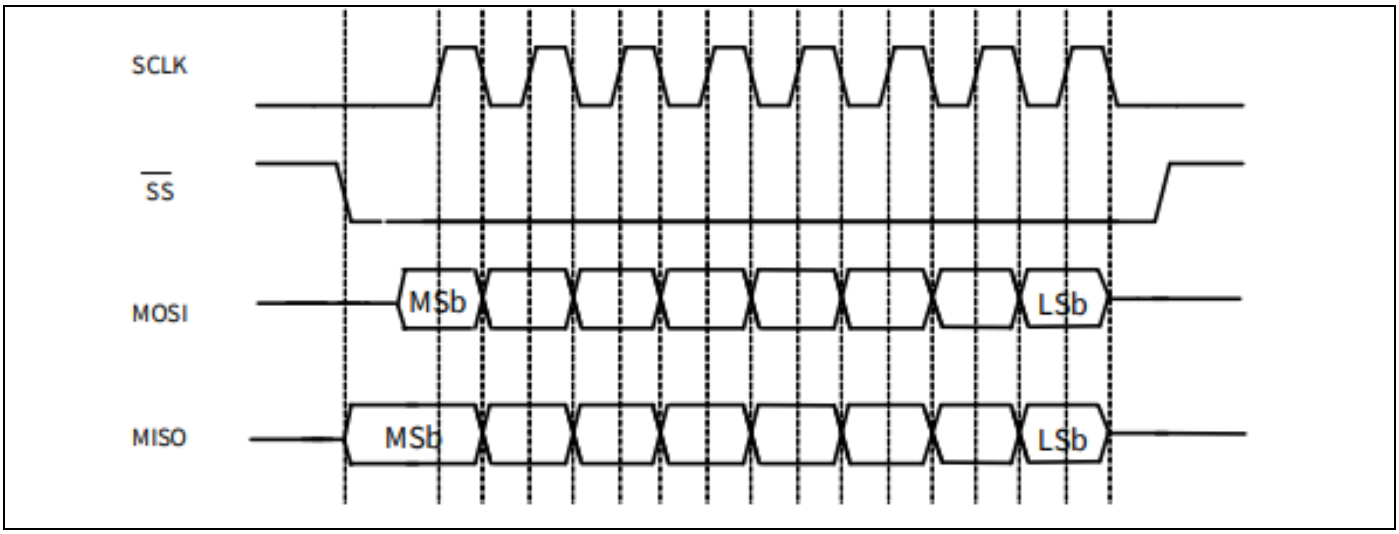



**Figure 121 SPI Motorola data transfer with CPOL = 0 and CPHA = 0**

The signals used for SPI are Master Out Slave In (MOSI), Master In Slave Out (MISO), Slave Select (SS), and the clock (SCLK). Since there is a signal for each direction, most SPI modes support full duplex communication. The slave select signal is driven by the master to indicate to a slave that the communication on the bus is directed at the slave. This allows for multiple SPI slaves on a single bus.

## Serial Communication Block with SPI

The PSOC™ SCB supports many types of SPI protocols like Motorola SPI, T.I. SPI, and National Semiconductor (Microwire) SPI. It can support master and slave functionality, multiple slave select lines, data frame sizing from 4 to 16 bits, and MSB or LSB bit order. The clock polarity (CPOL) and clock phase (CPHA) are configurable (depending on the SPI protocol chosen). See table 3 for more details on the different clock modes.

**Table 3 SPI clock modes**

| Mode | CPOL | CPHA | SCLK Idle State | Data Driven | Data Captured |
|---|---|---|---|---|---|
| 0 | 0 | 0 | Low | Falling edge of SCLK | Rising edge of SCLK |
| 1 | 0 | 1 | Low | Rising edge of SCLK | Falling edge of SCLK |
| 2 | 1 | 0 | High | Rising edge of SCLK | Falling edge of SCLK |
| 3 | 1 | 1 | High | Falling edge of SCLK | Rising edge of SCLK |

## 7.3 SPI master

1. The project created in the first exercise can be used, or the instructions shown there can be used to create a new project.
2. Once the project has been opened, navigate to the Eclipse IDE Quick Panel and open the Device Configurator. Navigate to the **Peripherals** tab and enable a SCB for SPI mode based on which SCB is associated with SPI pins that are available on the evaluation kit

    a. When using a PSOC™ 6 evaluation kit, see the appendix regarding setting the system clocks on PSOC™ 6 to get the SPI baud rate to work properly

    b. When using a PSOC™ Edge evaluation kit, adjusting the oversampling rate to 10 will allow for the input clock to be set such that the data rate of 1000 kbps is achievable

**Table 4 SCB associated with the SPI pins available on headers**

| Evaluation Kit | SCB | MOSI GPIO | MISO GPIO | SCLK | SS |
|---|---|---|---|---|---|
| CY8CKIT-041S-MAX | SCB2 | P10.0 | P10.1 | P10.2 | P10.3 |
| CY8CKIT-062S2-43012 | SCB6 | P12.0 | P12.1 | P12.2 | P12.3 |
| KIT_PSE84_EVAL | SCB10 | P16.1 | P16.2 | P16.0 | P16.3 |

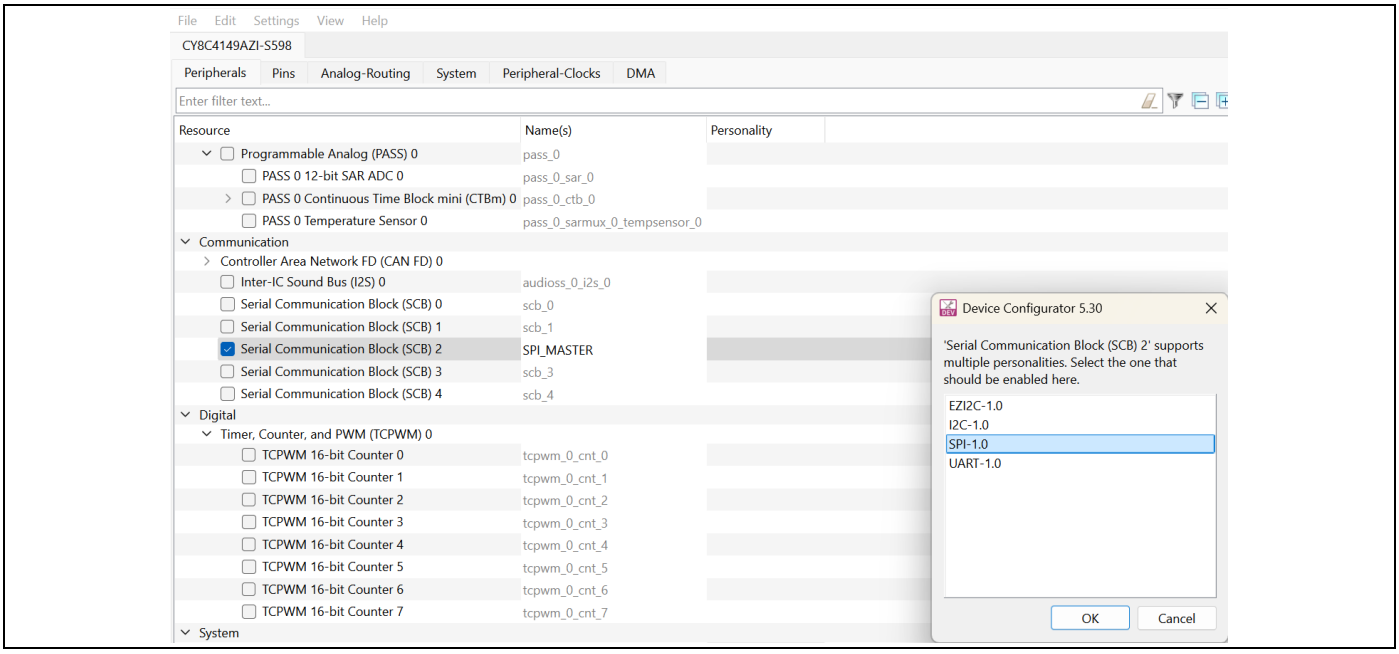



**Figure 122 Enabling a SCB for SPI mode**

## Serial Communication Block with SPI

3. Configure the SCB for SPI master mode, leave the sub mode set to Motorola, leave the SCLK mode as CPHA = 0 and CPOL = 0, leave the data rate at 1000 kbps, leave the bit order set to MSB first with a bit width of 8 for RX and TX, select an unused peripheral clock, and the GPIOs according to your hardware

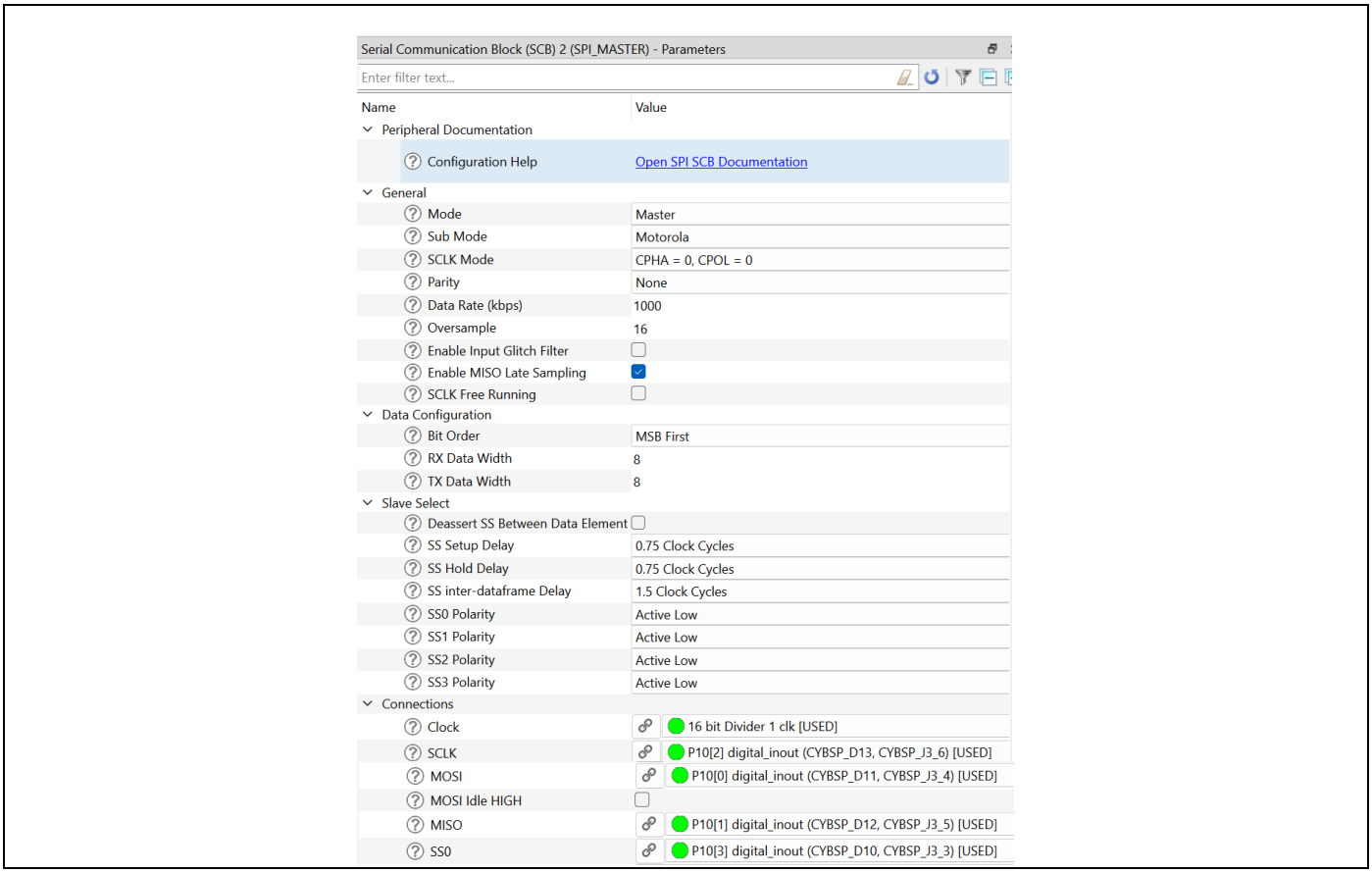



**Figure 123 Configuring a SCB for SPI slave mode**

4. Follow the steps here to enable a SCB for UART mode to allow for printing data to a terminal
    a. Note that it's best for the SCB used for SPI and the SCB used for UART don't share a peripheral clock
5. Save the device configurator
6. Open the **SPI SCB Documentation** and find the documentation for initializing, configuring the interrupt, and enabling the SPI interface
    a. Note that the steps for configuring a SPI master and a SPI slave will be the same when using the high-level driver option

## Serial Communication Block with SPI

7. The device configurator generates the configuration structure as well as generates and adds in the code to assign and configure pins, assign the clock divider, and configure the data rate. The code in the documentation to initialize the SCB, configure the interrupt, and enable the SPI interface will need to be copied into the user application

    b. Note that the alias chosen in step 2 is used when providing the inputs to these APIs

```
#include <stdio.h>

/* Allocate context for SPI operation */
cy_stc_scb_spi_context_t spiContext;

void spi_isr(void)
{
        Cy_SCB_SPI_Interrupt(SPI_MASTER_HW, &spiContext);
}

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

        /* Configure SPI to operate */
        (void) Cy_SCB_SPI_Init(SPI_MASTER_HW, &SPI_MASTER_config, &spiContext);

        /* Populate configuration structure */
        const cy_stc_sysint_t spi_intr_config =
        {
                .intrSrc      = SPI_MASTER_IRQ,
                .intrPriority = 3U,
        };

        /* Hook interrupt service routine and enable interrupt */
        (void)Cy_SysInt_Init(&spi_intr_config, &spi_isr);
        /* Enable interrupt in NVIC */
        NVIC_EnableIRQ(SPI_MASTER_IRQ);

        /* Enable the SPI Master block */
        Cy_SCB_SPI_Enable(SPI_MASTER_HW);
```

**Figure 124 Adding user code to initialize and enable the SPI master block**

8. We will need to add in the code to enable the UART SCB as well

```
Cy_SCB_UART_Init(CYBSP_UART_HW, &CYBSP_UART_config, NULL);

Cy_SCB_UART_Enable(CYBSP_UART_HW);

Cy_SCB_UART_PutString(CYBSP_UART_HW, "SPI Master\r\n");
```

**Figure 125 Adding user code to initialize and enable the debug UART**

9. Navigate back to the SPI PDL documentation and look for the **High-Level API** portion. This shows example code for using the **Cy_SCB_SPI_Transfer** and **Cy_SCB_SPI_GetTransferStatus** APIs to transfer bytes from the master to the slave
10. Let's add a GPIO input using the steps found here as a trigger for initiating the transfer
11. Update the main for loop

```
cy_en_scb_spi_status_t master_status;
uint8_t tx_buffer[4] = {0xDE, 0xAD, 0xBE, 0xEF};
uint8_t rx_buffer[4];

for (;;)
{
        if (0UL == Cy_GPIO_Read(CYBSP_SW2_PORT, CYBSP_SW2_PIN))
        {
                /* Initiate SPI Master write transaction. */
                master_status = Cy_SCB_SPI_Transfer(SPI_MASTER_HW, tx_buffer, rx_buffer, 4, &spiContext);

                if (CY_SCB_SPI_BAD_PARAM == master_status)
                {
                        CY_ASSERT(0);
                }

                /* Blocking wait for transfer completion */
                while (0UL != (CY_SCB_SPI_TRANSFER_ACTIVE & Cy_SCB_SPI_GetTransferStatus(SPI_MASTER_HW,
                                                                                         &spiContext)))
                {
                }

                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Send Bytes: ");

                for (uint8_t i = 0; i < 4; i++)
                {
                        char print[8];
                        sprintf(print, "0x%x; ", tx_buffer[i]);
                        Cy_SCB_UART_PutString(CYBSP_UART_HW, print);
                }
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "\r\n");

                /* Simple de-bounce on the GPIO press. */
                Cy_SysLib_Delay(500);

        }
}
```

**Figure 126 SPI Master transfer 4 bytes**

12. Program the device, open a terminal, press the button and observe the output

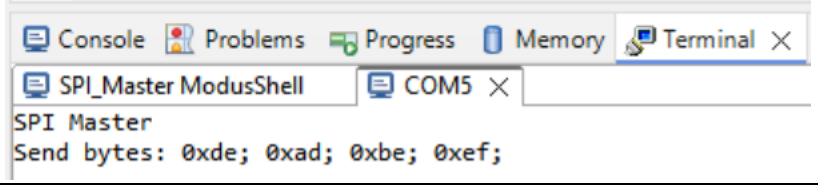



**Figure 127 SPI Master transfer 4 bytes**

## Serial Communication Block with SPI

13. Let's add a function to print a buffer so that we can easily print the RX and TX buffers

```
void print_buffer(char * message, uint8_t * buffer, uint8_t size)
{
	Cy_SCB_UART_PutString(CYBSP_UART_HW, message);
	for (uint8_t i = 0; i < size; i++)
	{
		char print[8];
		sprintf(print, "0x%x; ", buffer[i]);
		Cy_SCB_UART_PutString(CYBSP_UART_HW, print);
	}
	Cy_SCB_UART_PutString(CYBSP_UART_HW, "\r\n");
}
```

**Figure 128 Print buffer function**

```
cy_en_scb_spi_status_t master_status;
uint8_t tx_buffer[4] = {0xDE, 0xAD, 0xBE, 0xEF};
uint8_t rx_buffer[4];

for (;;)
{
	if (0UL == Cy_GPIO_Read(CYBSP_SW2_PORT, CYBSP_SW2_PIN))
	{
		/* Initiate SPI Master write transaction. */
		master_status = Cy_SCB_SPI_Transfer(SPI_MASTER_HW, tx_buffer, rx_buffer, 4, &spiContext);

		if (CY_SCB_SPI_BAD_PARAM == master_status)
		{
			CY_ASSERT(0);
		}

		/* Blocking wait for transfer completion */
		while (0UL != (CY_SCB_SPI_TRANSFER_ACTIVE & Cy_SCB_SPI_GetTransferStatus(SPI_MASTER_HW, &spiContext)))
		{
		}

		print_buffer("Send bytes: ", tx_buffer, 4);
		print_buffer("Receive bytes: ", rx_buffer, 4);

		/* Simple de-bounce on the GPIO press. */
		Cy_SysLib_Delay(500);

	}
}
```

**Figure 129 Print buffer function usage**

14. Program the device, press the button, and observe the new output

    c. Note that since a SPI slave has not been connected, the RX data will be 0x00 as the MISO pin is floating

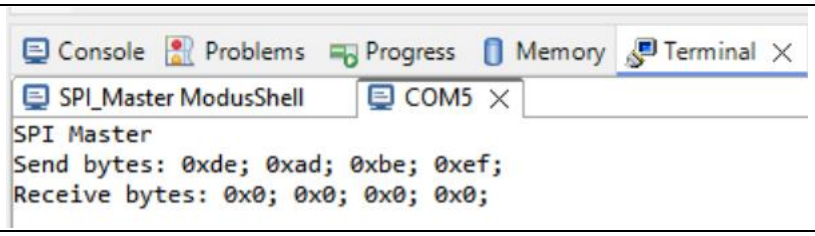



**Figure 130 Terminal output while printing SPI master send and receive data**

## 7.4 SPI slave

**Table 5 SCB associated with the SPI pins available on headers**

| Evaluation Kit | SCB | MOSI GPIO | MISO GPIO | SCLK | SS |
|---|---|---|---|---|---|
| CY8CKIT-041S-MAX | SCB2 | P10.0 | P10.1 | P10.2 | P10.3 |
| CY8CKIT-062S2-43012 | SCB6 | P12.0 | P12.1 | P12.2 | P12.3 |
| KIT_PSE84_EVAL | SCB10 | P16.1 | P16.2 | P16.0 | P16.3 |

1. The project created in the first exercise can be used, or the instructions shown there can be used to create a new project.
2. Once the project has been opened, navigate to the Eclipse IDE Quick Panel and open the Device Configurator. Navigate to the **Peripherals** tab and enable a SCB for SPI mode
    a. Note: When using a PSOC™ 6 evaluation kit, see the appendix regarding setting the system clocks on PSOC™ 6 to get the SPI baud rate to work properly

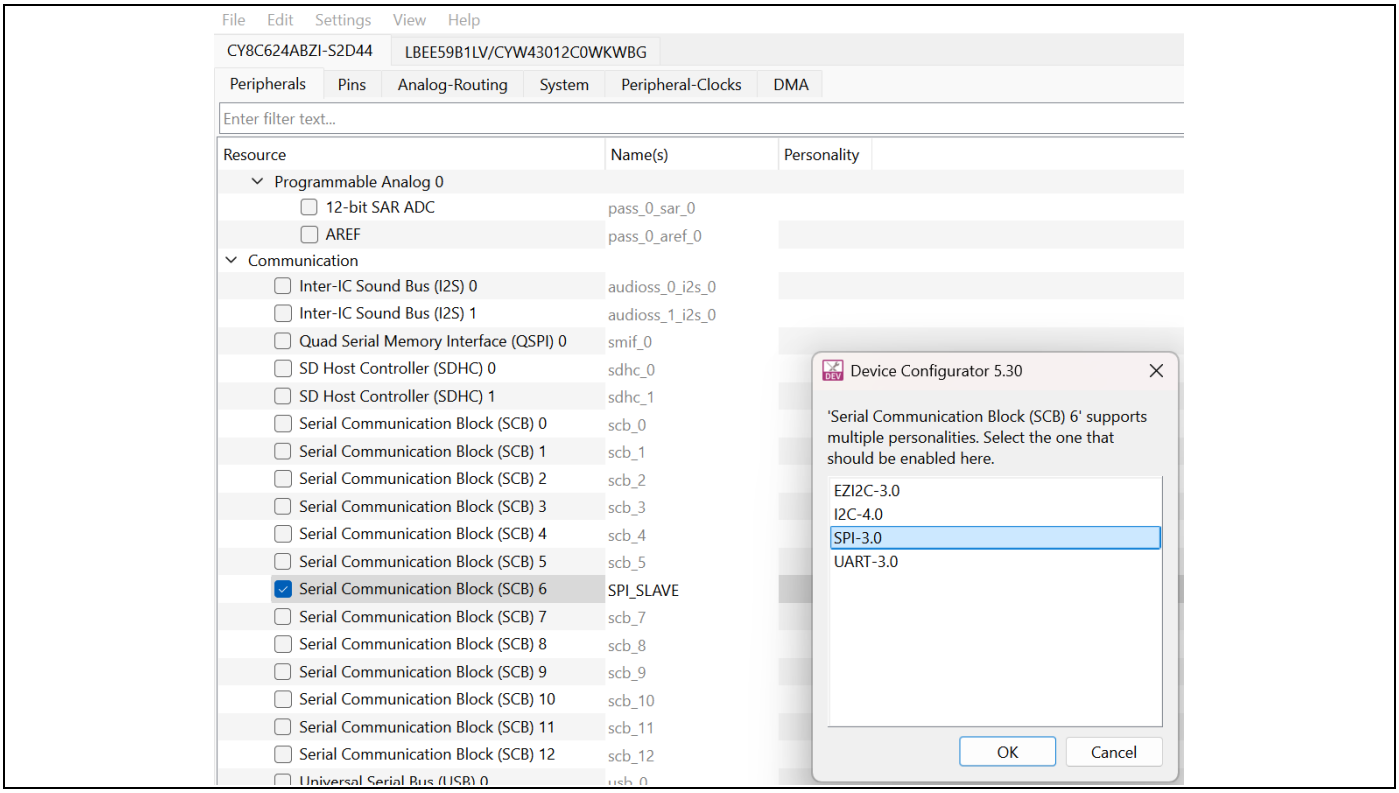



**Figure 131 Enabling a SCB for SPI mode**

## Serial Communication Block with SPI

3. Configure the SCB for SPI slave mode, leave the sub mode set to Motorola, leave the SCLK mode as CPHA = 0 and CPOL = 0, leave the data rate at 1000 kbps, leave the bit order set to MSB first with a bit width of 8 for RX and TX, select an unused peripheral clock, and the GPIOs according to your hardware

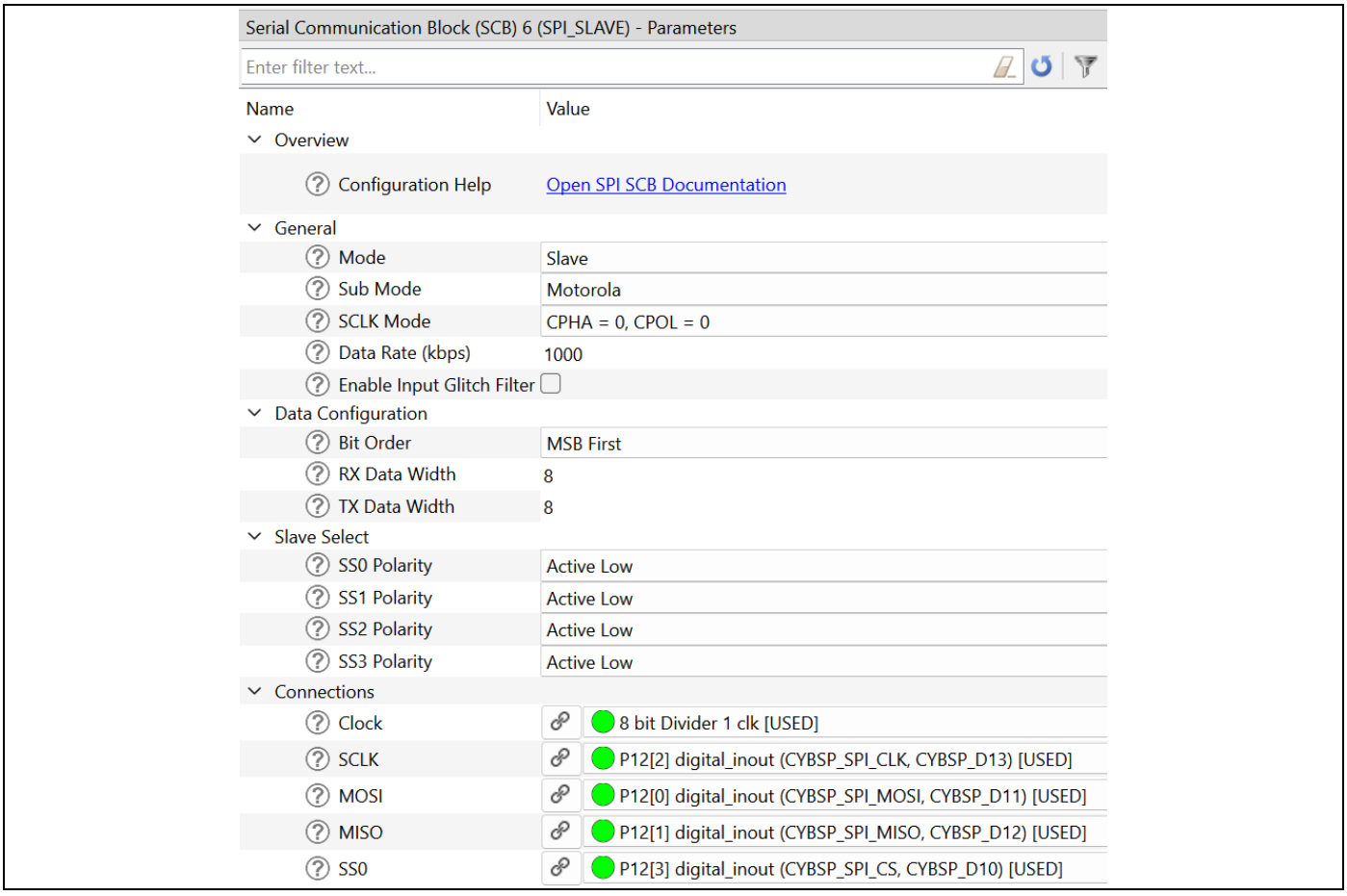



**Figure 132 Configuring a SCB for SPI slave mode**

4. Follow the steps here to enable a SCB for UART mode to allow for printing data to a terminal
    a. Note that it's best for the SCB used for SPI and the SCB used for UART don't share a peripheral clock
5. Save the device configurator
6. Open the SPI SCB Documentation and find the documentation for initializing, configuring the interrupt, and enabling the SPI interface
    a. Note that the steps for configuring a SPI master and a SPI slave will be the same when using the high-level driver option

7. The device configurator generates the configuration structure as well as generates and adds in the code to assign and configure pins, assign the clock divider, and configure the data rate. The code in the documentation to initialize the SCB, configure the interrupt, and enable the SPI interface will need to be copied into the user application
    a. Note that the alias chosen in step 2 is used when providing the inputs to these APIs

```
#include <stdio.h>

/* Allocate context for SPI operation */
cy_stc_scb_spi_context_t spiContext;

void spi_sr(void)
{
        Cy_SCB_SPI_Interrupt(SPI_SLAVE_HW, &spiContext);
}
```

**Figure 133 Adding SPI ISR**

```
/* Configure SPI to operate */
(void) Cy_SCB_SPI_Init(SPI_SLAVE_HW, &SPI_SLAVE_config, &spiContext);

/* Populate configuration structure */
const cy_stc_sysint_t spiIntrConfig =
{
        .intrSrc      = SPI_SLAVE_IRQ,
        .intrPriority = 3U,
};

/* Hook interrupt service routine and enable interrupt */
(void) Cy_SysInt_Init(&spiIntrConfig, &spi_sr);
NVIC_EnableIRQ(spiIntrConfig.intrSrc);

/* Enable the SPI Slave block */
Cy_SCB_SPI_Enable(SPI_SLAVE_HW);
```

**Figure 134 Adding user code to initialize and enable the SPI slave block**

The initialization code should be placed after the **cybsp_init** and **enable_irq** function calls.

8. We will need to add in the code to enable the UART SCB as well

```
Cy_SCB_UART_Init(CYBSP_UART_HW, &CYBSP_UART_config, NULL);

Cy_SCB_UART_Enable(CYBSP_UART_HW);

Cy_SCB_UART_PutString(CYBSP_UART_HW, "SPI Slave\r\n");
```

**Figure 135 Adding user code to initialize and enable the debug UART**

9. Navigate back to the SPI PDL documentation and look for the **High-Level API** portion. This shows example code for using the **Cy_SCB_SPI_Transfer** and **Cy_SCB_SPI_GetTransferStatus** APIs to transfer bytes from the master to the slave

10. Let's add this user code to the infinite for loop of the main function along with code to print the received data using the debug UART

```
uint8_t rx_buffer[4];

for (;;)
{
        volatile cy_en_scb_spi_status_t status;
        /* Master: start a transfer. Slave: prepare for a transfer. */
        status = Cy_SCB_SPI_Transfer(SPI_SLAVE_HW, NULL, rx_buffer, 4, &spiContext);

        if (CY_SCB_SPI_SUCCESS == status)
        {
                /* Blocking wait for transfer completion */
                uint32_t transfer_status;
                do {
                        transfer_status = Cy_SCB_SPI_GetTransferStatus(SPI_SLAVE_HW, &spiContext);
                }while (0UL != (CY_SCB_SPI_TRANSFER_ACTIVE & transfer_status));
        }

        Cy_SCB_UART_PutString(CYBSP_UART_HW, "Receive Bytes: ");
        for (uint8_t i = 0; i < 4; i++)
        {
                char print[8];
                sprintf(print, "0x%x; ", rx_buffer[i]);
                Cy_SCB_UART_PutString(CYBSP_UART_HW, print);
        }
        Cy_SCB_UART_PutString(CYBSP_UART_HW, "\r\n");
}
```

**Figure 136 SPI slave receive 4 bytes**

11. We need to connect the SPI slave to the SPI master created in the first portion of this lab. Be sure that the GPIOs used are operating at the same voltage. The PSOC™ 4100S Max evaluation kit can operate at 5V or 3.3V by adjusting the J10 jumper. The PSOC™ 6 evaluation kit can operate at 3.3V or 1.8V by adjusting the J14 jumper. The PSOC™ Edge evaluation kit SPI GPIOs can operate at 3.3V or 1.8V by adjusting the J23 jumper.

**Table 6 SPI pinout for PSOC™ evaluation kits**

| Evaluation kit | MOSI | MISO | SCLK | SS |
|---|---|---|---|---|
| CY8CKIT-041S-MAX | P10[0] (D11) | P10[1] (D12) | P10[2] (D13) | P10[3] (D10) |
| CY8CKIT-062S2-43012 | P12[0] (D11) | P12[1] (D12) | P12[2] (D13) | P12[3] (D10) |
| KIT_PSE84_EVAL | P16[1] (SPI.MOSI) | P16[2] (SPI.MISO) | P16[0] (SPI.SCK) | P16[3] (SPI.CS) |

12. Program the device, open a terminal, press the button on the SPI slave and observe the output
    a. Note that some evaluation kits cannot be programmed if other evaluation kits are connected. It is best practice to only connect the evaluation kit you are programming during the programming step

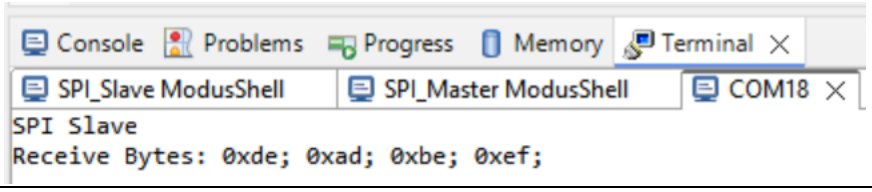



**Figure 137 Terminal output of a SPI slave receiving 4 bytes**

## 7.5 SPI slave with callbacks

The initial SPI slave implementation used a polled, blocking method. A more common and useful implementation is fully interrupt driven using callbacks to handle the SPI transactions.

1. Navigate to the SCB SPI PDL documentation and go to the **Interrupt** section, then look for the **Cy_SCB_SPI_RegisterCallback** API
2. Let's add a callback to service the **CY_SCB_SPI_TRANSFER_CMPLT_EVENT** complete event and enable sending data back to the master
    a. Make the **rx_buffer** and **tx_buffer** global variables
    b. Add the callback function that sets up a new transfer and flags the main code that new buffer data to print is available
    c. Register the callback after initializing the peripheral

```
/* Allocate context for SPI operation */
cy_stc_scb_spi_context_t spiContext;
volatile bool print_data;
uint8_t rx_buffer[4];
uint8_t tx_buffer[4] = {0xBE, 0xEF, 0xCA, 0xFE};


void spi_sr(void)
{
        Cy_SCB_SPI_Interrupt(SPI_SLAVE_HW, &spiContext);
}

void spi_callback(uint32_t event)
{
        if (CY_SCB_SPI_TRANSFER_CMPLT_EVENT == event)
        {
                Cy_SCB_SPI_Transfer(SPI_SLAVE_HW, tx_buffer, rx_buffer, 4, &spiContext);
                print_data = true;
        }
}
```

**Figure 138 SPI callback implementation**

```
/* Configure SPI to operate */
(void) Cy_SCB_SPI_Init(SPI_SLAVE_HW, &SPI_SLAVE_config, &spiContext);

Cy_SCB_SPI_RegisterCallback(SPI_SLAVE_HW, spi_callback, &spiContext);
```

**Figure 139 Register SPI callback**

3. Add in the print buffer function from the SPI Master portion

```
void print_buffer(char * message, uint8_t * buffer, uint8_t size)
{
        Cy_SCB_UART_PutString(CYBSP_UART_HW, message);
        for (uint8_t i = 0; i < size; i++)
        {
                char print[8];
                sprintf(print, "0x%x; ", buffer[i]);
                Cy_SCB_UART_PutString(CYBSP_UART_HW, print);
        }
        Cy_SCB_UART_PutString(CYBSP_UART_HW, "\r\n");
}
```

**Figure 140 Print buffer function**

4. Update the main function infinite for loop to only act when new buffer data to print is available

```
/* Setup the first transfer. */
Cy_SCB_SPI_Transfer(SPI_SLAVE_HW, tx_buffer, rx_buffer, 4, &spiContext);

for (;;)
{
        if(true == print_data)
        {
                print_data = false;
                print_buffer("Receive bytes: ", rx_buffer, 4);
                print_buffer("Send bytes: ", tx_buffer, 4);
        }
}
```

**Figure 141 User application code update to handle callback implementation**

5. Program the device, open the terminal, press the SPI Master button, and observe the output

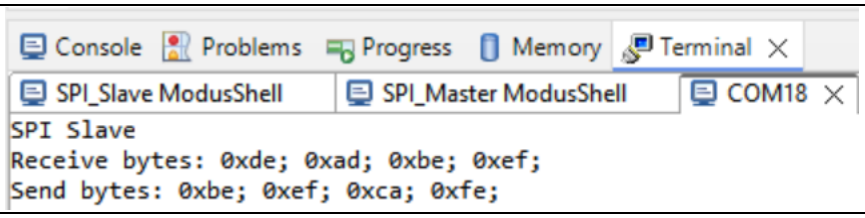



**Figure 142 Terminal output of callback implementation of SPI slave**

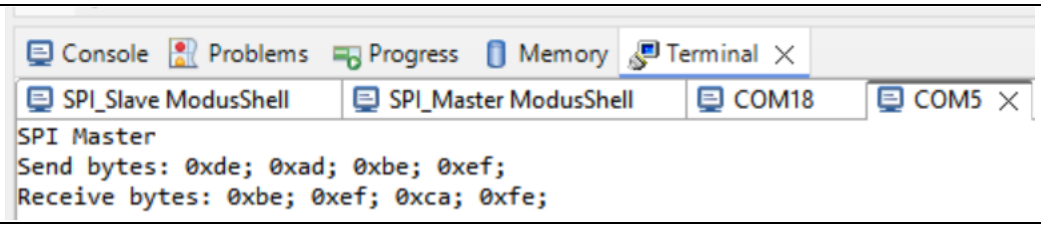



**Figure 143 Terminal output of SPI Master**

## 7.6 Conclusion

The user should now be able to use ModusToolbox™ to enable a SPI master or a SPI slave on a PSOC™ device. The user should also know how to access the PDL documentation for the SCB when configured as a SPI controller and how to implement a callback for the high-level version of the SPI PDL.

# 8 Watch dog timer

## 8.1 Objective

The objective of this lab is to demonstrate how to enable the watchdog timer (WDT) on PSOC™ devices using ModusToolbox™ and the peripheral driver library (PDL) libraries.

## 8.2 Description

The watchdog timer on PSOC™ devices can be used to handle error conditions where the device is unable to leave a task. The WDT is typically sourced from a low frequency clock (LFCLK) and can be configured for a few different use cases like a periodic interrupt or a system reset on the expiration of the timer.

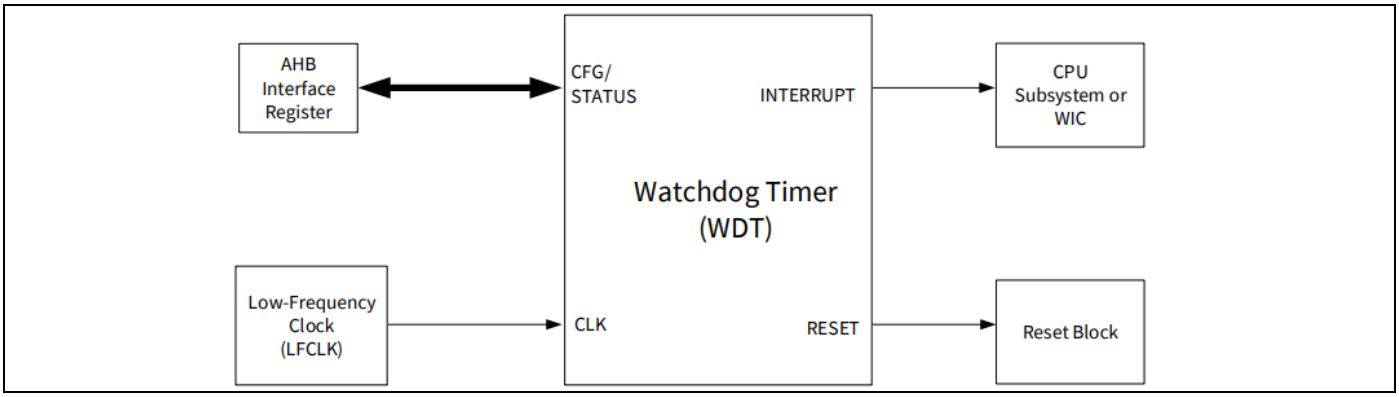



**Figure 144 PSOC™ Watchdog timer block diagram**

The WDT asserts a system reset to the device on the third WDT match event, unless it is periodically serviced in firmware.

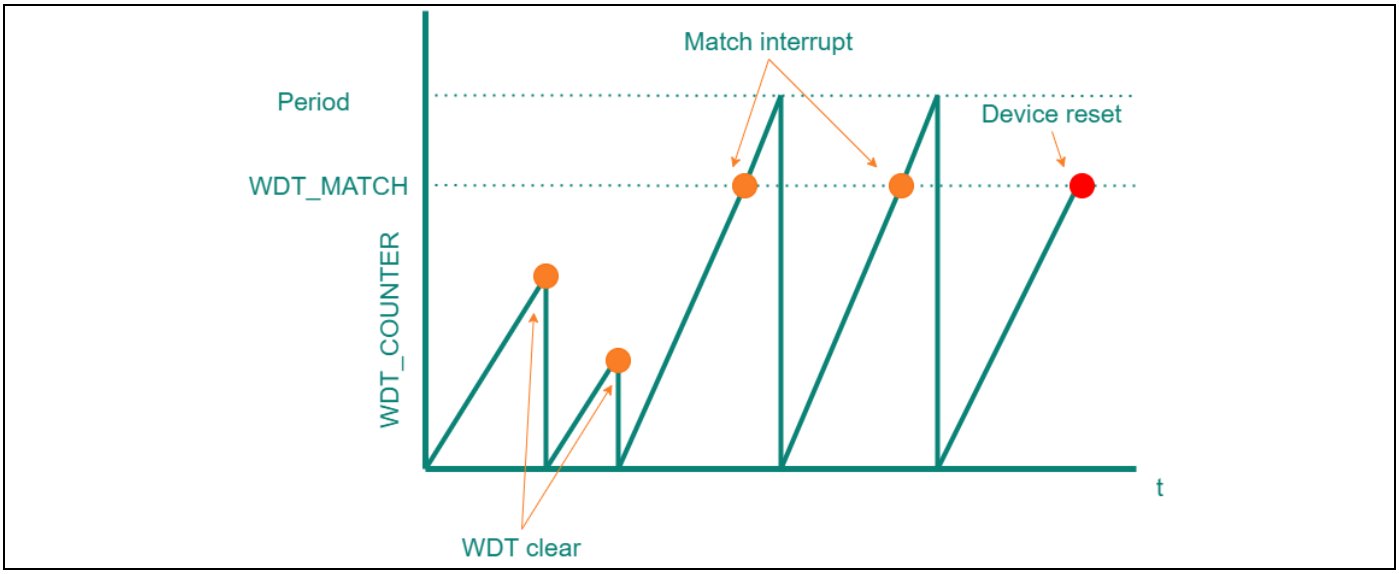



**Figure 145 PSOC™ Watchdog timer block diagram**

The WDT_COUNTER register provides the count value of the WDT. The WDT generates an interrupt when the count value in WDT_COUNTER equals the match value stored in the WDT_MATCH register, but it does not reset the count to '0'. Instead, the WDT keeps counting until it overflows (after 0xFFFF

## Watch dog timer

when the resolution is set to 16 bits) and rolls back to 0. When the count value again reaches the match value, another interrupt is generated. Note that the match count can be changed when the counter is running. A bit named WDT_MATCH in the SRSS_INTR register is set whenever the WDT interrupt occurs. This interrupt must be cleared by writing a '1' to the WDT_MATCH bit in SRSS_INTR to reset the watchdog. If the firmware does not reset the WDT for two consecutive interrupts, the third match event will generate a system reset. The IGNORE_BITS in the WDT_MATCH register can be used to reduce the entire WDT counter period. The ignore bits can specify the number of MSBs that need to be discarded.

## 8.3 WDT with a system reset on expiration

1. The project created in the first exercise can be used, or the instructions shown there can be used to create a new project.
2. Once the project has been opened, navigate to the Eclipse IDE Quick Panel and open the Device Configurator. Navigate to the **Peripherals** tab and enable the **Watchdog Timer**
    a. Note that on CAT 1 devices, the WDT cannot be configured in the device configurator and must be initialized in user code using the PDL APIs

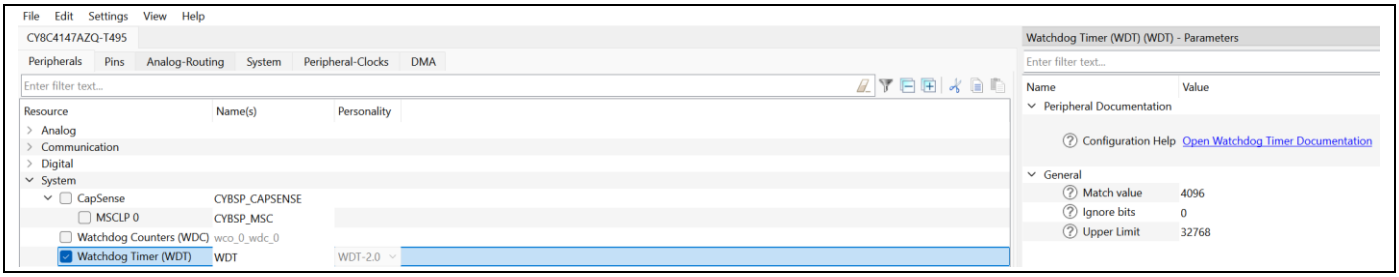



**Figure 146 Enabling the WDT in the device configurator**

3. Follow the steps here to enable a SCB for UART mode to allow for printing information to a terminal
    a. Let's add a GPIO input using the steps found here as a trigger for causing an infinite loop
    b. Save the device configurator, then open the **Watchdog Timer Documentation** and navigate to the Functions page. The **Cy_WDT_Enable** and **Cy_WDT_ClearWatchdog** APIs will be used.
4. The device configurator will configure the WDT settings as shown in the parameters window, but it will not enable the WDT, the user application code must enable the WDT and clear the timer

```
Cy_SCB_UART_Init(CYBSP_UART_HW, &CYBSP_UART_config, NULL);

Cy_SCB_UART_Enable(CYBSP_UART_HW);

Cy_SCB_UART_PutString(CYBSP_UART_HW, "WDT with system reset on expiration \r\n");

/* Enables the watchdog timer reset generation. */
Cy_WDT_Enable();
/* Enable global interrupts */
__enable_irq();

for (;;)
{
        if (0UL == Cy_GPIO_Read(CYBSP_SW4_PORT, CYBSP_SW4_PIN))
        {
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Button pressed, entering infinite loop \r\n");
                while(1);
        }
        Cy_WDT_ClearWatchdog();
}
```

**Figure 147 User code to enable and clear the WDT for PSOC™ 4**

## Watch dog timer

```
Cy_SCB_UART_Init(CYBSP_UART_HW, &CYBSP_UART_config, NULL);

Cy_SCB_UART_Enable(CYBSP_UART_HW);

Cy_SCB_UART_PutString(CYBSP_UART_HW, "WDT with system reset on expiration \r\n");

Cy_WDT_Unlock();

Cy_WDT_SetMatch(0x8000);
Cy_WDT_ClearInterrupt();
Cy_WDT_Enable();
Cy_WDT_Lock();

/* Enable global interrupts */
__enable_irq();

for(;;)
{
        if (0UL == Cy_GPIO_Read(CYBSP_SW4_PORT, CYBSP_SW4_PIN))
        {
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Button pressed, entering infinite loop \r\n");
                while(1);
        }
        Cy_WDT_ClearWatchdog();
}
```

**Figure 148 User code to enable and clear the WDT for PSOC™ 6**

```
Cy_SCB_UART_Init(CYBSP_UART_HW, &CYBSP_UART_config, NULL);

Cy_SCB_UART_Enable(CYBSP_UART_HW);

Cy_SCB_UART_PutString(CYBSP_UART_HW, "WDT with system reset on expiration \r\n");

Cy_WDT_Unlock();

/* PSOC™ Edge default match bits is 0,
*  set to 14 so that first 14 match bits are used. Only for PSOC™ Edge. */
Cy_WDT_SetMatchBits(14);

Cy_WDT_SetMatch(0x8000);
Cy_WDT_ClearInterrupt();
Cy_WDT_Enable();
Cy_WDT_Lock();

/* Enable global interrupts */
__enable_irq();

for(;;)
{
        if (0UL == Cy_GPIO_Read(CYBSP_SW4_PORT, CYBSP_SW4_PIN))
        {
                Cy_SCB_UART_PutString(CYBSP_UART_HW, "Button pressed, entering infinite loop \r\n");
                while(1);
        }
        Cy_WDT_ClearWatchdog();
}
```

**Figure 149 User code to enable and clear the WDT for PSOC™ Edge**

5. Program the device, open a terminal, and observe the output

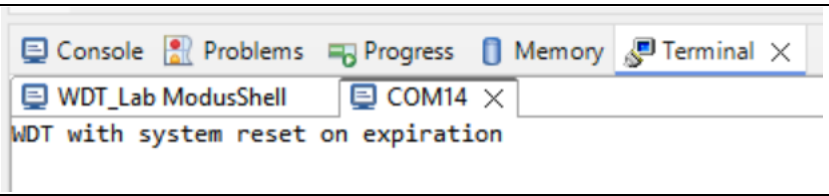



**Figure 150 Terminal output after programming**

6. Press the user button and see that it enters the infinite loop, then resets itself once the WDT expires

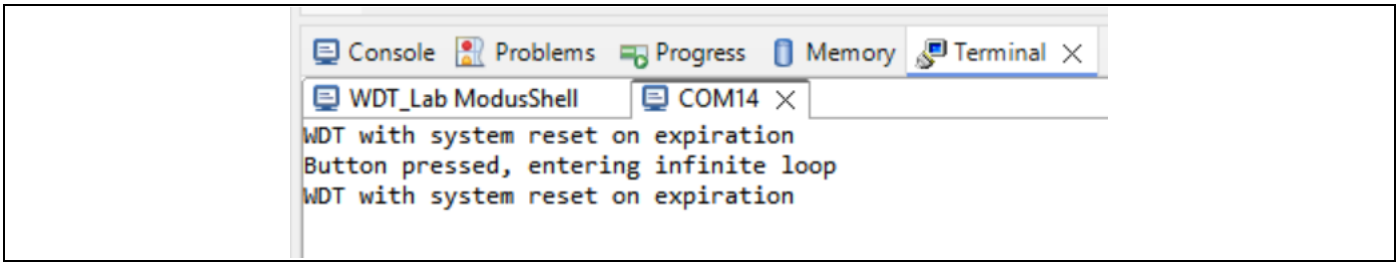



**Figure 151 Terminal output after pressing the user button and the WDT reset**

## 8.4 Conclusion

The user should now be able to configure, enable, and clear the WDT for PSOC™ devices using ModusToolbox™.

# 9 Appendix

## 9.1 Disabling the HAL on PSOC™ 6 BSPs

The hardware abstraction layer (HAL) is not required for this training and prevents the use of PDLs with certain libraries. For these reasons, it is best to disable the HAL if a PSOC6 is used during this training.

1. Once the PSOC6 Empty App project has been created, open the BSP assistant from the Eclipse IDE quick panel.

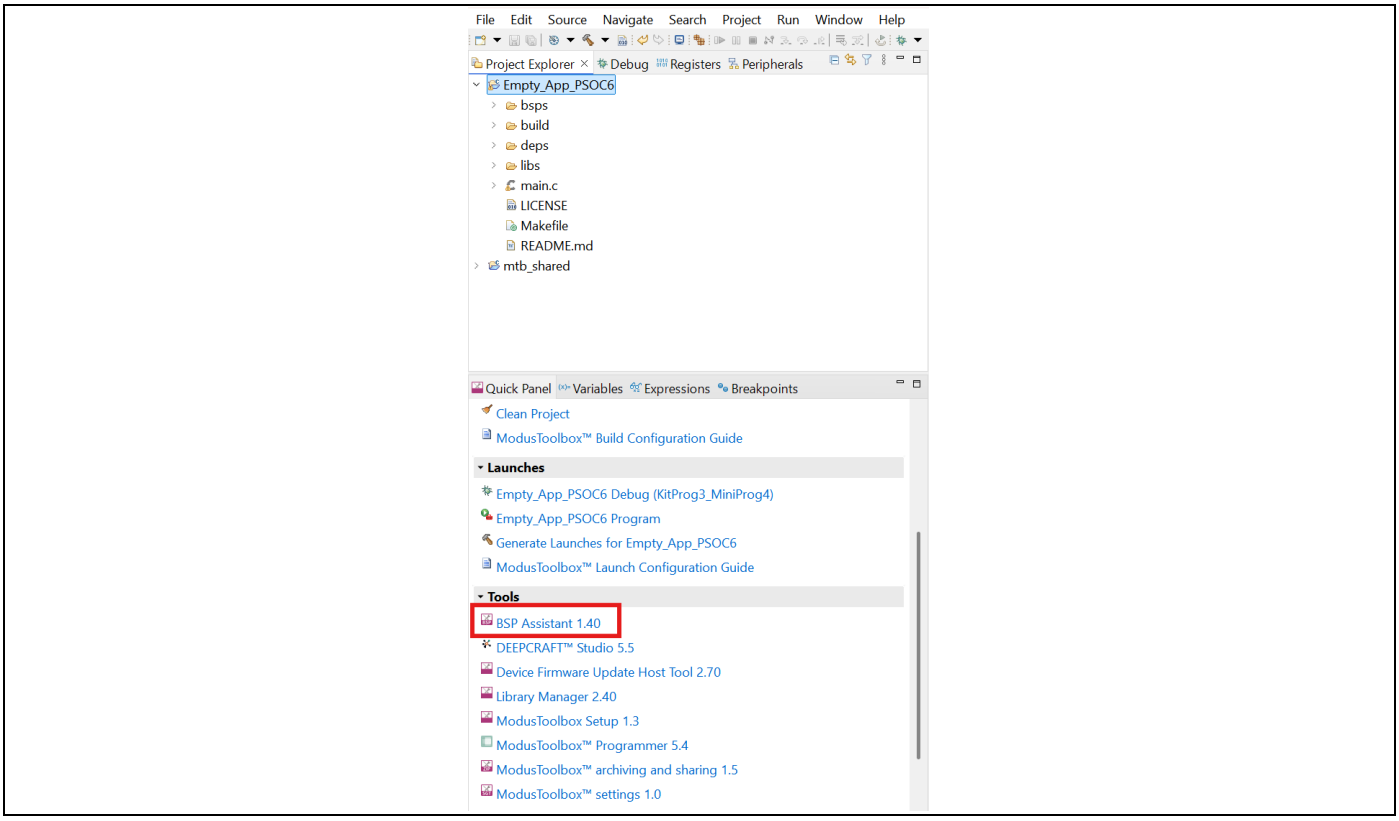



**Figure 152 Launching the BSP Assistant from the Eclipse IDE Quick Panel**

2. In the device properties, change the “Using HAL Interface” option to “N/A”

## Appendix

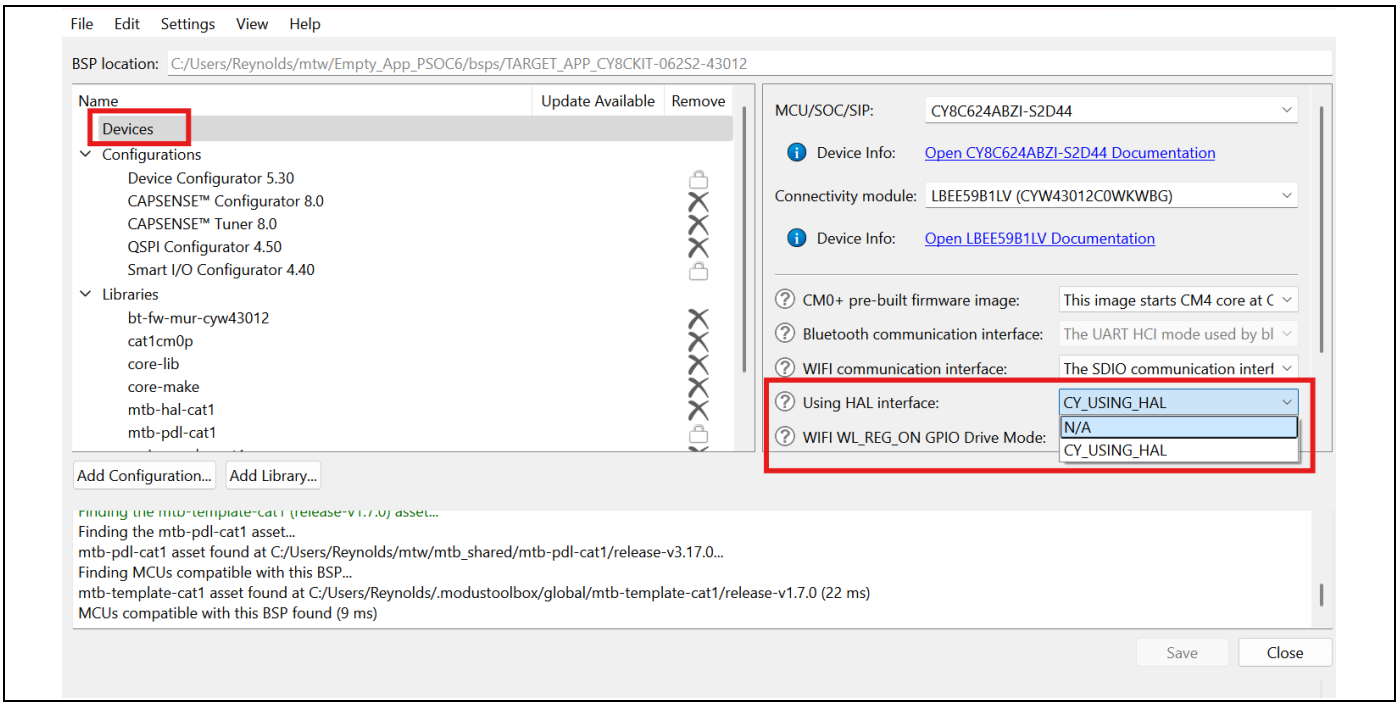



**Figure 153 Disabling the HAL for the device**

3. Remove the HAL libraries from the project by pressing the “X” next to the library

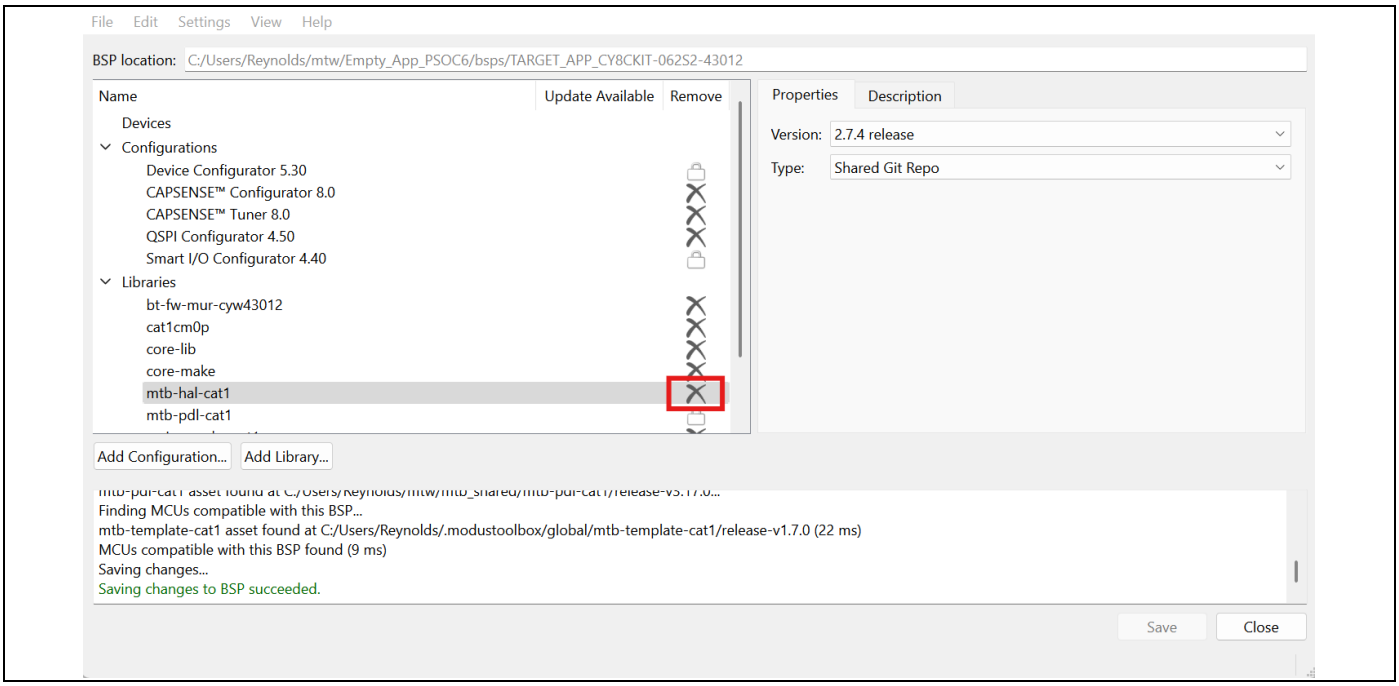



**Figure 154 Removing the HAL libraries from the project using the BSP Assistant tool**

4. Save the changes and close the BSP assistant tool
5. When closing the BSP assistant tool, a pop-up will appear instructing you to update the libraries from the library manager. Open the library manager from the Eclipse IDE quick panel and press the “Update” button

## Appendix

## 9.2 Disabling FreeRTOS from PSOC™ Edge Empty Application

The PSOC™ Edge Empty Application enables FreeRTOS and a MCWDT for both the CM33_NS and CM55 projects to show how to blink an LED in both cores. To disable these items, modifications to the main.c file, included libraries, and makefiles for the corresponding projects is required.

1. Once the project has been created, open the library manager from the Eclipse IDE Quick Panel

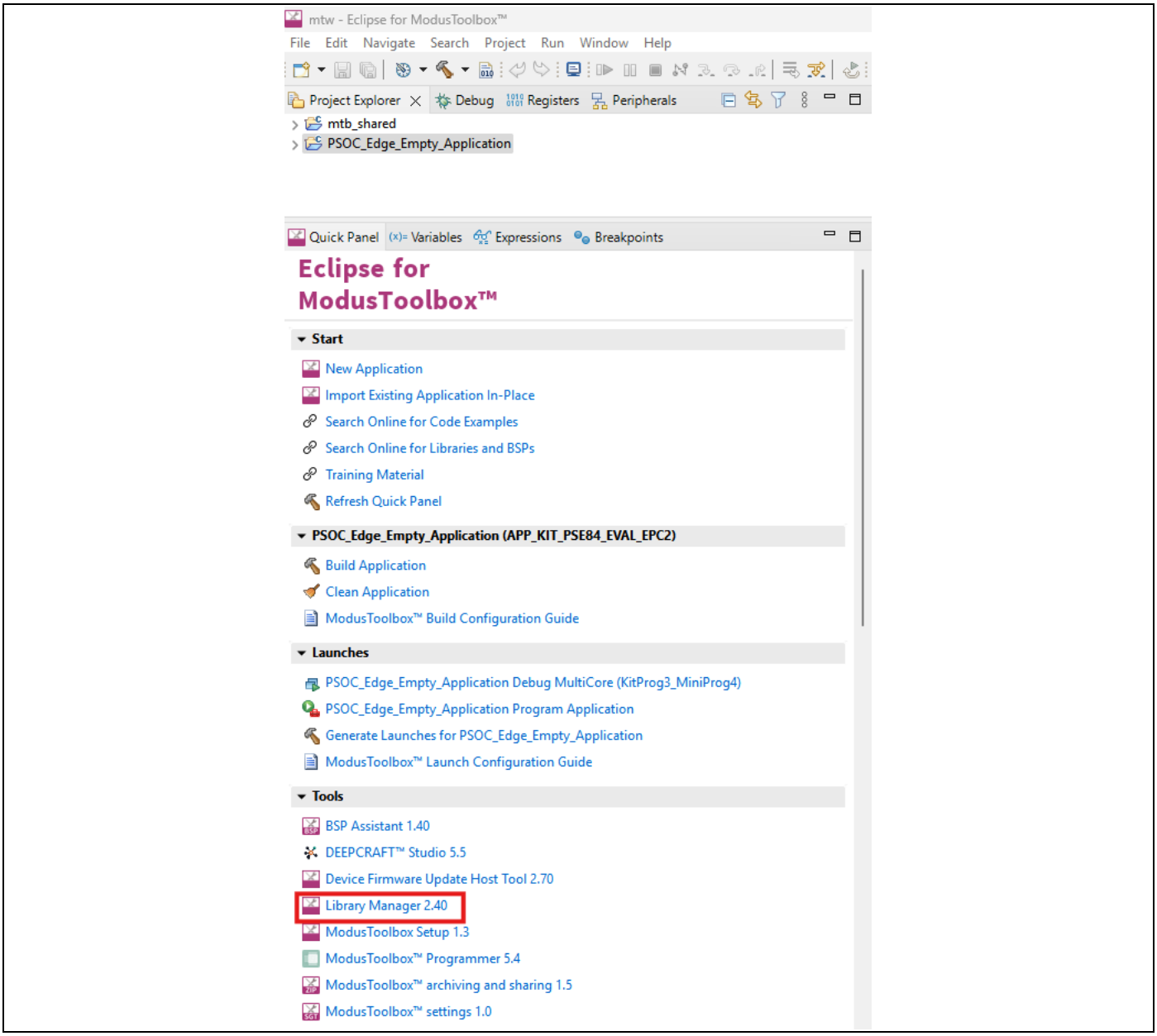



**Figure 155 Launching the library manager from the Eclipse IDE Quick Panel**

2. Delete the “freertos” library from the the “proj_cm33_ns” and “proj_cm55” libraries then press the **Update** button

## Appendix

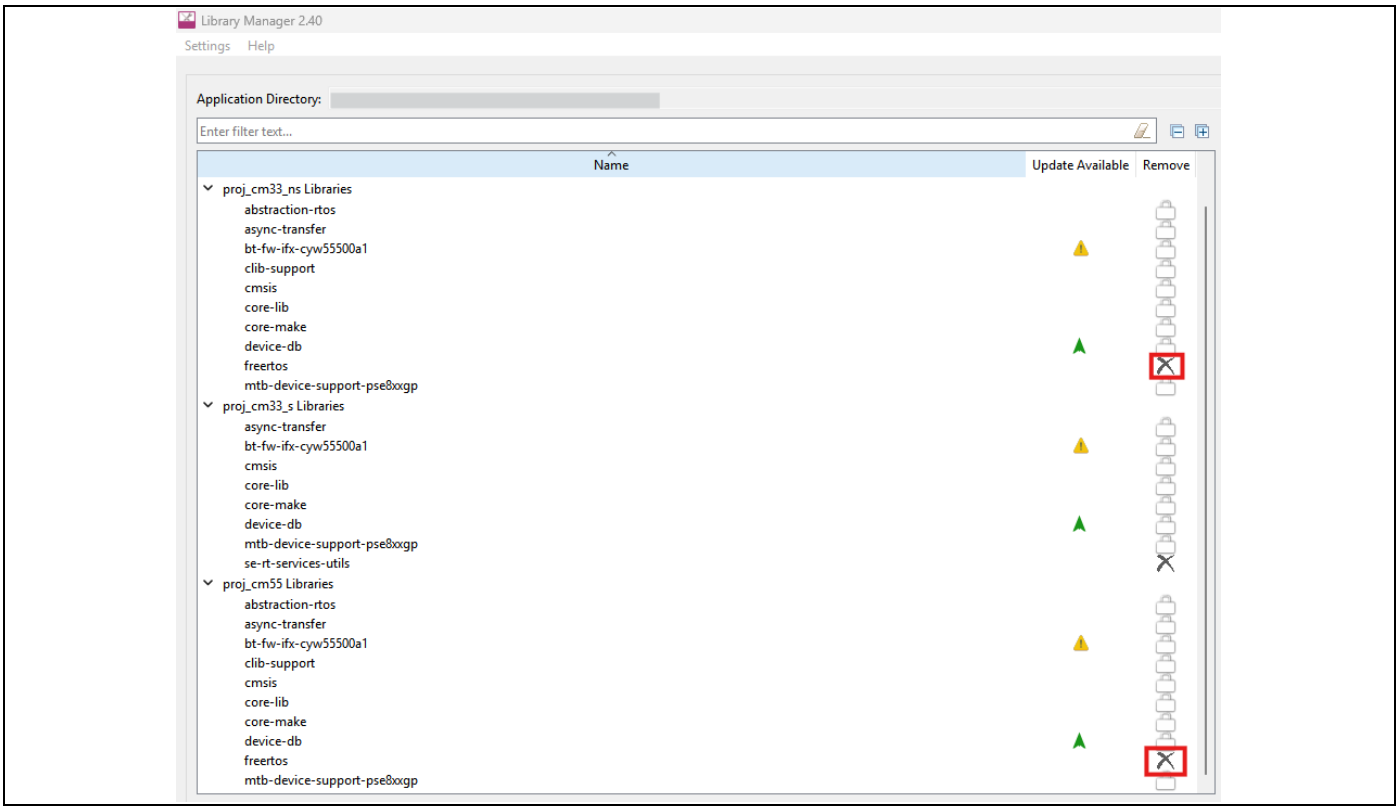



**Figure 156 Removing the freetos library from the library manager**

3. Navigate to the main.c file for the corresponding project

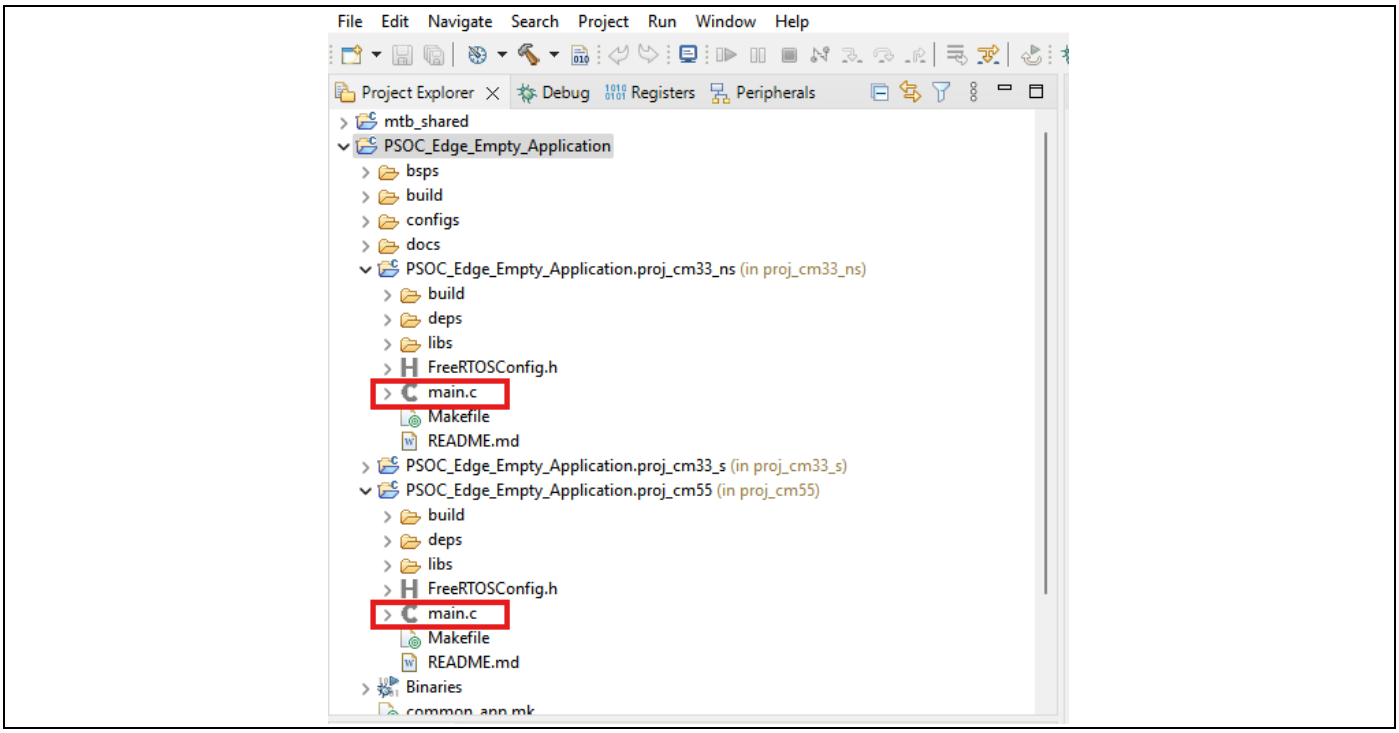



**Figure 157 Main.c file in the Eclipse IDE Project Explorer**

4. Remove “#include FreeRTOS.h”, “#include task.h”, “#include cyabs_rtos.h”, and “#include cyabs_rtos.impl.h”

## Appendix

```
main.c
/* Header Files
*******************************************

#include "cybsp.h"

#include "FreeRTOS.h"
#include "task.h"
#include "cyabs_rtos.h"
#include "cyabs_rtos_impl.h"
```

**Figure 158 Removing FreeRTOS related header files**

5. Remove all “BLINKY_LED_TASK” macros and the macros for the LP timer

```
main.c
/*******************************************************************************
 * Macros
 ******************************************************************************/
#define BLINKY_LED_TASK_NAME                ("CM33 Blinky Task")
#define BLINKY_LED_TASK_STACK_SIZE          (configMINIMAL_STACK_SIZE * 4)
#define BLINKY_LED_TASK_PRIORITY            (configMAX_PRIORITIES - 1)
#define BLINKY_LED_TASK_DELAY_MSEC          (1000)

/* The timeout value in microsecond used to wait for the CM55 core to be booted.
 * Use value 0U for infinite wait till the core is booted successfully.
 */
#define CM55_BOOT_WAIT_TIME_USEC            (10U)

/* Enabling or disabling a MCWDT requires a wait time of upto 2 CLK_LF cycles
 * to come into effect. This wait time value will depend on the actual CLK_LF
 * frequency set by the BSP.
 */
#define LPTIMER_0_WAIT_TIME_USEC            (62U)

/* Define the LPTimer interrupt priority number. '1' implies highest priority.
 */
#define APP_LPTIMER_INTERRUPT_PRIORITY      (1U)
```

**Figure 159 Removing FreeRTOS related macros**

6. Remove the “lptimer_obj” global variable

```
/*******************************************************************************
 * Global Variables
 ******************************************************************************/

/* LPTimer HAL object */
static mtb_hal_lptimer_t lptimer_obj;
```

**Figure 160 Removing FreeRTOS related global variables**

7. Remove the functions “cm33_blinky_task”, “lptimer_interrupt_handler”, and “setup_tickless_idle_timer” as well as where they are invoked in the main function

## Appendix

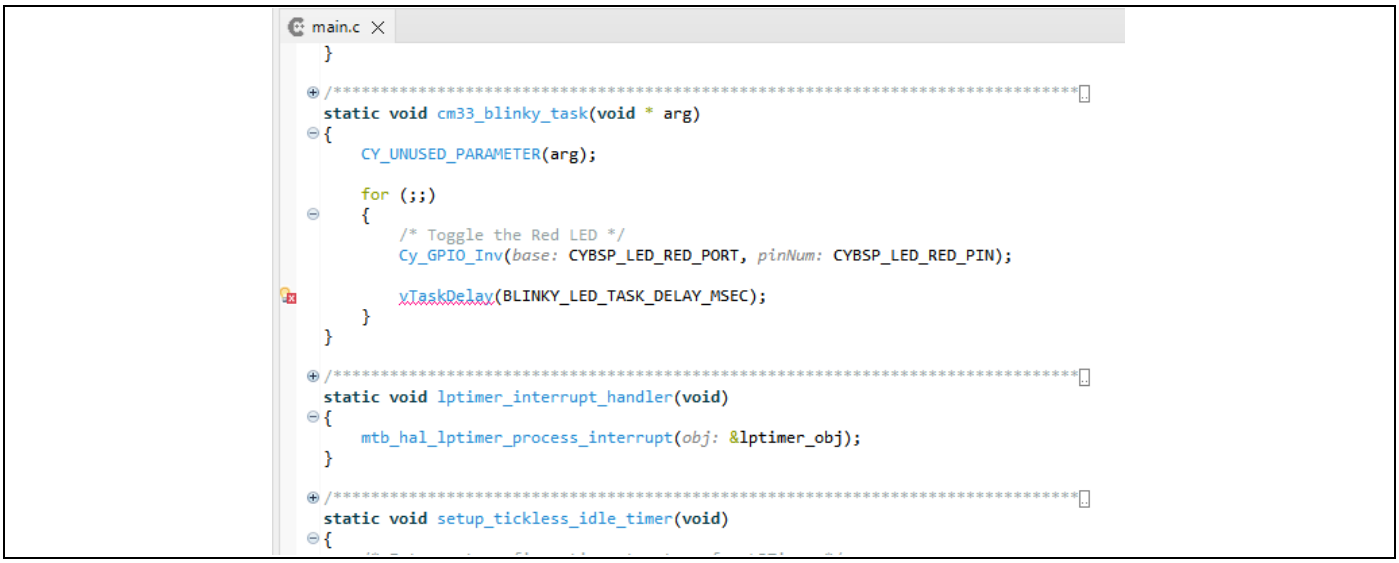



**Figure 161 Removing FreeRTOS related functions**

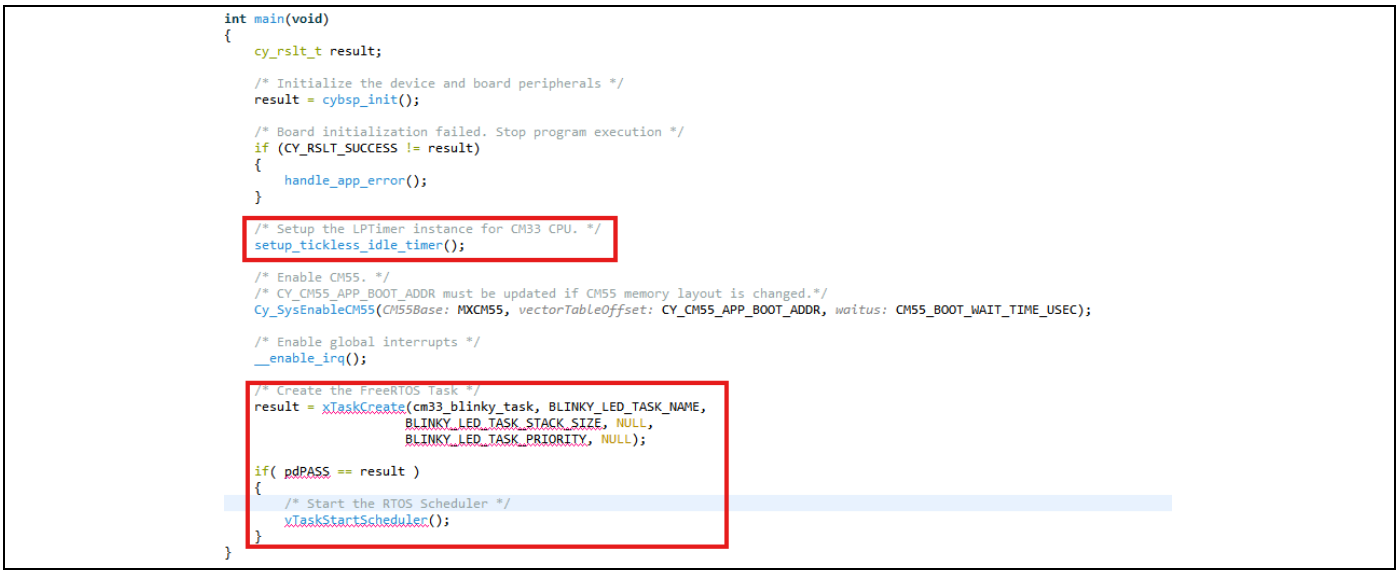



**Figure 162 Removing FreeRTOS related function calls in main function**

## Appendix

8. Add an infinite for loop to the main function

```
int main(void)
{
    cy_rslt_t result;

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


    Cy_SysEnableCM55(MXCM55, CM55_APP_BOOT_ADDR, CM55_BOOT_WAIT_TIME_USEC);

    /* Enable global interrupts */
    __enable_irq();

    for (;;)
    {


    }// for
}// main
```

**Figure 163 Adding infinite for loop to main function**

9. Navigate to the makefile for each project and remove the FREERTOS and RTOS_AWARE components

```
main.c  Makefile
# in use. In general, if you have
#
#    COMPONENTS=foo bar
#
# ... then code in directories named COMPONENT_foo and COMPONENT_bar will be
# added to the build
#
COMPONENTS+=FREERTOS RTOS_AWARE
```

**Figure 164 Removing the FREERTOS and RTOS_AWARE components from the Makefile**

10. Ensure that these steps are completed for the CM33_NS and CM55 projects
11. Build the project to ensure that all steps were completed succesfully

```
Console  Problems  Progress  Memory  Terminal
CDT Build Console [PSOC_Edge_Empty_Application.proj_cm55]
**********************************************************************
= Build complete =
**********************************************************************

Calculating memory consumption: PSE846GPS2DBZC4A GCC_ARM



13:56:39 Build Finished. 0 errors, 0 warnings. (took 23s.812ms)
```

**Figure 165 Passing build after disabling FreeRTOS from PSOC™ Edge Empty App**

## Appendix

### 9.3 PSOC™ Edge MTB-HAL for retarget-IO

The PSOC™ Edge support within ModusToolbox™ introduces a Device Support Library (DSL) concept. The DSL for PSOC™ Edge includes the stdlib-stubs, the MTB-HAL, the PDL, and the Syspm-Callbacks libraries. This means that disabling the MTB-HAL is not available for PSOC™ Edge. Please use the following source code to enable retarget-IO for the PSOC™ Edge.

```
#include "cy_retarget_io.h"

mtb_hal_uart_t DEBUG_UART_hal_obj;
cy_stc_scb_uart_context_t DEBUG_UART_context;


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

        Cy_SCB_UART_Init(CYBSP_UART_HW, &CYBSP_UART_config, NULL);

        /* Enable global interrupts */
        __enable_irq();

        Cy_SCB_UART_Enable(CYBSP_UART_HW);

        result = mtb_hal_uart_setup(&DEBUG_UART_hal_obj, &CYBSP_UART_hal_config, &DEBUG_UART_context, NULL);

        cy_retarget_io_init(&DEBUG_UART_hal_obj);

        printf("Hello from retarget-IO\r\n");
```

**Figure 166 Updated source code for retarget-IO use with MTB-HAL**

## Appendix

## 9.4 PSOC™ 6 system and peripheral clocks

The PSOC™ 6 default clock tree setup for BSPs sets the CLK_HF0 to 100 MHz using the FLL to generate the input. The CLK_HF0 is used to drive the CLK_PERI, which provides the source clock frequency for all peripheral clock dividers. The CLK_HF0 can also be derived from the PLL0 and PLL1, which can drive the input to CLK_HF0 at 150 MHz. This is important for PSOC™ peripherals like SPI that require a specific peripheral clock input to meet the required baud rate. When needing to achieve a baud rate of 1 MHz (1000 ksps) for a SPI bus, it is best to set the CLK_PERI to 75 MHz, which means that the CLK_HF0 needs to be set to 150 MHz. Follow these steps to adjust the clock system before enabling the SCB for SPI.

1. Open the device configurator and navigate to the **System** tab

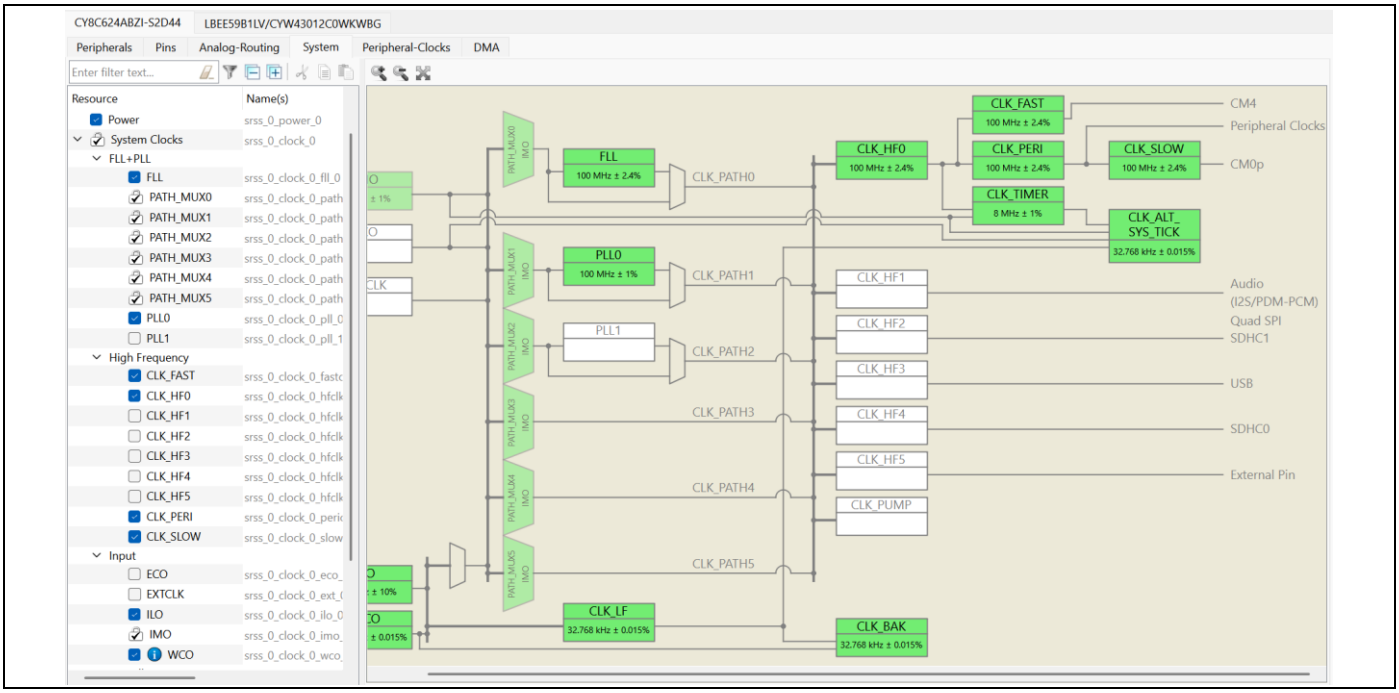



**Figure 167 PSOC™ 6 System tab view**

2. Click on the PLL0 block and adjust the frequency to 150 MHz

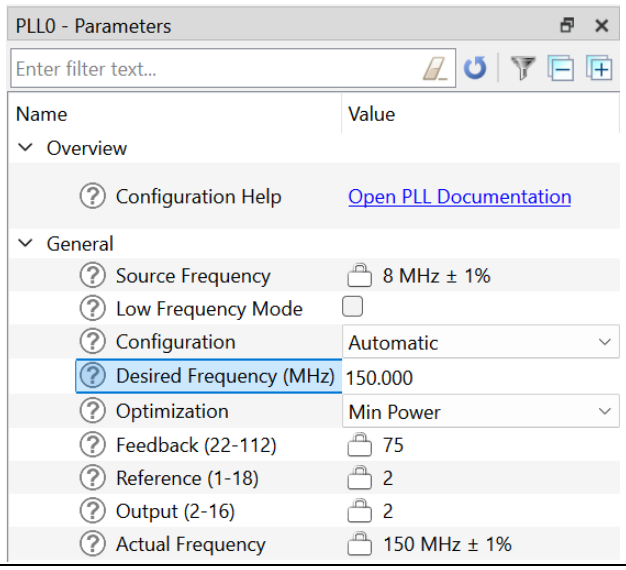



**Figure 168 PSOC™ 6 System tab view**

## Appendix

3. Click on the CLK_HF0 and change the clock path to CLK_PATH1 (PLL0)

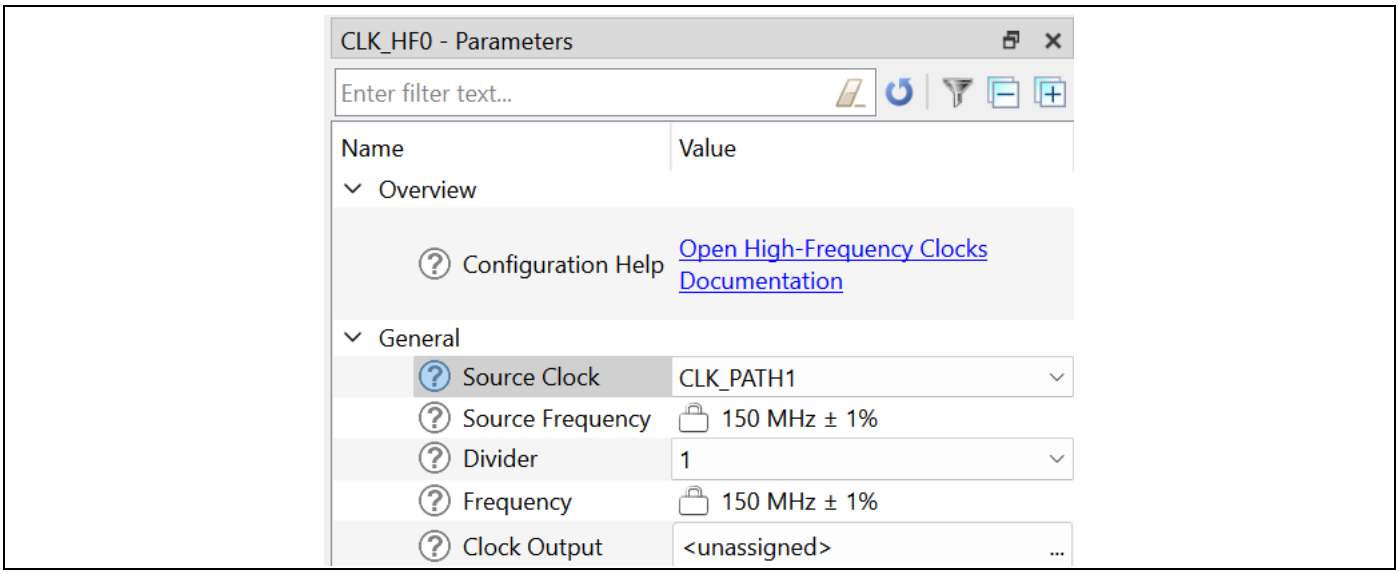



**Figure 169 PSOC™ 6 System tab view**

4. Click on the CLK_PERI and set the divider to 2

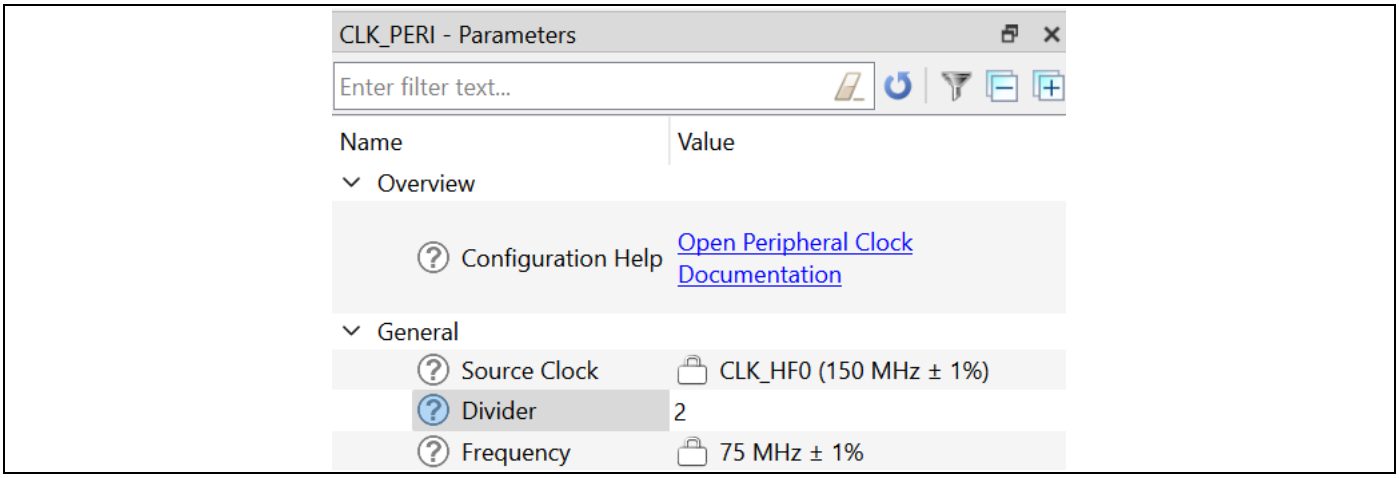



**Figure 170 PSOC™ 6 System tab view**

5. Now when enabling the SCB for SPI, and selecting an unused peripheral clock will cause the device configurator to modify the peripheral clock divider to meet the baud rate requirements automatically

## 9.5 Evaluation Kits

| Product Family | Evaluation Kit | Link |
|---|---|---|
| PSOC™ 4100T Plus | CY8CPROTO-041TP | https://www.infineon.com/evaluation-board/CY8CPROTO-041TP |
| PSOC™ 4000T | CY8CKIT-040T-MS | https://www.infineon.com/evaluation-board/CY8CPROTO-040T-MS |
| PSOC™ 4100S Max | CY8CKIT-041S-MAX | https://www.infineon.com/evaluation-board/CY8CKIT-041S-MAX |
| PSOC™ 6 | CY8CKIT-062S2-43012 | https://www.infineon.com/evaluation-board/CY8CKIT-062S2-43012 |
| PSOC™ Edge | KIT_PSE84_EVAL | https://www.infineon.com/evaluation-board/KIT-PSE84-EVAL |

## Appendix

# References

[1] Infineon Technologies AG: ModusToolbox™ Software Training Level 2 – PSOC™ MCUs; Available online

[2] Infineon Technologies AG: AN79953 Getting started with PSOC™ 4 MCU; Available online

[3] Infineon Technologies AG: AN228571 Getting started with PSOC™ 6 MCU on ModusToolbox™ software; Available online

[4] Infineon Technologies AG: AN235935 – Getting started with PSOC™ Edge E84 On ModusToolbox™; Available online

## Appendix

# Glossary

**RTC**

Real Time Clock

**PDL**

Peripheral Driver Library

**SCB**

Serial Communication Block

**DSL**

Device Support Library

**WDT**

Watch Dog Timer

## Appendix

# Revision history

| Document revision | Date | Description of changes |
|---|---|---|
| 1.00 | 2025-09-30 | Initial document |
| | | |
| | | |

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

**Edition 2025-09-30**
**Published by**

**Infineon Technologies AG**
**81726 Munich, Germany**



**Do you have a question about this document?**
**Email:**
erratum@infineon.com
**Document reference**
**User guide number**

**Important notice**
The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”).

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

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

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

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

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