Chapter 2: Connect to MCU Peripherals

Time 2 Hours

At the end of this chapter you will have ModusToolbox installed and working on your computer, and you will understand how to program an existing project into a kit. You should be able to write firmware for the MCU peripherals (GPIOs, PWMs, UART, and I2C) and to interface with the shield including the ambient light sensor, motion sensor, and TFT display. In addition, you will understand the role of the critical files related to the kit hardware abstraction layer (HAL).

Table of Contents

Document conventions

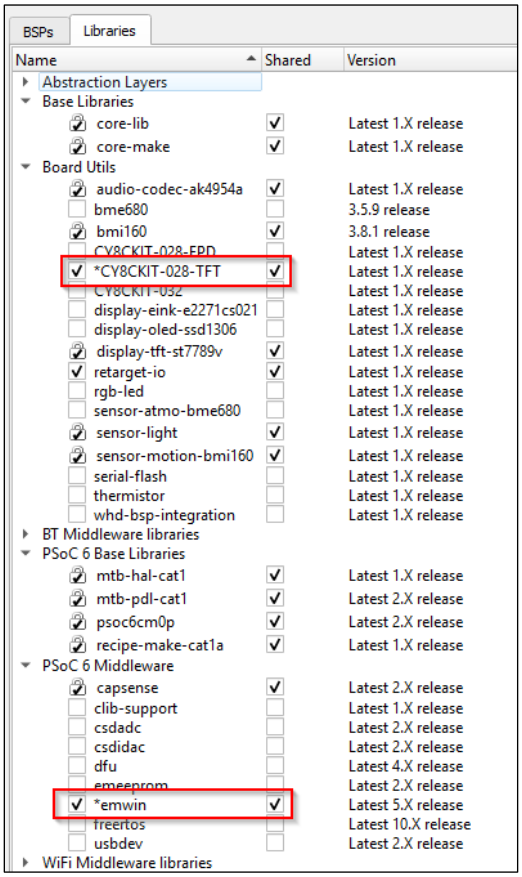
Convention	Usage	Example
`Courier New`	Displays code	`CY_ISR_PROTO(MyISR);`
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold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

2.1 Shield Board Support Libraries

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

After creating a new project, open the Library Manager by selecting the project in the Eclipse IDE Project Explorer, and then clicking the Library Manager link in the Quick Panel.

When the Library Manager opens, add the CY8CKIT-028-TFT and emWin libraries.



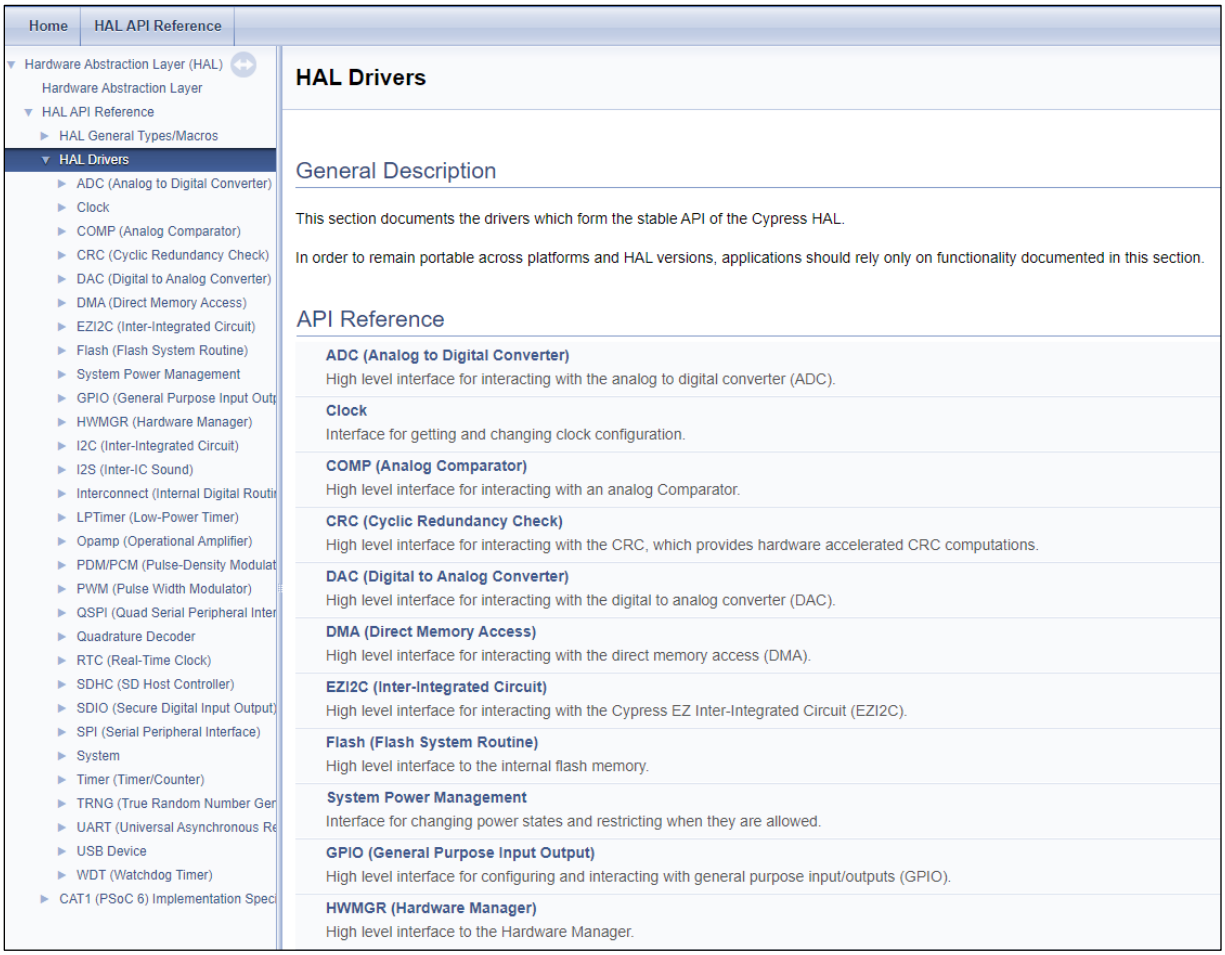
The CY8CKIT-028-TFT library provides support for initializing/freeing all the hardware peripheral resources on the shield, defining all pin mappings from the baseboard's Arduino interface to the shield peripherals, and providing access to each of the underlying peripherals on the shield. This library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

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

```
COMPONENTS+=EMWIN_NOSNTS
```

2.2 Documentation

Documentation can be found in the Eclipse IDE Quick Panel Documentation tab. The Hardware Abstraction Layer (HAL) link contains the documentation of the APIs that we will be using in this chapter. Open the documentation by clicking on it. Then expand "HAL API Reference" and "HAL Drivers" to see the list of supported components. We will be using GPIO, PWM, UART, and I2C.



Click on GPIO to see the list of GPIO APIs and then click on the `cyhal_gpio_init` function for a description.

You can further click on types for the arguments to see what choices are available. For example, clicking on `cyhal_gpio_drive_mode_t` will show you this list:

You can also find that information from inside the Eclipse IDE. You can highlight the parameter in the C code, right click, and select **Open Declaration** (you will try this later in the exercises). If you don't already have a valid parameter provided, you can also get there by using **Open Declaration** on the function name, then the parameter type, and then the type name. This will show you the datatype with an explanation of the allowed choices:

```
/** Pin drive mode */

/** \note When the <b> drive_mode </b> of the <b> pin </b> is set to <b> CYHAL_GPIO_DRIVE_PULL_NONE </b>,
 * it is set to <b> CYHAL_GPIO_DRIVE_STRONG </b> if the <b> direction </b>
 * of the <b> pin </b> is <b> CYHAL_GPIO_DIR_OUTPUT </b> or <b> CYHAL_GPIO_DIR_BIDIRECTIONAL</b>.
 * If not, the <b> drive_mode </b> of the <b> pin </b> is set to <b> CYHAL_GPIO_DRIVE_NONE</b>.
 */
typedef enum {
    CYHAL_GPIO_DRIVE_NONE,              /**< Digital Hi-Z. Input only. Input init value(s): 0 or 1 */
    CYHAL_GPIO_DRIVE_ANALOG,            /**< Analog Hi-Z. Use only for analog purpose */
    CYHAL_GPIO_DRIVE_PULLUP,            /**< Pull-up resistor. Input and output. Input init value(s): 1, output value(s): 0 */
    CYHAL_GPIO_DRIVE_PULLDOWN,          /**< Pull-down resistor. Input and output. Input init value(s): 0, output value(s): 1 */
    CYHAL_GPIO_DRIVE_OPENDRAINDRIVESLOW,  /**< Open-drain, Drives Low. Input and output. Input init value(s): 1, output value(s): 0 */
    CYHAL_GPIO_DRIVE_OPENDRAINDRIVESHIGH, /**< Open-drain, Drives High. Input and output. Input init value(s): 0, output value(s): 1 */
    CYHAL_GPIO_DRIVE_STRONG,            /**< Strong output. Output only. Output init value(s): 0 or 1 */
    CYHAL_GPIO_DRIVE_PULLUPDOWN,        /**< Pull-up and pull-down resistors. Input and output. Input init value(s): 0 or 1, output value(s): 0 or 1 */
    CYHAL_GPIO_DRIVE_PULL_NONE,         /**< No Pull-up or pull-down resistors. Input and output. Input init value(s): 0 or 1, output value(s): 0 or 1 */
} cyhal_gpio_drive_mode_t;
```

2.3 Creating a new ModusToolbox project

For a step by step guide for creating new ModusToolbox projects you can refer to the Eclipse IDE for ModusToolbox Quick Start Guide, or to section 1.8.1 of this training.

2.3.1 Programming your kit

The programmer firmware on the PSoC 6 development kits is called KitProg3. This firmware talks to your computer via USB and to the PSoC 6 target via a protocol called Serial Wire Debug (SWD). The host application on your computer needs to talk to the programmer to debug the PSoC 6 and to download firmware into the PSoC 6 flash. There are a bunch of different protocols out there for accomplishing this task. However, a few years ago Arm developed a standard called CMSIS-DAP, which has two variants that are implemented in the KitProg firmware (Bulk and DAPLink).

Note: Older versions of KitProg firmware also support HID mode, which we typically don't use anymore.

In order to program the PSoC, KitProg needs to be in the right mode – meaning the mode that has the functionality that works with your environment. You can switch modes by pressing the mode button on the development kit. Each PSoC 6 development kit has an LED that will be solid or ramping on/off at a 2 Hz rate to indicate the mode as shown below.

CMSIS Mode	Application	Mass Storage	Bridges	Solution	Description	LED
BULK	Eclipse IDE or ModusToolbox Command Line	No	UART I2C	PSoC 6 & AFR	The latest version of the protocol which uses USB bulk mode – by far the fastest.	Solid
DAPLink	Mbed Studio or Mbed CLI	Yes	UART	Mbed OS	A modified version of CMSIS-DAP that enables web debugging	2 Hz Ramping

To download the project to your board, you only need to do two things:

1. Verify that your kit is in CMSIS-DAP BULK mode (described above)
2. Click the **<project_name> Program (KitProg3_MiniProg4)** link in the Quick Panel

2.4 Peripherals

2.4.1 GPIO

As explained previously, GPIOs must be initialized before they are used. To initialize a GPIO, call `cyhal_gpio_init`. The following is from the file *cyhal_gpio.h*:

```
/** Initialize the GPIO pin <br>
 * See \ref subsection_gpio_snippet_1.
 *
 * @param[in] pin         The GPIO pin to initialize
 * @param[in] direction   The pin direction
 * @param[in] drive_mode  The pin drive mode
 * @param[in] init_val    Initial value on the pin
 *
 * @return The status of the init request
 *
 * Guidance for using gpio drive modes ( \ref cyhal_gpio_drive_mode_t for details).
 * For default use drive modes:
 * Input GPIO direction - \ref CYHAL_GPIO_DRIVE_NONE
 * Output GPIO direction - \ref CYHAL_GPIO_DRIVE_STRONG
 * Bidirectional GPIO - \ref CYHAL_GPIO_DRIVE_PULLUPDOWN
 * \warning Don't use \ref CYHAL_GPIO_DRIVE_STRONG for input GPIO direction. It may cause an overcurrent issue.
 */
cy_rslt_t cyhal_gpio_init(cyhal_gpio_t pin, cyhal_gpio_direction_t direction, cyhal_gpio_drive_mode_t drive_mode, bool init_val);
```

Once initialized, input pins can be read using `cyhal_gpio_read` and outputs can be driven using `cyhal_gpio_write`. The first parameter for these functions is the pin name such as `P1_5` or a BSP defined name such as `CYBSP_USER_LED`. The write function call also takes a value to write such as `0U`, or the BSP defined `CYBSP_LED_STATE_ON`.

GPIO interrupts are controlled using `__enable_irq` and `__disable_irq`.

The function `cyhal_gpio_free` is available to un-initialize a pin. This is necessary if you have a pin that was previously initialized as a GPIO and you want to use a peripheral such as PWM to drive.

2.4.2 PWM

The PWM has API functions to initialize a PWM resource and assign the output to a specified pin, as well as set the frequency (in Hz) and the duty cycle (in percent). These functions are used for initialization and to change the frequency or duty cycle once the PWM is running. After initializing the PWM, you must call the start function for PWM to generate an output. The PWM also has a function to stop the output. See the API documentation for details.

2.4.3 UART

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

The UART initialization function requires a configuration structure of type `cyhal_uart_cfg_t` with the following elements. `cyhal_uart_cfg_t` is defined in *cyhal_uart.h*. As mentioned above, you can find this structure by highlighting, right clicking, and selecting **Open Declaration** from inside Eclipse IDE for ModusToolbox on the function name, parameter type, and type name.

```
typedef struct
{
    uint32_t data_bits; //!< The number of data bits (generally 8 or 9)
    uint32_t stop_bits; //!< The number of stop bits (generally 0 or 1)
    cyhal_uart_parity_t parity; //!< The parity
    uint8_t *rx_buffer; //!< The rx software buffer pointer, if NULL, no rx software buffer will be used
    uint32_t rx_buffer_size; //!< The number of bytes in the rx software buffer
} cyhal_uart_cfg_t;
```

You can also use **Open Declaration** on each of the types inside the structure to find valid choices. For example, for the parity, the possible choices are:

```
/** UART Parity */
typedef enum
{
    CYHAL_UART_PARITY_NONE,   /**< UART has no parity check   */
    CYHAL_UART_PARITY_EVEN,   /**< UART has even parity check */
    CYHAL_UART_PARITY_ODD,    /**< UART has odd parity check  */
} cyhal_uart_parity_t;
```

2.4.4 Retargeting printf/scanf and Debug Printing

A utility library called *retarget-io* is available to simplify the process of setting up the UART to use with the C-standard `printf` and `scanf` functions. This is especially useful for using `printf` to print debug messages.

To use it, you must first include the retarget-io library in your application. This can be done easily by using the library manager. Once that is done, include the header file *cy_retarget_io.h* in your code to get access to the functions.

As with all libraries, the API is documentation and usage examples can be found in the *docs* directory of the library or from a link in the Quick Panel.

To initialize the debug UART, you can call:

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

As you can see, the pin definitions are taken from the BSP while the default baud rate is defined by the library (it is 115200).

Once that's done, you can use `printf` and `scanf` as normal.

If desired, the library can be configured to automatically convert `\n` characters to `\r\n`. Do to that, just define the macro `CY_RETARGET_IO_CONVERT_LF_TO_CRLF`.

The *regarget-io* library provides access to the UART HAL object so that you can use standard UART commands on a UART that was initialized using `cy_retarget_io_init`. You will find that object defined in the *cy_retarget_io.h* file.

2.4.5 I2C

The device contains multiple serial communication blocks that can implement both I2C masters and slaves.

As with other peripherals, you need to initialize the block using the initialization function. The initialization function `cyhal_i2c_init` among other things takes a pointer to a `cyhal_i2c_t` object, which has the following definition.

```
typedef struct {
#ifdef CY_IP_MXSCB
    CySCB_Type*                             base;
    cyhal_resource_inst_t                   resource;
    cyhal_gpio_t                            pin_sda;
    cyhal_gpio_t                            pin_scl;
    cyhal_clock_t                           clock;
    bool                                    is_shared_clock;
    cy_stc_scb_i2c_context_t                context;
    cy_stc_scb_i2c_master_xfer_config_t     rx_config;
    cy_stc_scb_i2c_master_xfer_config_t     tx_config;
    bool                                    is_slave;
    uint32_t                                address;
    uint32_t                                irq_cause;
    uint16_t                                pending;
    uint16_t                                events;
    cyhal_event_callback_data_t             callback_data;
#else
    void *empty;
#endif
} cyhal_i2c_t;
```

After an I2C is initialized it may be configured via the `cyhal_i2c_configure` function. This function takes a pointer to a `cyhal_i2c_t`, and a pointer to an object of type `cyhal_i2c_cfg_t` which has the following definition.

```
typedef struct
{
    bool is_slave;            /**<  Operates as a slave when set to (<b>true</b>), else as a master (<b>false</b>) */
    uint16_t address;         /**<  Address of this slave resource (7-bit), should be set to 0 for master */
    uint32_t frequencyhal_hz; /**<  Frequency that the I2C bus runs at (I2C data rate in bits per second) <br>
                                    Refer to the device datasheet for the supported I2C data rates */
} cyhal_i2c_cfg_t;
```

There are three ways for a master to read/write data from/to the slave:

1. There is a dedicated read function called `cyhal_i2c_master_read` and a dedicated write function called `cyhal_i2c_master_write`.
2. There is a dedicated read function `cyhal_i2c_master_mem_read` and a dedicated write function `cyhal_i2c_master_mem_write` that perform I2C reads and writes using a block of data at a specified memory address.
3. There is a function called `cyhal_i2c_master_transfer_async` which can do a read, a write, or both. We will focus on the separate functions here, but feel free to look at the transfer function in the documentation and experiment with it. Some kits may not support all these methods.

2.4.5.1 cyhal_i2c_master_read and cyhal_i2c_master_write

The read/write functions take a pointer to the master I2C object, device address of the I2C slave, the data to send/receive, the size of that data, a timeout value in milliseconds, and a boolean “send_stop” parameter that determines whether or not the stop signal should be sent. All of the following screenshots are from the file *cyhal_i2c.h*:

```
/**
 * I2C master blocking write
 *
 * This will write `size` bytes of data from the buffer pointed to by `data`. It will not return
 * until either all of the data has been written, or the timeout has elapsed.
 *
 * @param[in]  obj        The I2C object
 * @param[in]  dev_addr   device address (7-bit)
 * @param[in]  data       I2C send data
 * @param[in]  size       I2C send data size
 * @param[in]  timeout    timeout in millisecond, set this value to 0 if you want to wait forever
 * @param[in]  send_stop  whether the stop should be send, used to support repeat start conditions
 *
 * @return The status of the master_write request
 */
cy_rslt_t cyhal_i2c_master_write(cyhal_i2c_t *obj, uint16_t dev_addr, const uint8_t *data, uint16_t size, uint32_t timeout, bool send_stop);

/**
 * I2C master blocking read
 *
 * This will read `size` bytes of data into the buffer pointed to by `data`. It will not return
 * until either all of the data has been read, or the timeout has elapsed.
 *
 * @param[in]   obj        The I2C object
 * @param[in]   dev_addr   device address (7-bit)
 * @param[out]  data       I2C receive data
 * @param[in]   size       I2C receive data size
 * @param[in]   timeout    timeout in millisecond, set this value to 0 if you want to wait forever
 * @param[in]   send_stop  whether the stop should be send, used to support repeat start conditions
 *
 * @return The status of the master_read request
 */
cy_rslt_t cyhal_i2c_master_read(cyhal_i2c_t *obj, uint16_t dev_addr, uint8_t *data, uint16_t size, uint32_t timeout, bool send_stop);
```

2.4.5.2 cyhal_i2c_master_mem_read and cyhal_i2c_master_mem_write

These read and write functions both take a pointer to the master I2C object, the device address of the I2C slave, a memory address at which to store the written data, the size of that memory, a pointer to the master send data (this variable is written to in the read function), and the master send data size.

```
/** Perform an I2C write using a block of data stored at the specified memory location
 *
 * @param[in]  obj           The I2C object
 * @param[in]  address       device address (7-bit)
 * @param[in]  mem_addr      mem address to store the written data
 * @param[in]  mem_addr_size  number of bytes in the mem address
 * @param[in]  data          I2C master send data
 * @param[in]  size          I2C master send data size
 * @param[in]  timeout       timeout in millisecond, set this value to 0 if you want to wait forever
 * @return The status of the write request
 */

cy_rslt_t cyhal_i2c_master_mem_write(cyhal_i2c_t *obj, uint16_t address, uint16_t mem_addr, uint16_t mem_addr_size, const uint8_t *data, uint16_t size, uint32_t timeout);

/** Perform an I2C read using a block of data stored at the specified memory location
 *
 * @param[in]  obj           The I2C object
 * @param[in]  address       device address (7-bit)
 * @param[in]  mem_addr      mem address to store the written data
 * @param[in]  mem_addr_size  number of bytes in the mem address
 * @param[out] data          I2C master send data
 * @param[in]  size          I2C master send data size
 * @param[in]  timeout       timeout in millisecond, set this value to 0 if you want to wait forever
 * @return The status of the read request
 */
cy_rslt_t cyhal_i2c_master_mem_read(cyhal_i2c_t *obj, uint16_t address, uint16_t mem_addr, uint16_t mem_addr_size, uint8_t *data, uint16_t size, uint32_t timeout);
```

2.4.5.3 cyhal_i2c_master_transfer_async

The transfer function takes a pointer to the master I2C object, the device address of the I2C slave, a pointer to the transmit buffer, the number of bytes to transmit, a pointer to the receive buffer, and the number of bytes to receive.

```
/** Initiate a non-blocking I2C master asynchronous transfer. Supports simultaneous write and read operation.<br>
 *
 * This will transfer `rx_size` bytes into the buffer pointed to by `rx`, while simultaneously transfering
 * `tx_size` bytes of data from the buffer pointed to by `tx`, both in the background.
 * When the requested quantity of data has been received, the @ref CYHAL_I2C_MASTER_RD_CMPLT_EVENT will be raised.
 * When the requested quantity of data has been transmitted, the @ref CYHAL_I2C_MASTER_WR_CMPLT_EVENT will be raised.
 * See @ref cyhal_i2c_register_callback and @ref cyhal_i2c_enable_event.
 * If either of <b>tx_size</b> or <b>rx_size</b> is '0', the respective write or read operation is not performed.
 * See \ref subsection_i2c_snippet_3
 *
 * @param[in]  obj      The I2C object
 * @param[in]  address  device address (7-bit)
 * @param[in]  tx       The transmit buffer
 * @param[in]  tx_size  The number of bytes to transmit. Use '0' if write operation is not required.
 * @param[out] rx       The receive buffer
 * @param[in]  rx_size  The number of bytes to receive. Use '0' if read operation is not required.
 * @return The status of the master_transfer_async request
 *
 */
cy_rslt_t cyhal_i2c_master_transfer_async(cyhal_i2c_t *obj, uint16_t address, const void *tx, size_t tx_size, void *rx, size_t rx_size);
```

2.4.5.4 cyhal_i2c_slave_config_read_buffer and cyhal_i2c_slave_config_write_buffer

These functions are for configuring the buffers on slaves that the masters read data from and write data to. As arguments they take a pointer to the slave I2C object, a pointer to a data buffer, and the size of that data buffer.

```
/**
 * The function configures the write buffer on an I2C Slave. This is the buffer to which the master writes data to.
 * The user needs to setup a new buffer every time (i.e. call \ref cyhal_i2c_slave_config_write_buffer and \ref cyhal_i2c_slave_config_read_buffer
 * every time the buffer has been used up)<br>
 * See related code example: <a href="https://github.com/cypresssemiconductorco/mtb-example-psoc6-i2c-master" ><b>PSoC 6 MCU: I2C Master</b></a>
 *
 * @param[in]  obj      The I2C object
 * @param[in]  data     I2C slave send data
 * @param[in]  size     I2C slave send data size
 *
 * @return The status of the slave_config_write_buffer request
 */
cy_rslt_t cyhal_i2c_slave_config_write_buffer(cyhal_i2c_t *obj, const uint8_t *data, uint16_t size);

/**
 * The function configures the read buffer on an I2C Slave. This is the buffer from which the master reads data from.
 * The user needs to setup a new buffer every time (i.e. call \ref cyhal_i2c_slave_config_write_buffer and \ref cyhal_i2c_slave_config_read_buffer
 * every time the buffer has been used up)<br>
 * See related code example: <a href="https://github.com/cypresssemiconductorco/mtb-example-psoc6-i2c-master" ><b>PSoC 6 MCU: I2C Master</b></a>
 *
 * @param[in]   obj      The I2C object
 * @param[out]  data     I2C slave receive data
 * @param[in]   size     I2C slave receive data size
 *
 * @return The status of the slave_config_read_buffer request
 */
cy_rslt_t cyhal_i2c_slave_config_read_buffer(cyhal_i2c_t *obj, uint8_t *data, uint16_t size);
```

2.4.6 EZI2C

The I2C HAL provides standard commands for slave operation but in many instances, it is better to use the EZI2C HAL API on the slave side.

EZI2C adds a protocol on top of I2C which allows the 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.

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

2.4.7 ADC

In order to utilize an ADC, you must first initialize an ADC block by calling `cyhal_adc_init`. This function takes three arguments:

- `cyhal_adc_t*` – This must be previously declared by the caller but the `init` function will initialize its contents.
- `cyhal_gpio_t` – A pin corresponding to the ADC block to initialize. Note: this pin is not reserved, it is just used to identify which ADC block to allocate. The pin or pins are reserved later during channel configuration.
- `const cyhal_clock_t*` – The clock to use can be shared. If this argument is `NULL`, a new one will be allocated.

You must then initialize an ADC channel. To do this you need to first declare an object of type `const cyhal_adc_channel_config_t` as follows:

```
const cyhal_adc_channel_config_t channel_config =
{
    .enable_averaging = false,
    .min_acquisition_ns = 220,
    .enabled = true
};
```

Depending on your use case you may need different values for these options. For more information on them, or about ADC support in general, you can refer to the HAL documentation.

You then need to call `cyhal_adc_channel_init_diff`. This function takes the following arguments:

- `cyhal_adc_channel_t*` – The ADC channel object to initialize. You should have previously declared this
- `cyhal_adc_t*` – The ADC object used in `cyhal_adc_init`

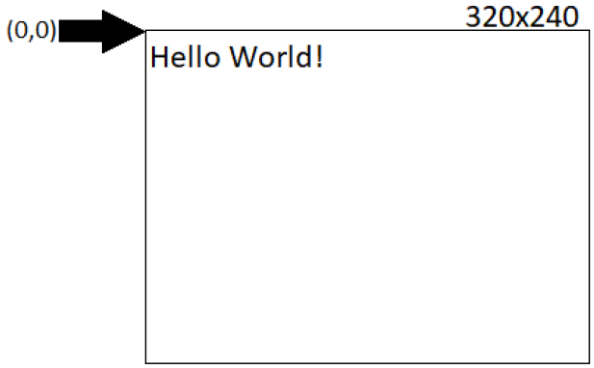
- `cyhal_gpio_t` – Non-inverting input pin
- `cyhal_gpio_t` – Inverting input pin. For a single ended channel, use `CYHAL_ADC_VNEG`
- `const cyhal_adc_channel_config_t*` – a pointer to the ADC config object you previously declared

You can then read from the ADC simply by calling `cyhal_adc_read` with the only argument being a pointer to the ADC channel object you previously declared and initialized.

2.4.8 TFT Display

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

The *emWin* library allows you to draw shapes and write text on the TFT screen. When you use a function such as `GUI_DispString("Hello World!")` the x and y coordinates start in the top left corner (0, 0), like this.



The steps to draw text to the TFT screen are:

1. Add the *emwim* and *CY8CKIT-028-TFT* libraries to your project via the library manager.

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

Note: *Documentation for these libraries can be found in the mtb_shared/latest-vX.X/<lib>/docs directories.*

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

   ```
   COMPONENTS+=EMWIN_NOSNTS
   ```

3. Include *cy8ckit_028_tft.h* and call `cy8ckit_028_tft_init` to initialize the shield.
4. Call `GUI_Init` with no arguments to initialize emWin internal data structures and variables.
5. Each time you want to display a string you call `GUI_DispString`. This function takes the string you want to display as its only argument.

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

   ```
   #include "cy_pdl.h"
   #include "cyhal.h"
   #include "cybsp.h"
   #include "GUI.h"
   #include "mtb_st7789v.h"
   #include "cy8ckit_028_tft.h"

   int main(void){
   ```

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

    __enable_irq();

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

    GUI_Init();
    GUI_DispString("PSoC");

    for(;;){
    }
}
```

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

a. Include *cy8ckit_028_tft_pins.h.*
b. Setup a structure of type `mtb_st7789v_pins_t` to specify the shield's pins.

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

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

2.5 Exercises

2.5.1 Exercise 1: Install Shield Support Libraries and use the TFT Display

1. Use what you learned in the fundamentals to install the libraries for the appropriate kit/shield combination into your SDK Workspace.

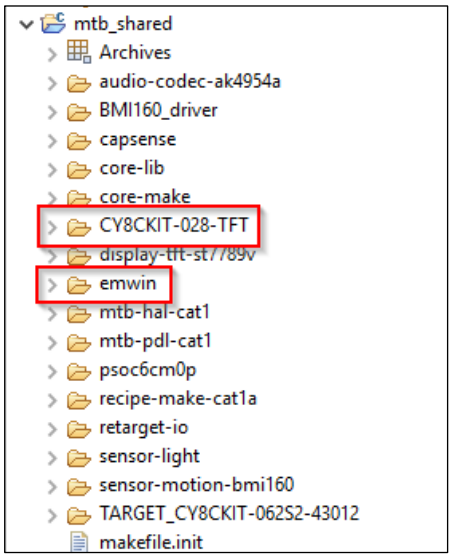
   a. Launch the Project Creator tool from the Eclipse IDE, select your kit name, and use the Empty PSoC 6 example application. Name your application **ch02_ex01_tft**.

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

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

2. Once you have installed the libraries, click on the *mtb_shared* directory from inside the Eclipse IDE Project Explorer.

   You should see the libraries that you just installed. If you do not see them, ask for help - don't go forward until the libraries are properly installed.



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

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

int main(void)
{
    cy_rslt_t result;

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

    __enable_irq();

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

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

    for(;;)
    {
    }
}
```

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

2.5.2 Exercise 2: (GPIO) Blink an LED

1. Create a new application called **ch02_ex02_blinkled** using Empty PSoC6 App as the template.
2. Add code to *main.c* before the infinite loop to initialize `CYBSP_USER_LED` as a strong drive digital output.

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

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

   Hint See the HAL API documentation for the GPIO functions to drive the LED high and low.

   Hint Use the `cyhal_system_delay_ms` function for the delay.

4. Program your project to the board by clicking **ch02_ex02_blinkled Program (KitProg3_MiniProg4)** in the Quick Panel.

Questions to Answer

1. Why can't you read the value of the LED using the `cyhal_gpio_read` function instead of using a variable to remember the state?

2. In what file and on what line does the `CYBSP_USER_LED` get assigned to the correct pin for this kit?

2.5.3 Exercise 3: (GPIO) Add Debug Printing to the LED Blink Project

1. Use the project creator to create a new application called **ch03_ex03_blinkled_print** using the **Import** button on the “Select Application” page to select your previous exercise (ch02_ex02_blinkled) as a template.
2. Include the *retarget-io* library using the ModusToolbox Library Manager.
3. In *main.c*, before the infinite loop, call the following function to initialize retarget-io to use the debug UART port:

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

   Hint Remember to #include "cy_retarget_io.h".

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

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

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

   Hint if you don’t have terminal emulator software installed, you can use putty.exe which is included in the class files under *Software_tools*. To configure putty:

   - Go to the Serial tab, select the correct COM port (you can get this from the device manager under **Ports (COM & LPT)** as "KitProg3 USB-UART"), and set the speed to 115200.
   - Go to the session tab, select the **Serial** button, and click on **Open**.

   Hint For MacOS, you may want to use the `screen` command as a terminal window.

   - Look for a USB serial device in */dev/tty.**
   - Use the command:

     ```
     screen /dev/tty.<your_device> 115200
     ```

2.5.4 Exercise 4 (GPIO) Read the State of a Mechanical Button

1. Create a new project called **ch02_ex04_button** using the Empty PSoC 6 application as the template.
2. In the C file, initialize the pin for the button (`CYBSP_USER_BTN`) as an input with a resistive pullup and initialize the LED (`CYBSP_USER_LED`) as a strong drive output.

   Hint The button pulls the pin to ground when pressed. An input with a resistive pullup is required so that the pin is pulled high when the button is not being pressed.
3. In the infinite loop, check the state of the button. Turn the LED ON if the button is pressed and turn it OFF if the button is not pressed.
4. Program your project to the board and press the button to observe the behavior.

2.5.5 Exercise 5: (GPIO) Use an Interrupt to Toggle the State of an LED

1. Create a new project called **ch02_ex05_interrupt** using the **Import** button on the "Select Application" page to select your previous exercise (ch02_ex04_blinkled) as a template.
2. In the *main.c* file, set up a falling edge interrupt for the GPIO connected to the button.

   Hint See the documentation for `cyhal_gpio_enable_event`.

   Hint Build the project so that Intellisense will work properly. Then in your C code:

   - Type `cyhal_gpio_enable_event`.
   - Highlight `cyhal_gpio_enable_event`, right click on it, and select **Open Declaration**. This will show the required parameters for the function.
   - Highlight `cyhal_gpio_event_t`, right click on it, and select **Open Declaration**.
   - Identify the correct value to use for a falling edge interrupt.
3. Register a callback function to the button by calling `cyhal_gpio_register_callback`.

   Hint Look at the function documentation to find out what arguments it takes.

   Hint Use `NULL` for `callback_arg`
4. Create the interrupt service routine (ISR) so that it toggles the state of the LED each time the button is pressed.

   Your ISR should look something like this:

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

   Hint You can use the function `cyhal_gpio_toggle`.
5. Program your project to the board and press the button to observe the behavior.

2.5.6 Exercise 6: (I2C READ) Read PSoC Sensor Values over I2C

1. Create a new project called **ch02_ex06_i2cread** using the Empty PSoC 6 template.
2. Use the library manager to add the *CY8CKIT-028-TFT* library (to get access to the motion sensor) and the *retarget-io* library to allow `printf`.
3. Initialize the *retarget-io* library *main.c* and include the header file.
4. Add code so that every 100ms the motion sensor's acceleration and gyroscope data are read from the I2C slave.

   Hint Look at the documentation in the BMI160 Motion Sensor library to see an example of how to read the data. Don't forget to include the header file.

   Note Aliases for your kit's I2C SCL and SDA pins are defined in the BSP – you can use the device configurator to find them by entering "I2C" in the search box on the Pins tab.

5. Print the acceleration and gyroscope values to the terminal using `printf`.
6. Make sure the TFT shield is plugged in (that's where the motion sensor is), program the board and observe the results on the UART when you move and turn the kit.

2.5.7 Exercise 7: (Advanced) (ADC READ) Read Potentiometer Sensor Value via an ADC

1. Create a new project called **ch02_ex07_adcread** using the Empty PSoC 6 template.
2. Add the *retarget-io* library and initialize it in *main.c*. Don't forget to include the header file.
3. Update the code so that every 100 ms the potentiometer's voltage is read using an ADC. Print the values using `printf`.

   Hint You can find POT pin number by looking at the sticker on the back of the kit.

   Hint You can use the function `cyhal_adc_read_uv` to get a result in microvolts.

4. Program the project to the board and observe the range of values for the potentiometer.

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

2.5.8 Exercise 8: (Advanced) (PWM) LED brightness

1. Create a new project called **ch02_ex08_pwm** using the Empty PSoC 6 template.
2. In the C file, use a PWM to drive `CYBSP_USER_LED` instead of using the GPIO functions.
3. Configure the PWM and change the duty cycle in the main loop so that the LED gradually changes intensity.

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

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

2.5.9 Exercise 9: (Advanced) Display sensor information on the TFT display

Read the sensor data from the CY8CKIT-028-TFT light sensor and motion sensor and then use the emWim library to write the information to the display.

1. Create a new project called **ch02_ex09_sensorData** using the **Import** button to select your ch02_ex01_tft exercise as a template.
2. Add the *CY8CKIT-028-TFT, emWin* and *retarget-io* libraries.
3. Add the required includes for the new libraries.
4. Update the code so that the ambient light, acceleration, and gyro values are read from the shield and displayed to the screen every ½ second.

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

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

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

5. Program the project to the board.
6. Shine light on the light sensor and move the kit to see the values update.

2.5.10 Exercise 10: (Advanced) (UART) Write a value using the standard UART functions

1. Create a new project called **ch02_ex10_uartsend** using the **Import** button to select your ch02_ex05_interrupt exercise as a template.
2. Modify the C file so that the number of times the button has been pressed is sent out over the UART interface whenever the button is pressed.

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

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

   Hint The function to send a single character over UART is `cyhal_uart_putc`
3. Program your project to the board.
4. Open a terminal window. Press the button and observe the value displayed in the terminal.

   Hint If you are using `printf` rather than `cyhal_uart_putc`, you will need to also send an '\n' character as well.

2.5.11 Exercise 11: (Advanced) (UART) Read a value using the standard UART functions

1. Create a new project called **ch02_ex11_uartreceive** using the **Import** button to select your previous exercise (ch02_ex10_uartsend) as a template.
2. Update the code so that it looks for characters from the UART.

   If it receives a 1, turn on an LED. If it receives a 0, turn off an LED. Ignore any other characters.

   Hint Remove the code for the button press and its interrupt.

   Hint The function to receive a single character over UART is `cyhal_uart_getc`
3. Program your project to the board.
4. Open a terminal window and press the 1 and 0 keys on the keyboard and observe the LED turn on/off.