

# Chapter 2: Using the BT\_20819A1 SDK to Connect Inputs and Outputs to MCU Peripherals

Time: 3 Hours

At the end of this chapter you should be able to write firmware for the MCU peripherals (GPIOs, UARTs, Timers, PWMs, NVRAM, I2C, ADC and RTC). In addition, you will understand the role of the critical files related to the kit hardware platform and you will know how to re-map pin functions to different peripherals.

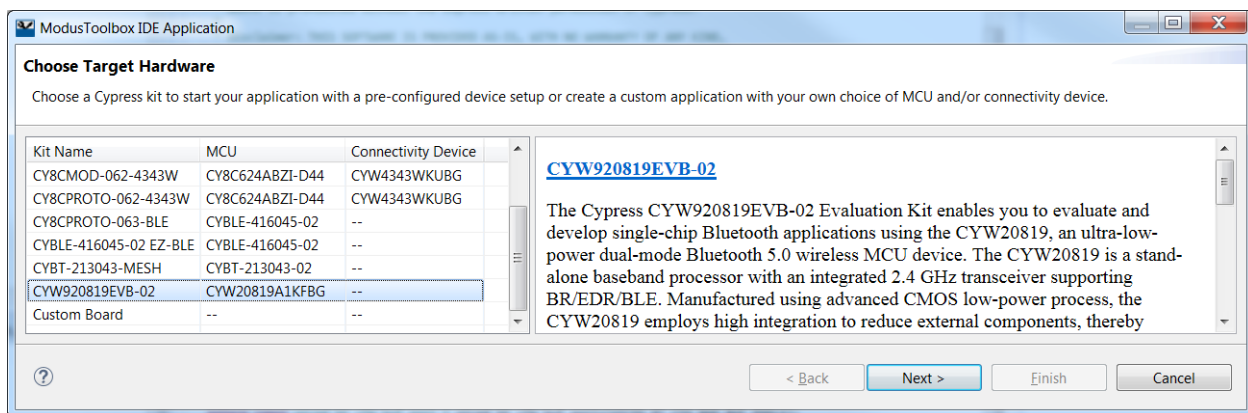
<b>2.1</b>	<b>CREATING A NEW MODUSTOOLBOX IDE PROJECT</b>	<b>2</b>
2.1.1	NEW APPLICATION WIZARD	2
2.1.2	DIRECTORY STRUCTURE	3
2.1.3	CONFIGURATION DATABASE - DESIGN.MODUS FILE	4
2.1.4	GENERATEDSOURCE FOLDER	4
2.1.5	PLATFORM FILES (WICED_PLATFORM.*)	5
2.1.6	STACK CONFIGURATION FILES	5
2.1.7	APPLICATION C FILE	5
2.1.8	APPLICATION SETTINGS	7
<b>2.2</b>	<b>CODE EXAMPLES</b>	<b>8</b>
2.2.1	BUILT-IN CODE EXAMPLES	8
2.2.2	GitHub CODE EXAMPLES	8
<b>2.3</b>	<b>BOARD SUPPORT PACKAGE</b>	<b>26</b>
<b>2.4</b>	<b>DOCUMENTATION</b>	<b>27</b>
<b>2.5</b>	<b>PERIPHERALS</b>	<b>30</b>
2.5.1	GPIO	30
2.5.2	DEBUG PRINTING	31
2.5.3	PUART (PERIPHERAL UART)	33
2.5.4	TIMERS	34
2.5.5	PWM	35
2.5.6	NVRAM	37
2.5.7	I2C	38
2.5.8	ADC	40
2.5.9	RTC (REAL TIME CLOCK)	41
<b>2.6</b>	<b>WICED_RESULT_T</b>	<b>42</b>
<b>2.7</b>	<b>EXERCISES</b>	<b>44</b>
	EXERCISE - 2.1 (GPIO) BLINK AN LED	44
	EXERCISE - 2.2 (GPIO) ADD DEBUG PRINTING TO THE LED BLINK PROJECT	46
	EXERCISE - 2.3 (GPIO) READ THE STATE OF A MECHANICAL BUTTON	47
	EXERCISE - 2.4 (GPIO) USE AN INTERRUPT TO TOGGLE THE STATE OF AN LED	47
	EXERCISE - 2.5 (TIMER) USE A TIMER TO TOGGLE AN LED	47
	EXERCISE - 2.6 (PWM) LED BRIGHTNESS	48
	EXERCISE - 2.7 (PWM) LED TOGGING AT SPECIFIC FREQUENCY AND DUTY CYCLE	49
	EXERCISE - 2.8 (I2C) READ MOTION SENSOR DATA	50
	EXERCISE - 2.9 (ADVANCED) (NVRAM) WRITE AND READ DATA IN THE NVRAM	51
	EXERCISE - 2.10 (ADVANCED) (ADC) CALCULATE THE RESISTANCE OF A THERMISTOR	52
	EXERCISE - 2.11 (ADVANCED) (UART) SEND A VALUE USING THE STANDARD UART FUNCTIONS	53
	EXERCISE - 2.12 (ADVANCED) (UART) GET A VALUE USING THE STANDARD UART FUNCTIONS	53
	EXERCISE - 2.13 (ADVANCED) (RTC) DISPLAY TIME AND DATE DATA ON THE UART	54

## 2.1 Creating a new ModusToolbox IDE project

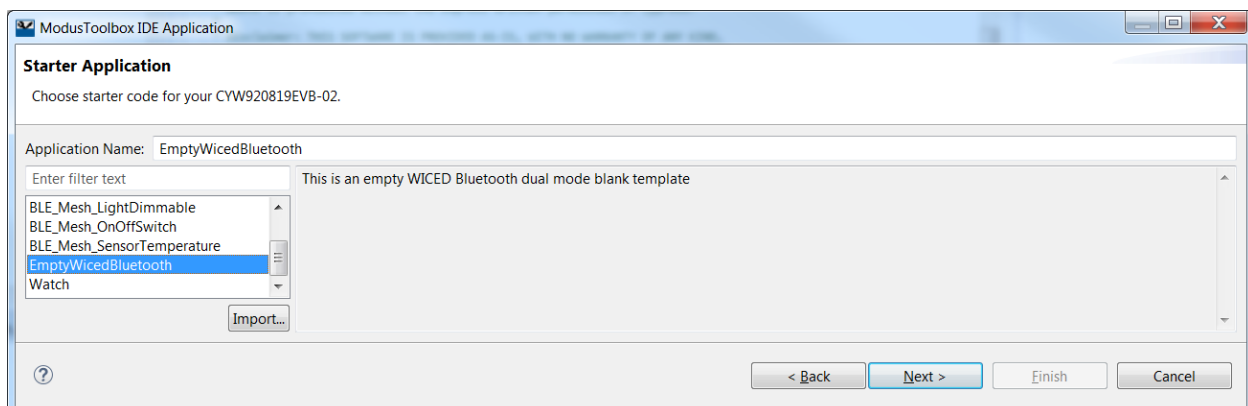
### 2.1.1 New Application Wizard

ModusToolbox IDE includes a wizard that sets up new applications with the required target hardware support, Bluetooth configuration code, middleware libraries, build, program and debug settings, and a "starter" application. Launch the wizard from the "New Application" button in the Quick Panel or use "New ModusToolbox Application" item in the *File->New menu*.

It is a multi-step wizard that first asks you to select the "Target Hardware". Note that it is possible for users to create custom Bluetooth SoC board support packages (BSPs) but that is beyond the scope of this course. The kits used in this course are the [CYBT-213043-MESH](#) and [CYW920819EVB-02](#).

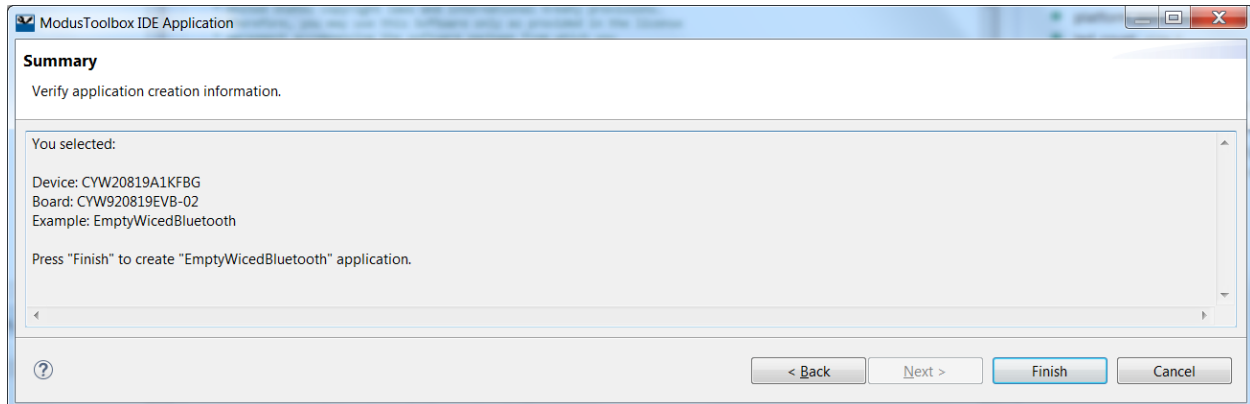


Clicking "Next" lets you choose the Starter Application from a list of applications that support the selected hardware. Note that there is no real equivalent to the PSoC Creator empty project, but there is an EmptyWicedBluetooth application that gives you just enough to get started. Unlike the PSoC Creator empty project, using this as a starting point is more of an expert-level approach since it doesn't provide much in the way of code. In most cases, we recommend starting with one of the code examples and modifying it as needed for your application.



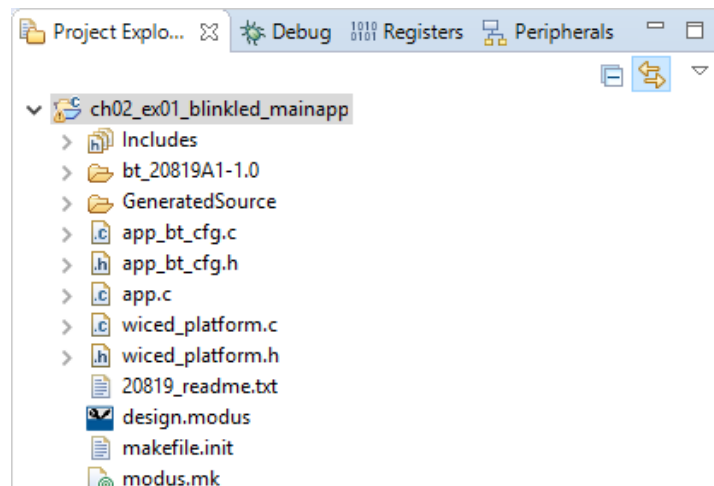
Each starter application includes a description and a default name, which you can modify if you wish (if you create multiple copies of the same template without changing the name the tool adds an incrementing number to each created project name).

Pressing "Next" takes you to the final step, where you verify your choices and press "Finish" to create the application.



## 2.1.2 Directory Structure

ModusToolbox IDE uses Eclipse workspaces as containers for projects. All projects within a workspace reside at the top level of the workspace folder. You can choose any legal folder/file name for the project, but you cannot change its location on disk.

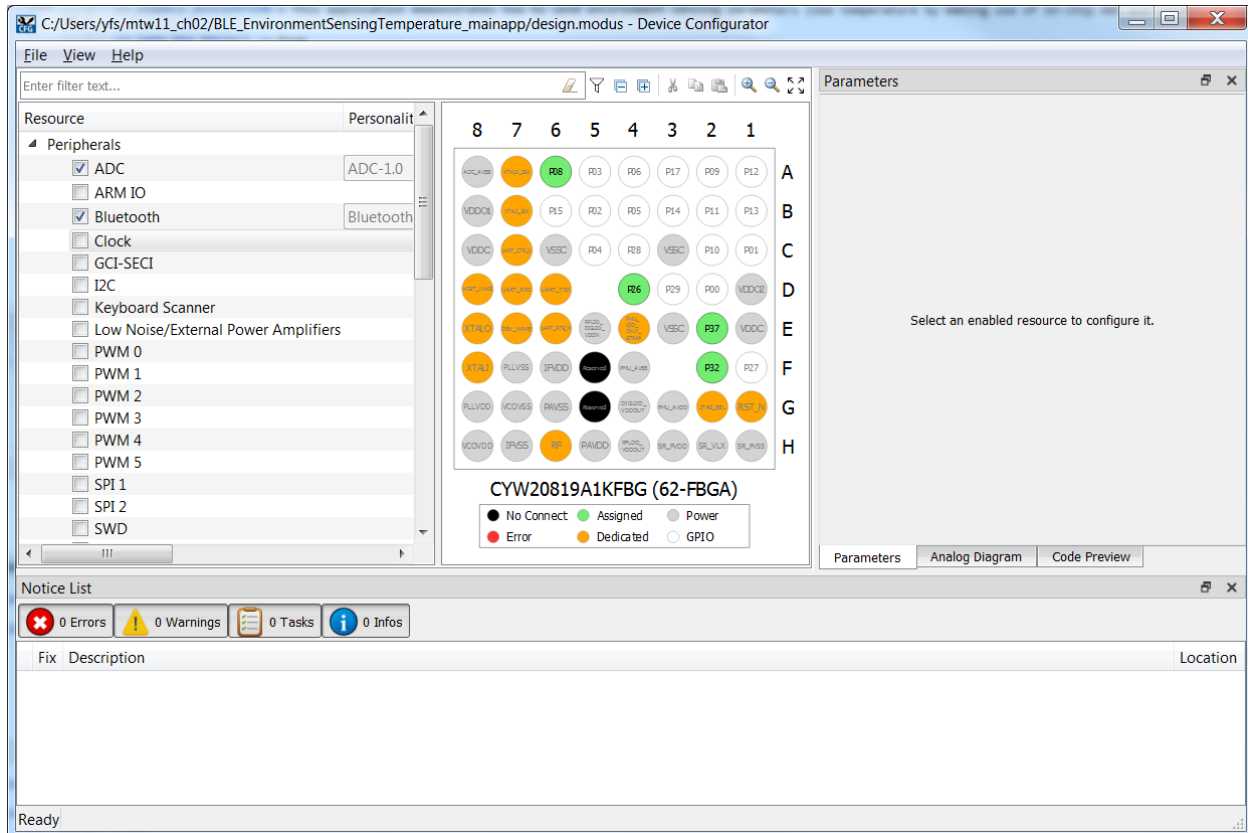


The key parts of a project are:

- A folder with the name of the project.
- The Device Configurator database file – design.modus
- A GeneratedSource folder with the output files from the Configurator tool
- Platform files (wiced\_platform.\*)
- Stack configuration files (app\_bt\_cfg.\*)
- Application C source files

### 2.1.3 Configuration Database - design.modus File

The design.modus file is the database of configuration choices for the device. It is the source document for the Device Configurator tool which presents a list of peripherals on the device that you can set up.



When you create a new project in ModusToolbox IDE you always select the target board and starter application. These choices give you a pre-configured design.modus file that sets up the usual hardware, such as buttons and LEDs, so you do not need to make the same edits every time you create a new project. You can, of course, modify these settings, for example to drive an LED with a PWM instead of firmware. This is discussed later, in the Board Support Package section.

### 2.1.4 GeneratedSource Folder

When you save the device configuration, the tool generates firmware in the GeneratedSource folder. The following are the most interesting/useful files.

cycfg\_pins.c and cycfg\_pins.h (definitions and setup code for GPIO, LEDs and Buttons)

cycfg\_notices.h (enables error reporting in the ModusToolbox IDE Console view)

### 2.1.5 Platform Files (wiced\_platform.\*)

These files enable the startup code to configure all the GPIOs by calling the `wiced_platform_init()` function. This function loops through all the GPIOs, LEDs and Buttons on the board, setting them up according to the design.modus settings in the `cycfg_pins.c` file.

The `wiced_platform.h` file defines useful labels for the configured pins that you can use in your code to create reliable, portable applications. Here are the defines you will be using for buttons and pins on the CYW920819EVB-02 kit.

```
/*! pin for button 1 */
#define WICED_GPIO_PIN_BUTTON_1    WICED_P00

/*! pin for LED 1 */
#define WICED_GPIO_PIN_LED_1      WICED_P27
/*! pin for LED 2 */
#define WICED_GPIO_PIN_LED_2      WICED_P26
```

Note that in future versions of ModusToolbox (2.0 and beyond) these files will no longer exist. The pertinent information will instead be incorporated into the `design.modus` file so that everything exists in one consistent place.

### 2.1.6 Stack Configuration Files

Most templates include `app_bt_cfg.c` and `app_bt_cfg.h`, which create static definitions of the stack configuration and buffer pools. You will edit the stack configuration, for example, to change the name of your device or to optimize the scanning and advertising parameters. The buffer pools determine the availability of various sizes of memory blocks for the stack, and you might edit those to optimize performance and RAM usage.

Note: The actual file names may vary in some code examples, but the definitions of the `wiced_bt_cfg_settings` struct and `wiced_bt_cfg_buf_pools` array are required.

### 2.1.7 Application C file

Most templates include an application C file that is expected to be edited by the user to (at least) start up the application. The actual name of this file varies according to the template used to create the application. It is usually the same name as the template but there are exceptions to that rule.

Note: MESH examples are implemented with a library called `mesh_app_lib`. The library includes the `application_start()` function in `mesh_application.c` inside the library. This is because the application itself is very regimented, and the developer does not really "own" the way devices interact. The user part of the application is restricted to activity supported by the device's capabilities, such as dimming the light with a PWM or controlling a door lock solenoid.

The application C file begins with various `#include` lines depending on the resources used in your application. These files can be found in the SDK under `ModusToolbox_1.1\libraries\bt_20819A1-1.0\components\BT-SDK`. A few examples are shown below. The first 4 are usually required in any

project. The "hal" includes are only needed if the specific peripheral are used in the project, e.g. `wiced_hal_i2c.h` is required if you are using I2C.

```
#include "wiced.h"           // Basic formats like stdint, wiced_result_t, WICED_FALSE, WICED_TRUE
#include "wiced_platform.h"   // Platform file for the kit
#include "sparcommon.h"       // Common application definitions
#include "wiced_bt_stack.h"   // Bluetooth Stack
#include "wiced_bt_dev.h"     // Bluetooth Management
#include "wiced_bt_ble.h"     // BLE
#include "wiced_bt_gatt.h"    // BLE GATT database
#include "wiced_bt_uuid.h"    // BLE standard UUIDs
#include "wiced_rtos.h"       // RTOS
#include "wiced_bt_app_common.h" // Miscellaneous helper functions including wiced_bt_app_init
#include "wiced_transport.h"   // HCI UART drivers
#include "wiced_bt_trace.h"   // Trace message utilities
#include "wiced_timer.h"      // Built-in timer drivers
#include "wiced_hal_i2c.h"    // I2C drivers
#include "wiced_hal_adc.h"    // ADC drivers
#include "wiced_hal_pwm.h"    // PWM drivers
#include "wiced_hal_puart.h"  // PUART drivers
#include "wiced_rtos.h"       // RTOS functions
#include "wiced_hal_nvrnm.h"  // NVRAM drivers
#include "wiced_hal_wdog.h"   // Watchdog
#include "wiced_spar_utils.h" // Required for stdio functions such as sprintf and snprintf
#include <stdio.h>             // Stdio C functions such as printf and sprintf
```

After the includes list you will find the `application_start()` function, which is the main entry point into the application. That function typically does a minimal amount of initialization then it starts the Bluetooth stack and registers a stack callback function by calling `wiced_bt_stack_init()`. Note that the configuration parameters from `app_bt_cfg.c` are provided to the stack here. The callback function is called by the stack whenever it has an event that the user's application might need to know about. It typically controls the rest of the application based on Bluetooth events.

Most application initialization is done once the Bluetooth stack has been enabled. That event is called `BTM_ENABLED_EVT` in the callback function. The full list of events from the Bluetooth stack can be found in the file `ModusToolbox_1.1/libraries/bt_20819A1-1.0/components/BT-SDK/20819-A1_Bluetooth/include/20819/wiced_bt_dev.h`.

A minimal C file for an application will look something like this:

```
#include "sparcommon.h"
#include "wiced_platform.h"
#include "wiced_bt_dev.h"
#include "wiced_bt_stack.h"
#include "app_bt_cfg.h"

wiced_bt_dev_status_t app_bt_management_callback( wiced_bt_management_evt_t event,
wiced_bt_management_evt_data_t *p_event_data );

void application_start(void)
{
    wiced_bt_stack_init( app_bt_management_callback, &wiced_bt_cfg_settings,
                        wiced_bt_cfg_buf_pools );
}
```

```

}

wiced_result_t app_bt_management_callback( wiced_bt_management_evt_t event,
                                           wiced_bt_management_evt_data_t *p_event_data )
{
    wiced_result_t status = WICED_BT_SUCCESS;

    switch( event )
    {
    case BTM_ENABLED_EVT:                // Bluetooth Controller and Host Stack Enabled

        if( WICED_BT_SUCCESS == p_event_data->enabled.status )
        {
            /* Initialize and start your application here once the BT stack is running */
        }
        break;

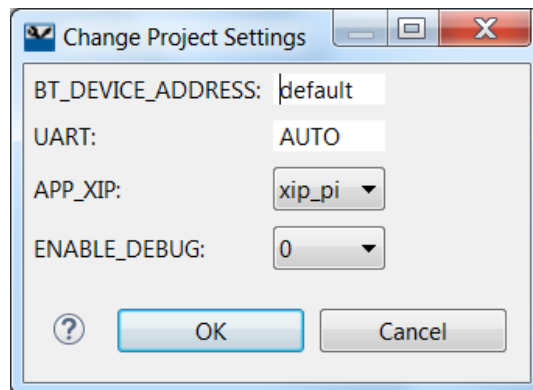
    default:
        break;
    }

    return status;
}

```

### 2.1.8 Application Settings

Unlike WICED Studio, ModusToolbox does not require the user to create and maintain a make target for every application. Source file compilation is handled by the IDE automatically. Useful application-level settings, such as debugging, are provided by the template and can be modified by right-clicking on the project name in the Explorer view and selecting "Change Application Settings".

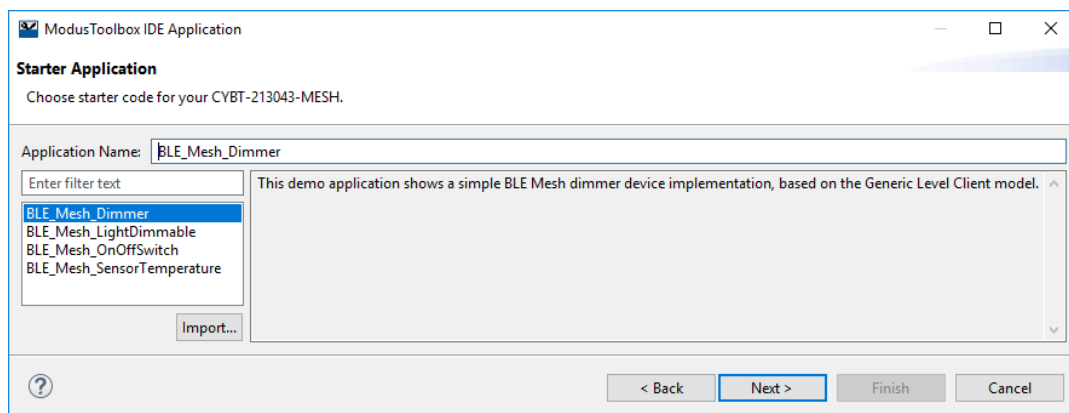
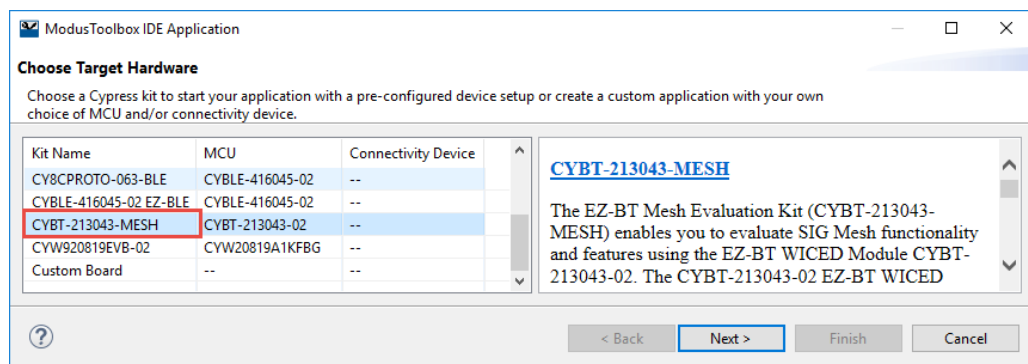


## 2.2 Code Examples

ModusToolbox contains a wealth of code examples. Some code examples are built into the Bluetooth SDK while others are available for download from GitHub. The online code examples are generally categorized in one of two types: demo (fully featured applications) and snip (examples that show one or a few concepts).

### 2.2.1 Built-In Code Examples

For the built-in examples, you can just choose "New Application" from the Quick Panel in the ModusToolbox IDE, select one of the 20819 kits (CYW920819EVB-02 or CYBT-213043-MESH), and then pick one of the available starter applications:



### 2.2.2 GitHub Code Examples

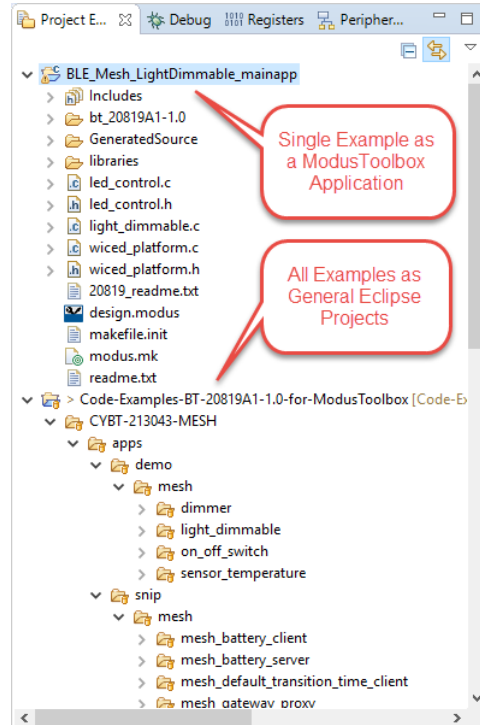
To search for code examples online, click on "Search Online for Code Examples" from the Quick Panel. This opens a web browser to the GitHub repository. From the web browser, choose the *20189A1 Bluetooth Examples* repository. This will present you with a repository containing code examples for the CYW920819EVB-02 and CYBT-213043-MESH kits. Under each kit, you can browse through the demo and snip applications under the various folders.

A direct link to the 20819 Bluetooth code example repository is:

<https://github.com/cypresssemiconductorco/Code-Examples-BT-20819A1-1.0-for-ModusToolbox>



There are two basic ways to use the code example repository. First, you can import all the examples at once into ModusToolbox IDE as general Eclipse projects. This allows you to view (and copy from) the source code of all examples, but the examples cannot be built as ModusToolbox applications. Second, you can create a new ModusToolbox application from any of the code examples. This gives you a full application that you can build and download to a device. Note that you can use both methods in a single workspace if you so desire.

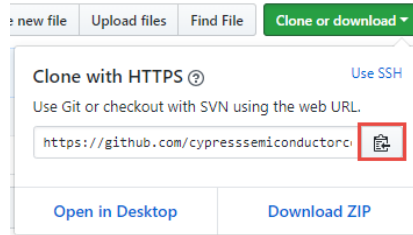


There are three ways to get a copy of the entire repository. They are:

1. Clone the repository and import projects as general Eclipse projects directly from GitHub as a single step in ModusToolbox IDE. Then create individual ModusToolbox applications in IDE as desired.
2. Use Git Clone to get a copy of the repository on your local machine. Then import all projects as general Eclipse projects and/or create individual ModusToolbox applications as desired.
3. Download a Zip file from GitHub and unzip on your local machine. Then import all projects as general Eclipse projects and/or create individual ModusToolbox applications as desired.

Note: You don't need to clone or download a zip of the repository for every workspace – just one copy is enough. After that you can import all the projects as existing Eclipse projects or into any workspace or create new ModusToolbox applications based on any individual examples. The methods for all these operations are detailed below.

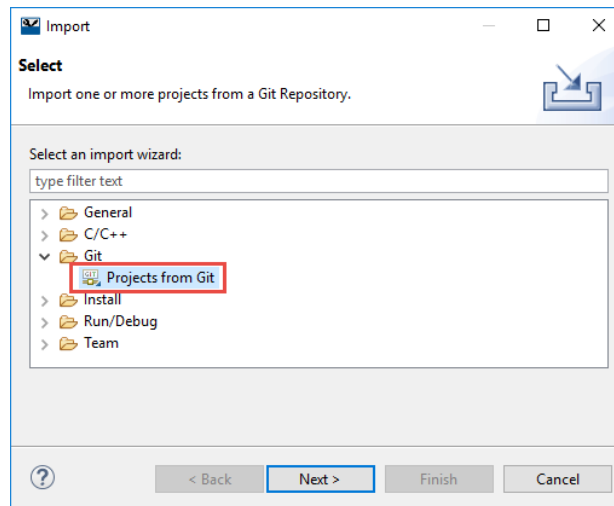
Note: From the GitHub page, you can click the button "Clone or download" to open a window that will allow you to download a Zip file or to copy the URI for the repository to the clipboard to use in cloning (the copy button is shown with a red box around it in the figure below).



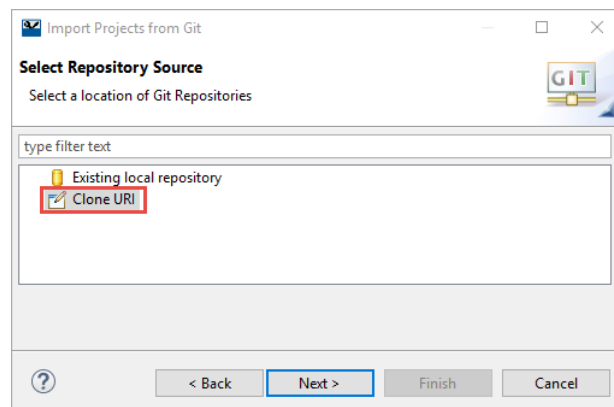
## Method 1: Clone and Import all Eclipse Projects directly from GitHub

The steps to clone a repository from GitHub and import all the projects in that repository as general Eclipse projects from within ModusToolbox IDE are shown below. Note that this should be done only if you have not previously downloaded the repository.

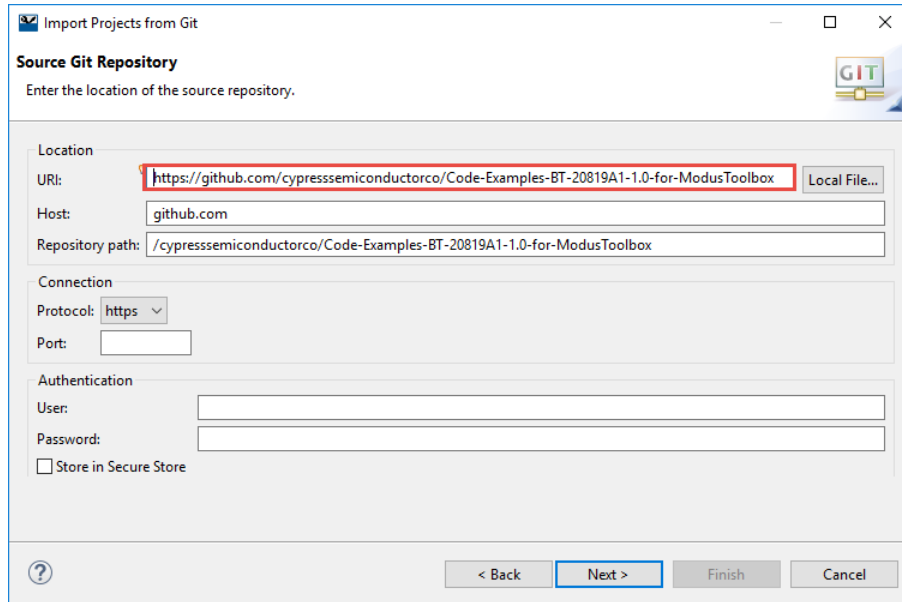
1. Choose File -> Import -> Git -> Projects from Git and click Next.



2. Choose Clone URI and click Next.



3. Enter the URI. The host and repository path will be filled in automatically. Click Next.
  - a. Hint: Remember that you can copy the URI from the "Clone or download" window from GitHub so you don't need to type it in manually.

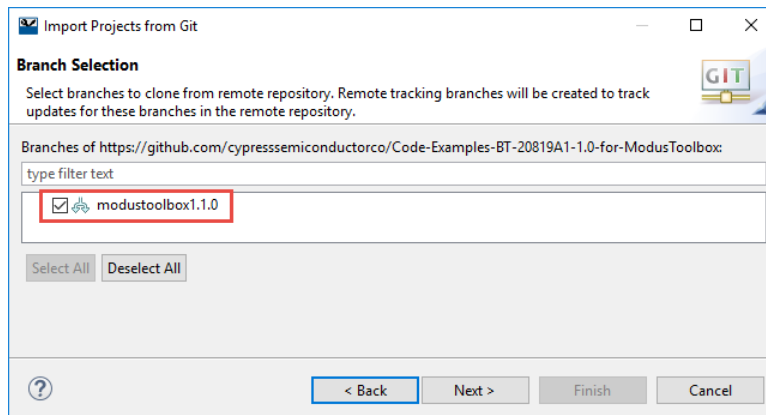


The "Import Projects from Git" dialog box is shown. The "Source Git Repository" section is active, with the instruction "Enter the location of the source repository." The "Location" section contains the following fields:

- URI: `https://github.com/cypresssemiconductorco/Code-Examples-BT-20819A1-1.0-for-ModusToolbox` (highlighted with a red box)
- Host: `github.com`
- Repository path: `/cypresssemiconductorco/Code-Examples-BT-20819A1-1.0-for-ModusToolbox`

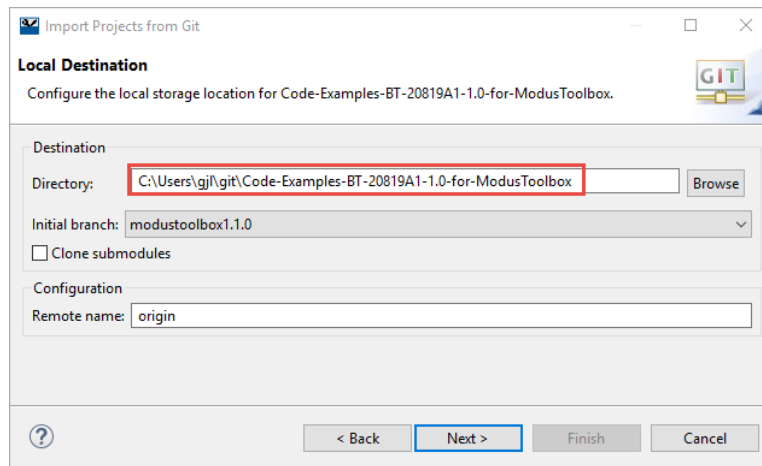
The "Connection" section shows the Protocol set to `https` and the Port field is empty. The "Authentication" section has fields for User and Password, and a checkbox for "Store in Secure Store" which is unchecked. At the bottom, there are buttons for "< Back", "Next >" (highlighted with a blue box), "Finish", and "Cancel".

4. Select the branch to download and click Next. The branch you choose will depend on the version of ModusToolbox that you are using.

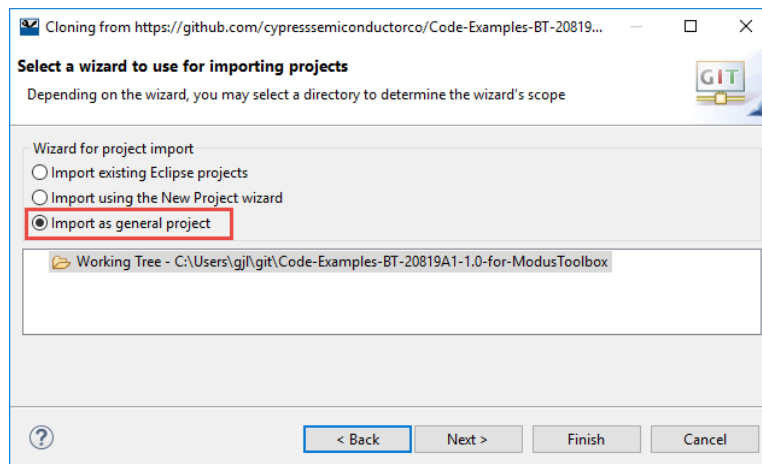


The "Branch Selection" dialog box is shown. The instruction "Select branches to clone from remote repository. Remote tracking branches will be created to track updates for these branches in the remote repository." is displayed. The "Branches of https://github.com/cypresssemiconductorco/Code-Examples-BT-20819A1-1.0-for-ModusToolbox:" section contains a list of branches. The branch `modustoolbox1.1.0` is selected (checked) and highlighted with a red box. Below the list are "Select All" and "Deselect All" buttons. At the bottom, there are buttons for "< Back" (highlighted with a blue box), "Next >", "Finish", and "Cancel".

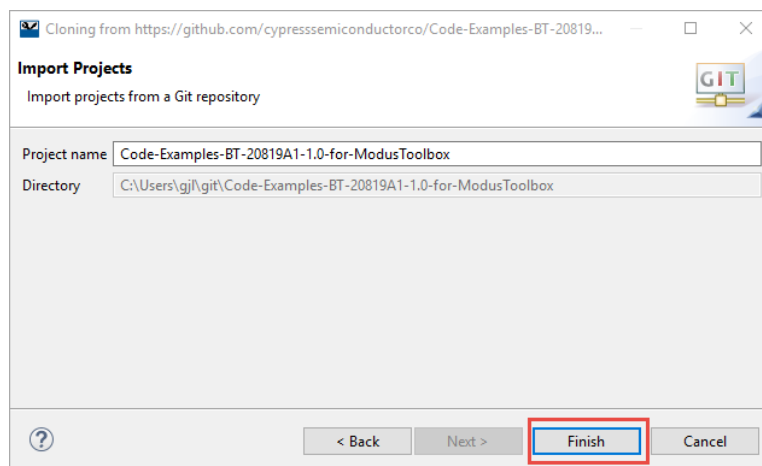
- Choose a destination on the local machine for the repository and click Next



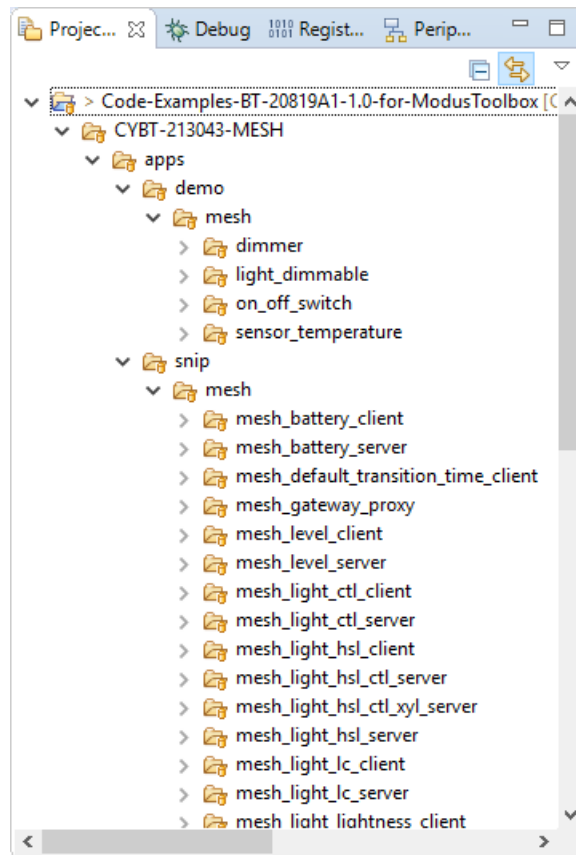
- Choose Import as general project and click Next



- Use the provided project name or enter another name of your choice. Click Finish to complete the import.



8. Projects will appear hierarchically in the Project Explorer Window as shown below.

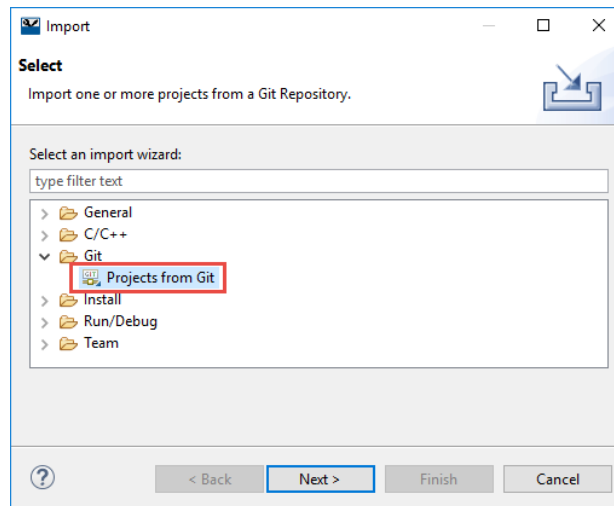


If you want to skip the other two methods for importing the entire repository, click [here](#).

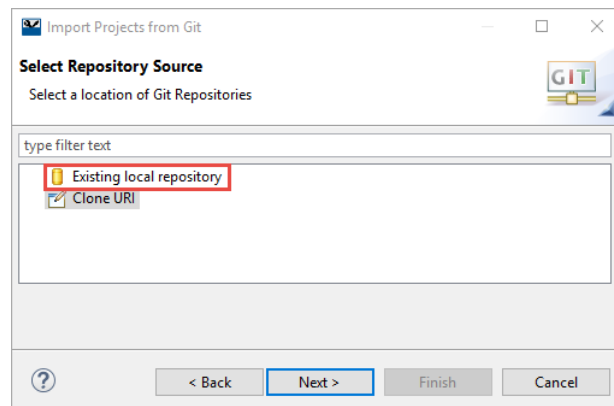
## Method 2: Clone from GitHub, then Import all Eclipse Projects (if desired)

If you use Git Clone using normal Git commands to get a copy of the repository outside of ModusToolbox IDE, you can still import all the projects into a workspace as general Eclipse projects. This will allow you to view/copy source code of all the applications without having to import each one as a ModusToolbox application. The steps are as follows:

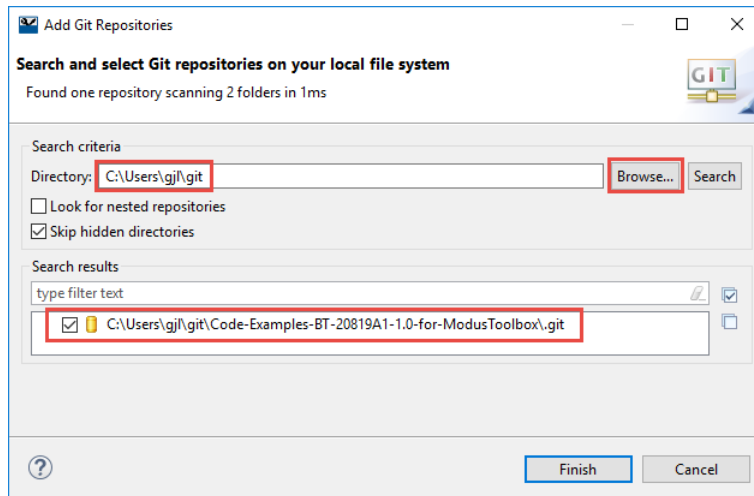
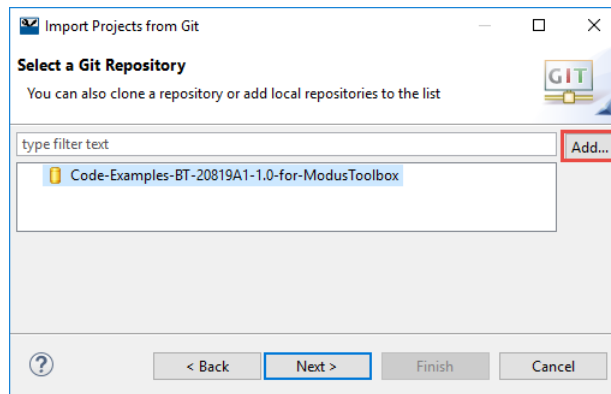
1. Choose File -> Import -> Git -> Projects from Git and click Next



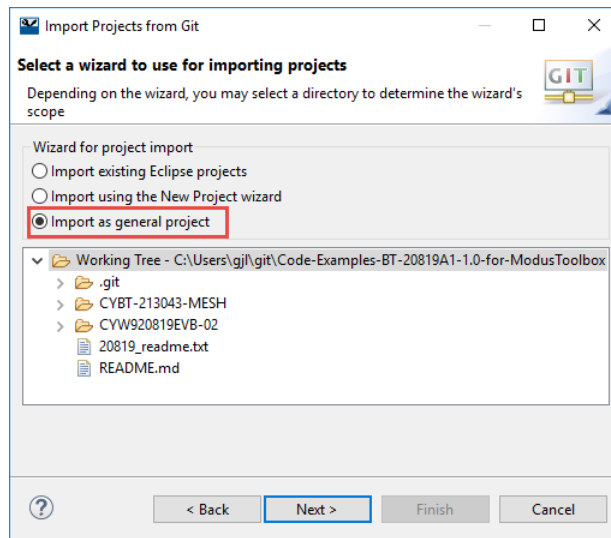
2. Choose Existing local repository and click Next.



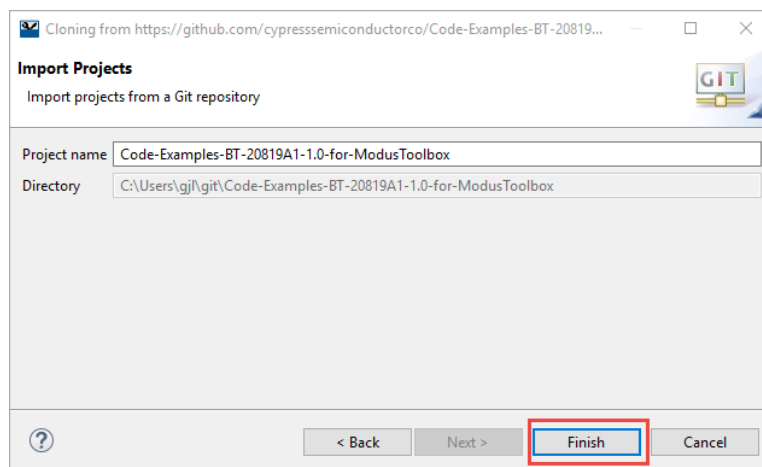
3. Click Add then Browse to specify the path to the local Git repository. Check the box next to the repository desired. Click Finish, then Next.



4. Select Import as general project and click Next.

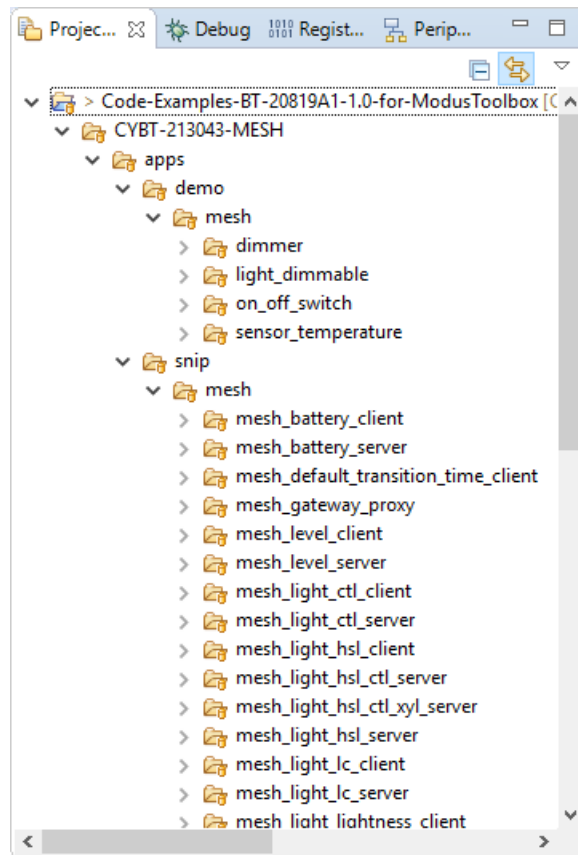


5. Use the provided project name or enter another name of your choice. Click Finish to complete the import.





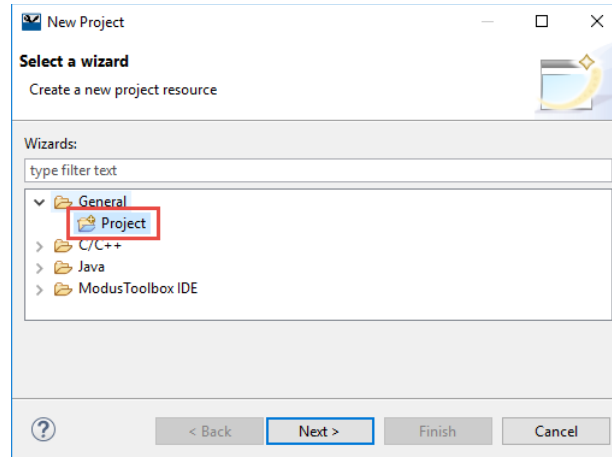
6. Projects will appear hierarchically in the Project Explorer Window as shown below.



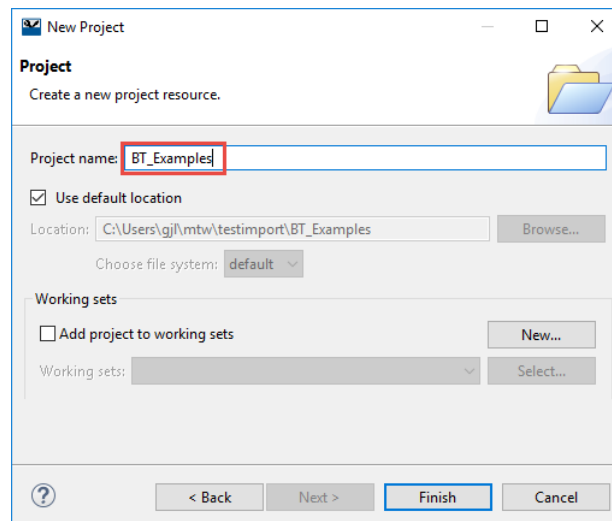
### Method 3: Download Zip from GitHub then Import all Eclipse Projects (if desired)

If you choose to download a zip file from GitHub, save the file to a location on your local machine and unzip it. Once you have done that, if you want to import all the examples as general Eclipse projects you must first create a top-level project and then import the examples. The procedure is as follows:

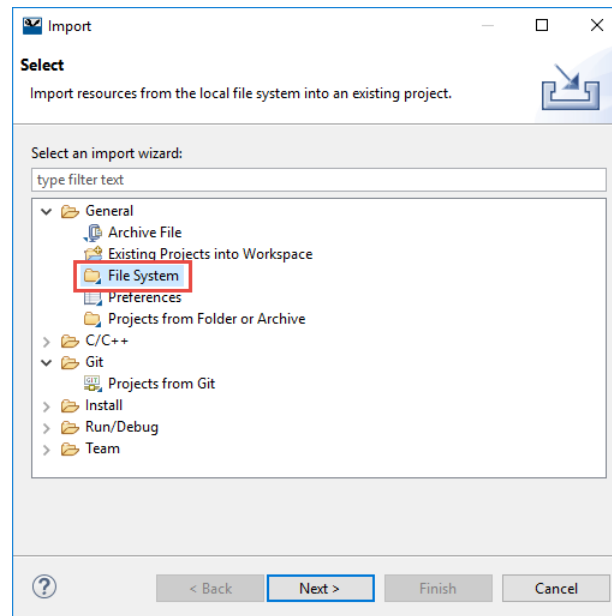
1. Choose File -> New -> Project -> General -> Project and click Next.



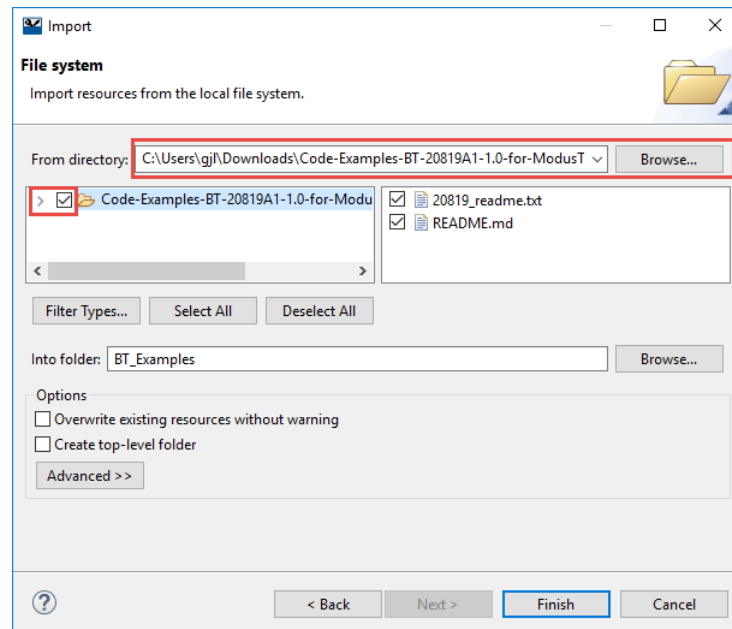
2. Give the project a name. The default location will be your current workspace. Click Finish. This will create a new general Eclipse project.



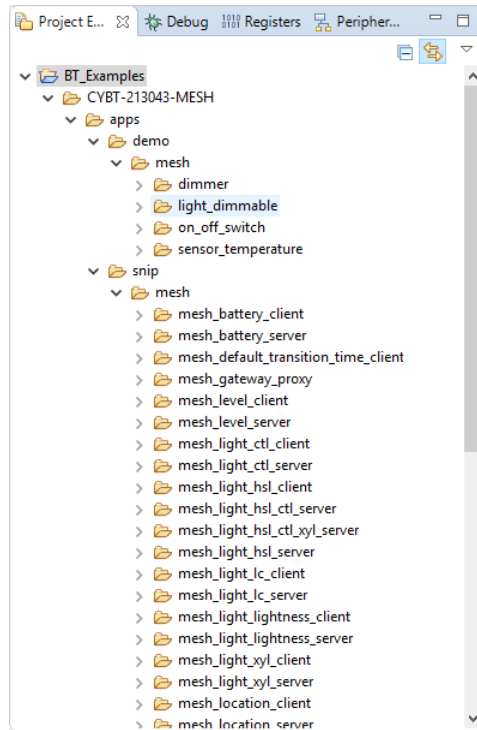
3. Choose File -> Import -> General -> File System.



4. Click Browse to choose the location where you unzipped the file. Check the box next to the folder name and then click Finish.



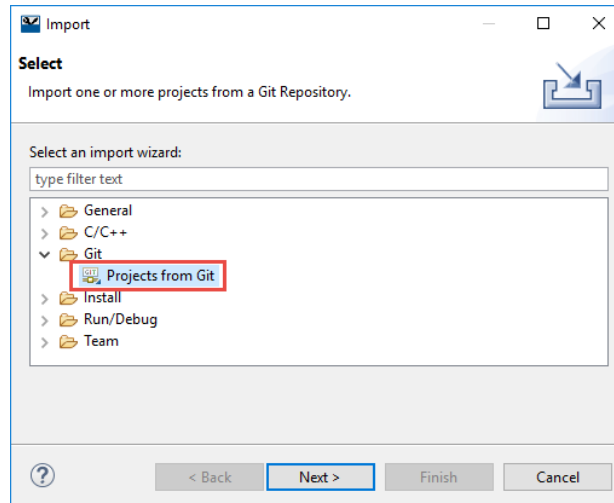
5. Projects will appear hierarchically in the Project Explorer Window as shown below.



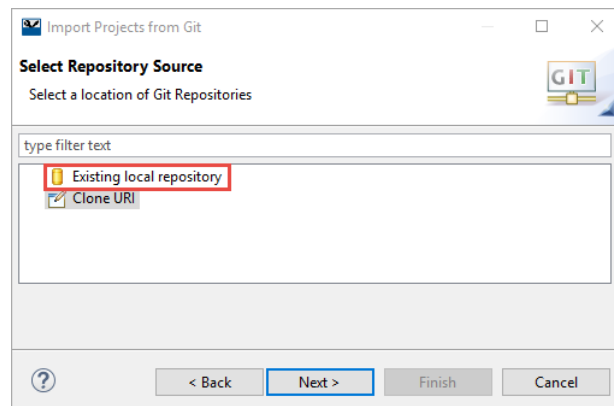
## Importing all Examples as General Eclipse Projects from a local Git Repository to another Workspace

If you already have a local Git repository (either from method 1 or method 2) and have already imported the projects as general Eclipse projects but want to import them into another workspace, the procedure is as shown below. (The only difference from what was done on the first import from a Git repository is that you must choose to Import existing Eclipse projects instead of Import as general project.)

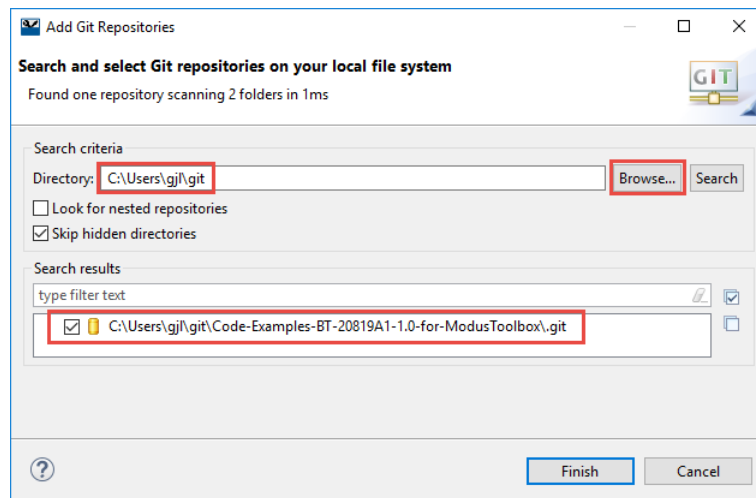
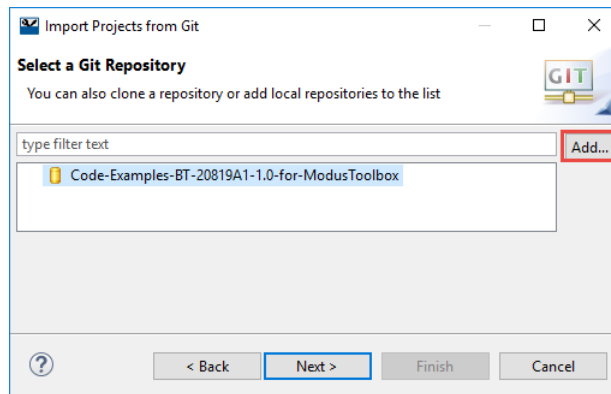
1. Choose File -> Import -> Git -> Projects from Git and click Next



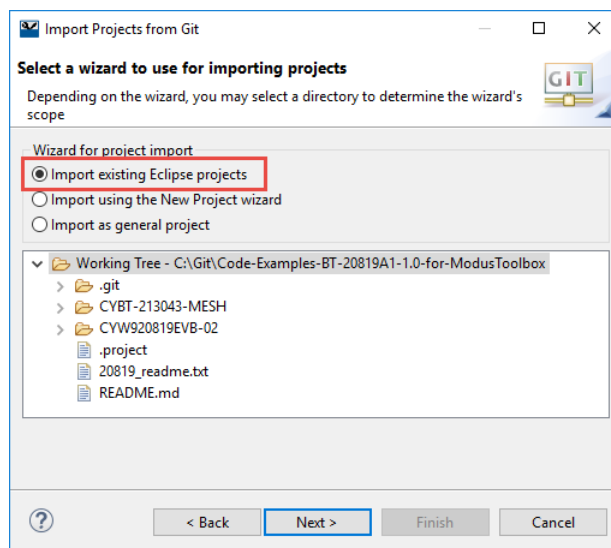
2. Choose Existing local repository and click Next.



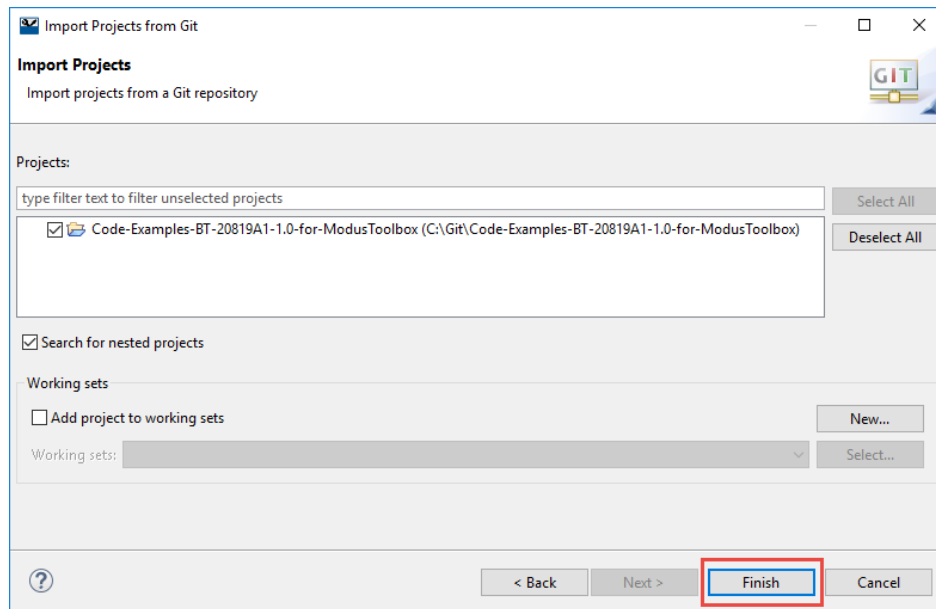
- Click Add then Browse to specify the path to the local Git repository. Check the box next to the repository desired. Click Finish, then Next.



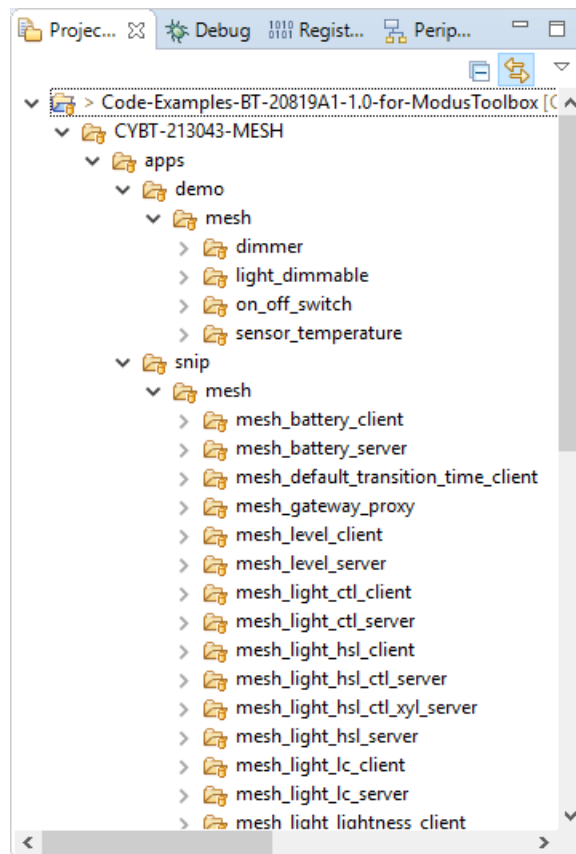
- Select Import existing Eclipse projects and click Next.



- Click Finish to complete the import.



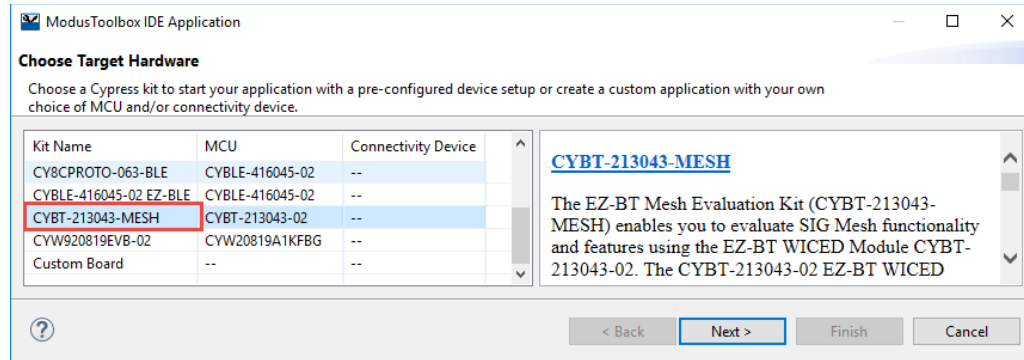
- Projects will appear hierarchically in the Project Explorer Window as shown below.



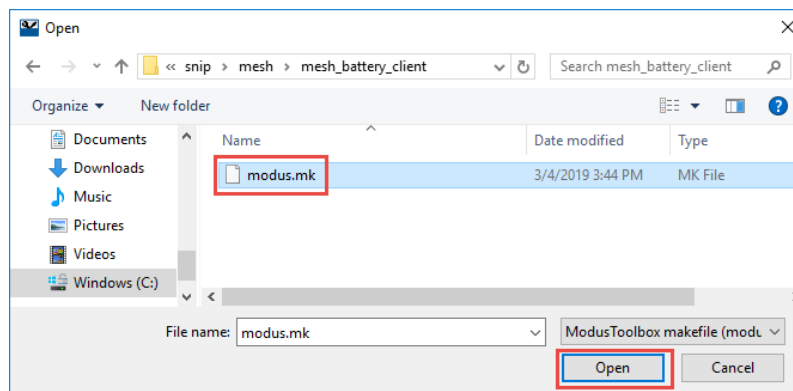
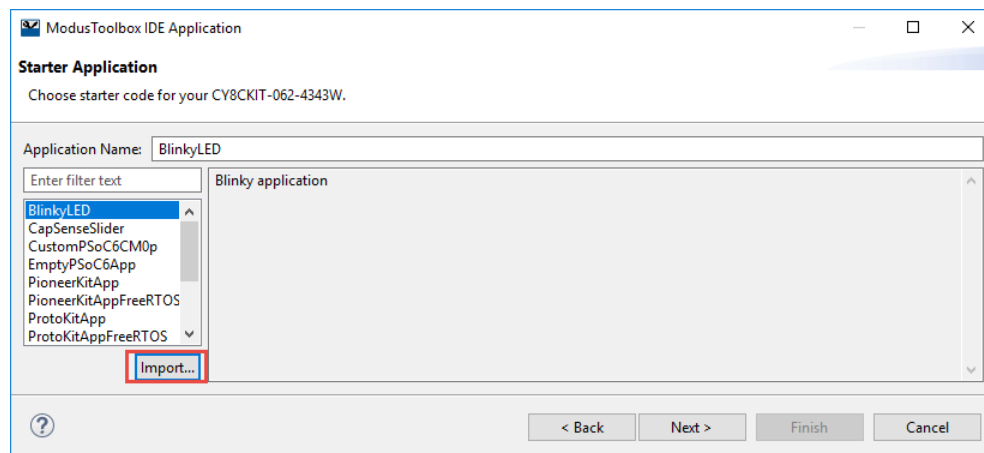
## Importing a Single ModusToolbox Example Application

Once you have a copy of the repository (using any of the three methods described above) you just use "New Application" from the Quick Panel or choose File -> New -> ModusToolbox IDE Application. Then follow the steps as shown:

1. Select the desired target hardware and click Next.

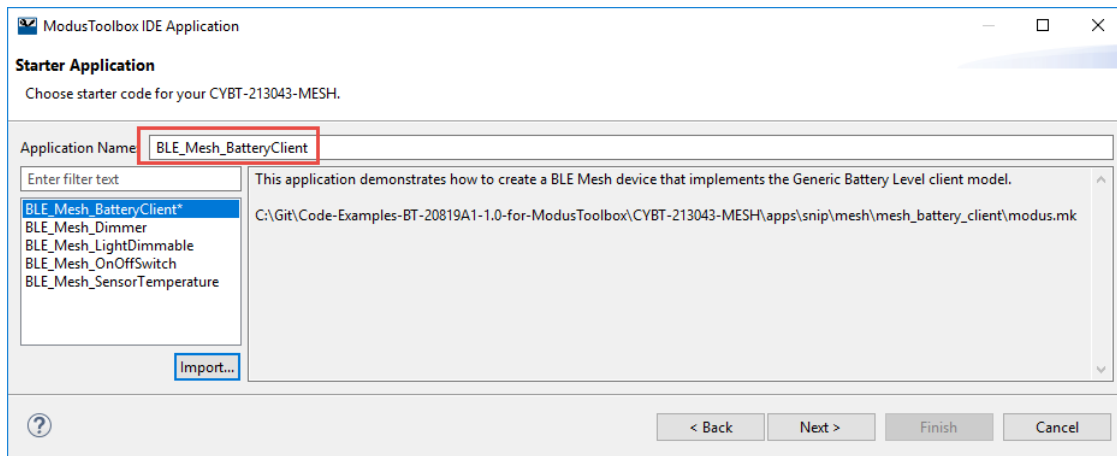


2. From the Starter Application window, Click Import and navigate to the desired application. Select the "modus.mk" file and click Open.





3. Change the Application name if desired and click Next.



4. Click Finish. This will create a complete ModusToolbox application that you can build and program. This can be done as often as you like and into as many workspaces as you want.

## 2.3 Board Support Package

Every ModusToolbox application makes use of a Board Support Package (BSP), which is chosen in the first step of the New Application wizard (every kit has exactly one BSP). The BSP makes it easier to work with the peripherals on the kit. A BSP is a collection of files that specify which device is on the board, how it is programmed, what peripherals are available, which pins they are mapped to, etc.

In case you design your own hardware, you can either create a new BSP (beyond the scope of this course) or create an application with the BSP that most closely matches your hardware then modify the files as necessary. For example, you may have buttons and LEDs connected to different pins, so you would update the appropriate file to make those changes for your hardware.

Most of the configurable content of the BSP is managed by the design.modus file, which is the database for the Device Configurator. It generates the GeneratedSource/cycfg\_pins.c file, which contains constant arrays that are used to configure the peripherals and to initialize the pins to the correct state. For example, LED pins are initialized as outputs, and the button pins are initialized as inputs with a resistive pullup. For example, for LED1:

```
#define LED1_config \
{\
    .gpio = (wiced_bt_gpio_numbers_t*)&platform_gpio_pins[PLATFORM_GPIO_3].gpio_pin, \
    .config = GPIO_OUTPUT_ENABLE | GPIO_PULL_UP, \
    .default_state = GPIO_PIN_OUTPUT_HIGH, \
}
```

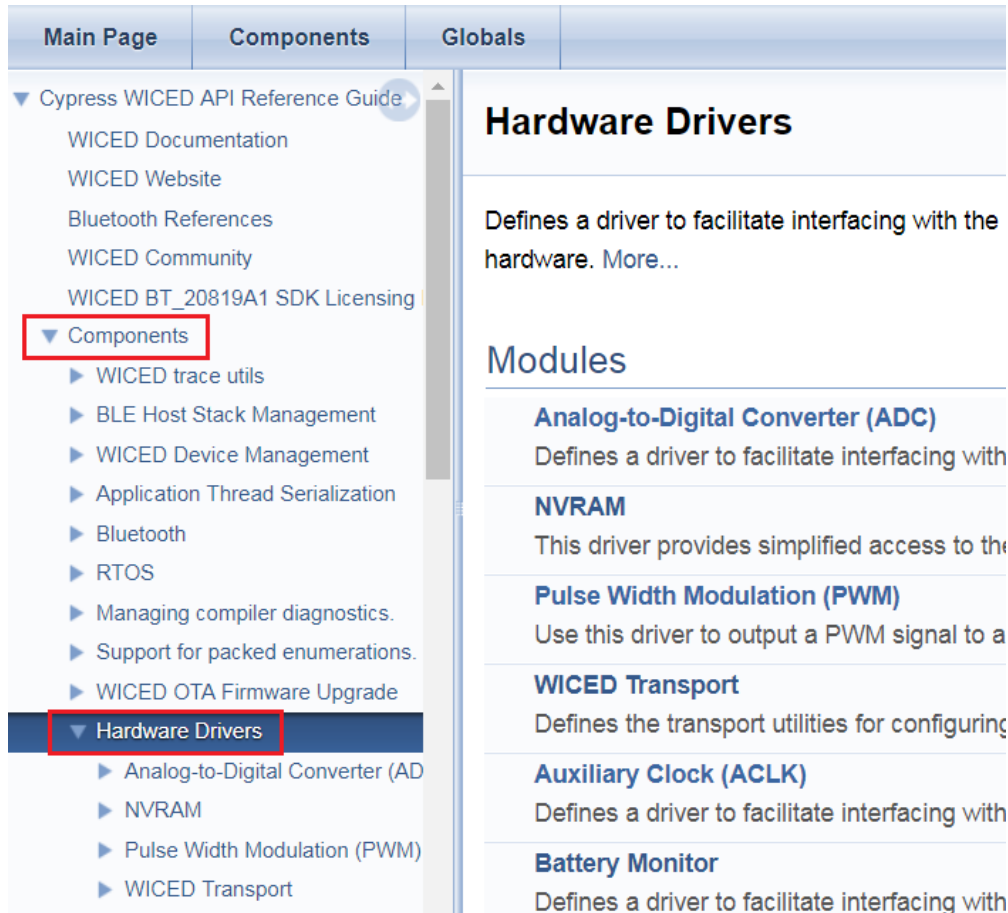
The other key file here is wiced\_platform.h, which contains #define and type definitions used to set up and access the various kit peripherals. For example, the CYW920819EVB-02 kit contains two LEDs and one mechanical button. These are identified in wiced\_platform.h using the names WICED\_GPIO\_PIN\_BUTTON\_1, WICED\_GPIO\_PIN\_LED\_1 and WICED\_GPIO\_PIN\_LED\_2:

```
/*! pin for button 1 */
#define WICED_GPIO_PIN_BUTTON_1      WICED_P00

/*! pin for LED 1 */
#define WICED_GPIO_PIN_LED_1         WICED_P28
/*! pin for LED 2 */
#define WICED_GPIO_PIN_LED_2         WICED_P26
```

## 2.4 Documentation

CPU peripheral documentation can be found in the Documents tab in the lower left panel (next to the Quick Panel) or in the Help menu under *ModusToolbox API Reference->WICED API Reference*. The peripheral APIs are presented under Components->Hardware Drivers as shown below. We will be using GPIO, Pulse Width Modulation (PWM), Peripheral UART (PUART), I2C, and Real-Time Clock (RTC).



The screenshot displays the Cypress WICED API Reference Guide interface. The left sidebar contains a navigation menu with the following items:

- ▼ Cypress WICED API Reference Guide
  - WICED Documentation
  - WICED Website
  - Bluetooth References
  - WICED Community
  - WICED BT\_20819A1 SDK Licensing
  - ▼ Components
    - ▶ WICED trace utils
    - ▶ BLE Host Stack Management
    - ▶ WICED Device Management
    - ▶ Application Thread Serialization
    - ▶ Bluetooth
    - ▶ RTOS
    - ▶ Managing compiler diagnostics.
    - ▶ Support for packed enumerations.
    - ▶ WICED OTA Firmware Upgrade
  - ▼ Hardware Drivers
    - ▶ Analog-to-Digital Converter (ADC)
    - ▶ NVRAM
    - ▶ Pulse Width Modulation (PWM)
    - ▶ WICED Transport

The main content area is titled **Hardware Drivers** and contains the following text:

Defines a driver to facilitate interfacing with the hardware. [More...](#)

**Modules**

- Analog-to-Digital Converter (ADC)**  
Defines a driver to facilitate interfacing with
- NVRAM**  
This driver provides simplified access to the
- Pulse Width Modulation (PWM)**  
Use this driver to output a PWM signal to a
- WICED Transport**  
Defines the transport utilities for configuring
- Auxiliary Clock (ACLK)**  
Defines a driver to facilitate interfacing with
- Battery Monitor**  
Defines a driver to facilitate interfacing with

Click on GPIO to see the list of GPIO APIs, and then click on the *wiced\_hal\_gpio\_configure\_pin* function for a description.

```
void wiced_hal_gpio_configure_pin ( UINT32 pin,  
                                   UINT32 config,  
                                   UINT32 outputVal  
                                   )
```

Configures a GPIO pin.

Note that the GPIO output value is programmed before the GPIO is configured. This ensures that the GPIO will activate with the correct external value. Also note that the output value is always written to the output register regardless of whether the GPIO is configured as input or output.

Enabling interrupts here isn't sufficient; you also want to register with `registerForMask()`.

All input parameter values must be in range or the function will have no effect.

#### Parameters

**pin** - pin id (0-39).

**config** - Gpio configuration. See Parameters section above for possible values.

For example, to enable interrupts for all edges, with a pull-down, you could use:

```
/// GPIO_EDGE_TRIGGER | GPIO_EDGE_TRIGGER_BOTH |  
/// GPIO_INTERRUPT_ENABLE_MASK | GPIO_PULL_DOWN_MASK  
///
```

```
\param outputVal - output value.
```

```
\return none
```

The description tells you what the function does but does not give complete information on the configuration value that is required. To find that information, once you create a project in ModusToolbox IDE, you can highlight the function in the C code, right click, and select "Open Declaration" (F3). This will take you to the function declaration in the file *wiced\_hal\_gpio.h*. If you scroll to the top of this file, you will find a list of allowed choices. A subset of the choices is shown here:

```

/* Interrupt Enable
 * GPIO configuration bit 3, interrupt enable/disable defines
 */
GPIO_INTERRUPT_ENABLE_MASK = 0x0008, /**< GPIO configuration bit 3 mask */
GPIO_INTERRUPT_ENABLE      = 0x0008, /**< Interrupt Enabled */
GPIO_INTERRUPT_DISABLE     = 0x0000, /**< Interrupt Disabled */

/* Interrupt Config
 * GPIO configuration bit 0:3, Summary of Interrupt enabling type
 */
GPIO_EN_INT_MASK           = GPIO_EDGE_TRIGGER_MASK | GPIO_TRIGGER_POLARITY_MASK | GPIO_DUAL_EDGE_TRIGGER_MASK | GPIO_INTERRUPT_ENABLE_MASK,
GPIO_EN_INT_LEVEL_HIGH    = GPIO_INTERRUPT_ENABLE | GPIO_LEVEL_TRIGGER, /**< Interrupt on level HIGH */
GPIO_EN_INT_LEVEL_LOW     = GPIO_INTERRUPT_ENABLE | GPIO_LEVEL_TRIGGER | GPIO_TRIGGER_NEG, /**< Interrupt on level LOW */
GPIO_EN_INT_RISING_EDGE   = GPIO_INTERRUPT_ENABLE | GPIO_EDGE_TRIGGER, /**< Interrupt on rising edge */
GPIO_EN_INT_FALLING_EDGE  = GPIO_INTERRUPT_ENABLE | GPIO_EDGE_TRIGGER | GPIO_TRIGGER_NEG, /**< Interrupt on falling edge */
GPIO_EN_INT_BOTH_EDGE     = GPIO_INTERRUPT_ENABLE | GPIO_EDGE_TRIGGER | GPIO_EDGE_TRIGGER_BOTH, /**< Interrupt on both edges */

/* GPIO Output Buffer Control and Output Value Multiplexing Control
 * GPIO configuration bit 4:5, and 14 output enable control and
 * muxing control
 */
GPIO_INPUT_ENABLE          = 0x0000, /**< Input enable */
GPIO_OUTPUT_DISABLE       = 0x0000, /**< Output disable */
GPIO_OUTPUT_ENABLE        = 0x4000, /**< Output enable */
GPIO_KS_OUTPUT_ENABLE     = 0x0001, /**< Keyscan output enable*/
GPIO_OUTPUT_FN_SEL_MASK   = 0x0000, /**< Output function select mask*/
GPIO_OUTPUT_FN_SEL_SHIFT  = 0,

/* Global Input Disable
 * GPIO configuration bit 6, "Global_input_disable" Disable bit
 * This bit when set to "1", P0 input_disable signal will control
 * ALL GPIOs. Default value (after power up or a reset event) is "0".
 */
GPIO_GLOBAL_INPUT_ENABLE   = 0x0000, /**< Global input enable */
GPIO_GLOBAL_INPUT_DISABLE  = 0x0040, /**< Global input disable */

/* Pull-up/Pull-down
 * GPIO configuration bit 9 and bit 10, pull-up and pull-down enable
 * Default value is [0,0]--means no pull resistor.
 */
GPIO_PULL_UP_DOWN_NONE    = 0x0000, /**< No pull [0,0] */
GPIO_PULL_UP              = 0x0400, /**< Pull up [1,0] */
GPIO_PULL_DOWN            = 0x0200, /**< Pull down [0,1] */
GPIO_INPUT_DISABLE        = 0x0600, /**< Input disable [1,1] (input disabled the GPIO) */

```

For example:

An input pin with an active-low button would typically have the config set to:

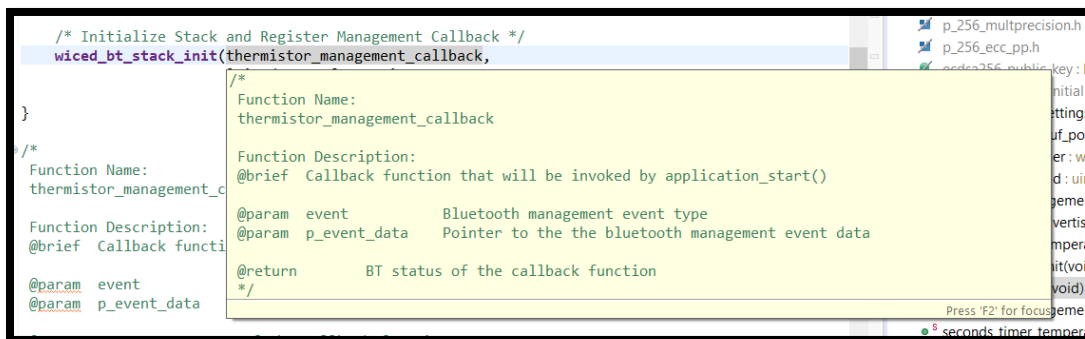
`GPIO_INPUT_ENABLE | GPIO_PULL_UP`

An output pin driving an active-low LED would typically have the config set to:

`GPIO_OUTPUT_ENABLE | GPIO_PULL_UP`

Note that right-clicking and selecting "Open Declaration" on function names and data-types inside ModusToolbox IDE is often very useful in finding information on how to use functions and what values are allowed for parameters. You can also just hover over a function name to get information like this:

You can also just hover over a function name and get information like this:



## 2.5 Peripherals

### 2.5.1 GPIO

As explained previously, GPIOs must be configured using the function `wiced_hal_gpio_configure_pin()`. The IOs on the kit that are connected to specific peripherals such as LEDs and buttons are usually configured for you as part of the platform files, so you don't need to configure them explicitly in your projects unless you want to change a setting (for example to enable an interrupt on a button pin).

Once configured, input pins can be read using `wiced_hal_gpio_get_pin_input_status()` and outputs can be driven using `wiced_hal_gpio_set_pin_output()`. You can also get the state that an output pin is set to (not necessarily the actual value on the pin) using `wiced_hal_gpio_get_pin_output()`. The parameter for these functions is the WICED pin name such as `WICED_P01` or a peripheral name from your BSP such as `WICED_GPIO_PIN_LED_1`.

For example:

```
btnState = wiced_hal_gpio_get_pin_input_status( WICED_GPIO_PIN_BUTTON_1 ); /* Get pin state */

wiced_hal_gpio_set_pin_output(WICED_GPIO_PIN_LED_2, 0); /* Set pin low */

wiced_hal_gpio_set_pin_output(WICED_GPIO_PIN_LED_2,
    !wiced_hal_gpio_get_pin_output(WICED_GPIO_PIN_LED_2) /* Invert desired pin state */
```

You will use code like this in exercise 1.

GPIO interrupts are enabled or disabled during pin configuration. For pins with interrupts enabled, the interrupt callback function (i.e. interrupt service routine or interrupt handler) is registered using `wiced_hal_gpio_register_pin_for_interrupt()`. For example, the following would enable a falling edge interrupt on `BUTTON1` with a callback function called `my_interrupt_callback`.

```
wiced_hal_gpio_register_pin_for_interrupt( WICED_GPIO_PIN_BUTTON_1,
    my_interrupt_callback, NULL);

wiced_hal_gpio_configure_pin( WICED_GPIO_PIN_BUTTON_1,
    ( GPIO_INPUT_ENABLE | GPIO_PULL_UP | GPIO_EN_INT_FALLING_EDGE ),
    GPIO_PIN_OUTPUT_HIGH );
```

The interrupt callback function is passed user data (optional) and the pin number. The callback function should clear the interrupt using `wiced_hal_gpio_clear_pin_interrupt_status(0)`. For example:

```
void gpio_interrupt_callback(void *data, uint8_t port_pin)
{
    /* Clear the gpio interrupt */
    wiced_hal_gpio_clear_pin_interrupt_status( WICED_PLATFORM_BUTTON_1 );

    /* Add other interrupt functionality here */
}
```

Note: The call to `wiced_hal_gpio_clear_pin_interrupt_status()` is shown in the code above for completeness. For most peripherals it is necessary to clear the interrupt in the callback function. However, for GPIOs this is done automatically before the callback is executed and so it is not strictly necessary.

## 2.5.2 Debug Printing

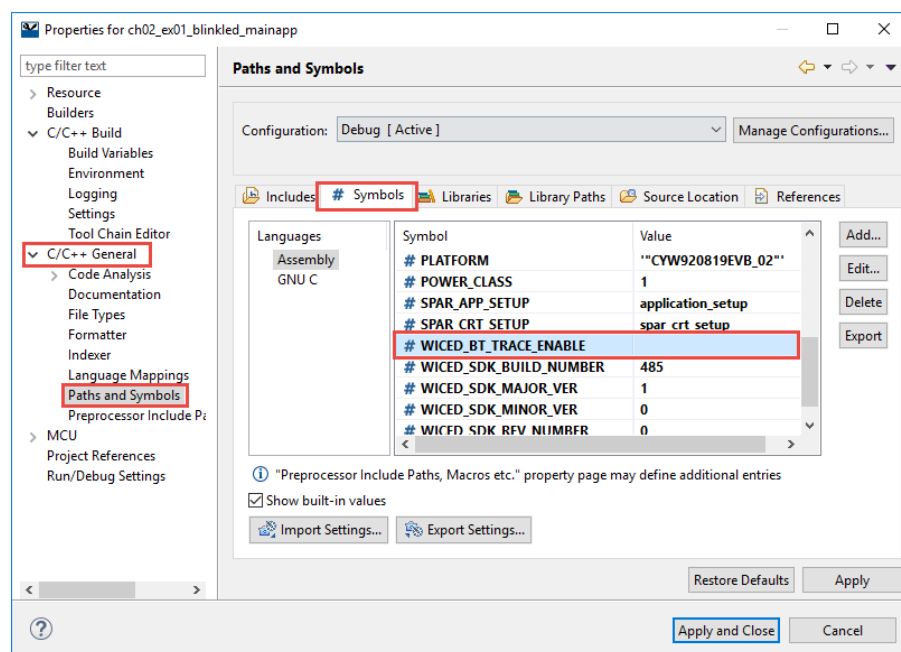
The kit has two separate UART interfaces –the HCI UART (Host Controller Interface UART) and the PUART (peripheral UART). The HCI UART interface is used for programming the kit and often is used by a host microcontroller to communicate with the BLE device. It will be discussed in more detail in a later chapter. The PUART is not used for any other specific functions so it is useful for general debug messages. When you plug a kit into your USB port both UART channels appear as COM ports. The PUART driver will always appear as the larger COM port number for a given kit. If you attach a kit to a Windows machine, the Device Manager will display entries that look something like this.



There are 3 things required to allow debug print messages:

1. Make sure that the symbol `WICED_BT_TRACE_ENABLE` is defined in the project Properties in the *Symbols* tab under *C/C++ General->Paths and Symbols*.

Note: to get to the window, first select the project that you want to modify in the project explorer, then either click "Project Build Settings" in the Quick Panel or use the menu item *Project->Properties*.



The provided starter templates all set this up automatically so editing the properties is not usually necessary.

2. Include the following header in the top-level C file:

```
#include "wiced_bt_trace.h"
```

3. Indicate which interface you want to use by choosing one of the following:

```
wiced_set_debug_uart( WICED_ROUTE_DEBUG_NONE);  
wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_PUART );  
wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_HCI_UART );  
wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_WICED_UART);
```

The last of these is used for sending formatted debug strings over the HCI interface specifically for use with the BTSpY application. The BTSpY application will be discussed in detail in the debugging chapter.

Once the appropriate debug UART is selected, messages can be sent using sprintf-type formatting using the WICED\_BT\_TRACE function. For example:

```
WICED_BT_TRACE( "Hello - this is a debug message \n\r");  
WICED_BT_TRACE("The value of X is: %d\n\r", x);
```

Note: this function does NOT support floating point values (i.e. %f).

You can easily print arrays using WICED\_BT\_TRACE\_ARRAY. The usage is:

```
WICED_BT_TRACE_ARRAY(arrayName, arrayLength, "String to be printed before the  
array data: ");
```

The WICED\_BT\_TRACE\_ARRAY function automatically adds a newline (\n) to the end of the printed string.

You will get a chance to try this in exercise 2.



### 2.5.3 PUART (Peripheral UART)

In addition to the debug printing functions, the PUART can also be used as a generic Tx/Rx UART block. To use it, first include the header file in your top level C file:

```
#include "wiced_hal_puart.h"
```

Next, initialize the block and setup the flow control and baud rate. For example:

```
wiced_hal_puart_init( );  
wiced_hal_puart_flow_off( );  
wiced_hal_puart_set_baudrate( 115200 );
```

For transmitting data, enable Tx, and then use the desired functions for sending strings (print), single bytes (write), or an array of bytes (synchronous\_write).

```
wiced_hal_puart_enable_tx( );  
wiced_hal_puart_print("Hello World!\n\r");  
/* Print value to the screen */  
wiced_hal_puart_print("Value = ");  
/* Add '0' to the value to get the ASCII equivalent of the number */  
wiced_hal_puart_write(value+'0');  
wiced_hal_puart_print("\r");
```

For receiving data, register an interrupt callback function, set the watermark to determine how many bytes should be received before an interrupt is triggered, and enable Rx.

```
wiced_hal_puart_register_interrupt(rx_interrupt_callback);  
/* Set watermark level to 1 to receive interrupt up on receiving each byte */  
wiced_hal_puart_set_watermark_level(1);  
wiced_hal_puart_enable_rx();
```

The Rx processing is done inside the interrupt callback function. You must clear the interrupt inside the callback function so that additional characters can be received.

```
void rx_interrupt_callback(void* unused)  
{  
    uint8_t readbyte;  
  
    /* Read one byte from the buffer and then clear the interrupt */  
    wiced_hal_puart_read( &readbyte );  
    wiced_hal_puart_reset_puart_interrupt();  
  
    /* Add your processing here */  
}
```

## 2.5.4 Timers

A timer allows you to schedule a function to run at a specified interval - e.g. send data every 10 seconds.

First, you initialize the timer using `wiced_init_timer`, you give it a pointer to a timer structure, specify the function want run, provide an argument to the function (or NULL if you don't need it), and the timer type. There are four types of timer. The first two are one-shot timers while the last two will repeat:

```
WICED_SECONDS_TIMER
WICED_MILLI_SECONDS_TIMER
WICED_SECONDS_PERIODIC_TIMER
WICED_MILLI_SECONDS_PERIODIC_TIMER
```

The function that you specify takes a single argument of `uint32_t arg`. If the function doesn't require any arguments you can specify 0 in the timer initialization function, but the function itself must still have the `uint32_t arg` argument in its definition.

Once you initialize the timer, you then start it using `wiced_start_timer`. This function takes a pointer to the timer structure and the actual time interval for the timer (either in seconds or milliseconds depending on the timer chosen).

Note that this is an interrupt function for when the timer expires rather than a continually executing thread so the function should NOT have a while(1) loop – it should just run and exit each time the timer calls it.

For example, to setup a timer that runs a function called `myTimer` every 100ms, you would do something like this:

```
wiced_timer_t my_timer_handle; /* Typically defined as a global */
.
.
.
/* Typically inside the BTM_ENABLED_EVT */
wiced_init_timer(&my_timer_handle, myTimer, 0,
WICED_MILLI_SECONDS_PERIODIC_TIMER);
wiced_start_timer(&my_timer_handle, 100);
.
.
.
/* The timer function */
void myTimer( uint32_t arg )
{
    /* Put timer code here */
}
```

## 2.5.5 PWM

There are 6 PWM blocks (PWM0 – PWM5) on the device each of which can be routed to any GPIO pin. The PWMs are 16 bits (i.e. they count from 0 to 0xFFFF).

The PWMs can use either the LHL\_CLK (which is 32 kHz) or PMU\_CLK (a.k.a ACLK1) which is configurable.

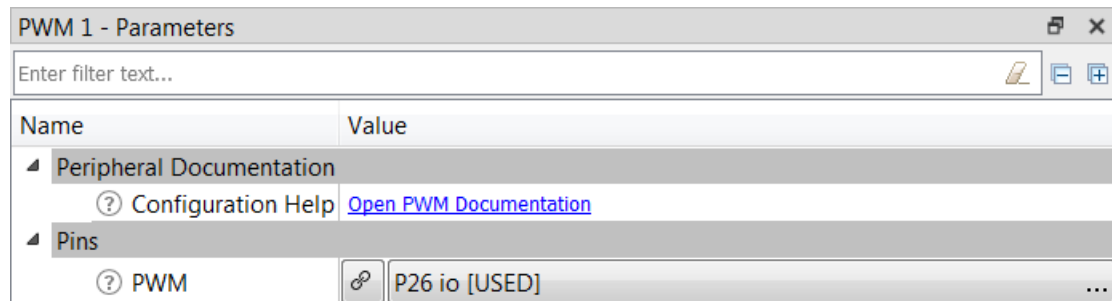
You must include the PWM header file to use the PWMs:

```
#include "wiced_hal_pwm.h"
```

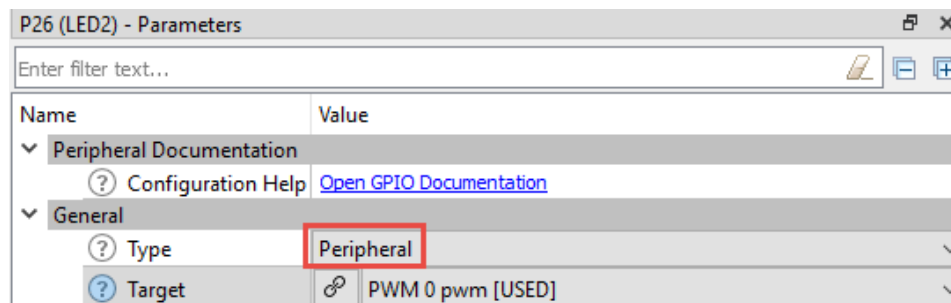
To initialize a PWM block you need to specify the PWM to be used and the pin to be connected to it. You can do this from your application code as follows:

```
wiced_hal_pwm_configure_pin( WICED_GPIO_PIN_LED_2 , PWM1 );
```

Alternatively, you can use the Device Configurator to enable the PWM and assign its output to one of the LEDs. With this approach there is no need to make the call to `wiced_hal_pwm_configure_pin()`.



You also need to change the pin configuration from LED to Peripheral:



Note: You can jump back and forth from the PWM to its associated pin using the button that looks like 2 links in a chain.

Once the configuration is done, call the start function like this:

```
wiced_hal_pwm_start( PWM1, LHL_CLK, toggleCount, initCount, 0 );
```

The `initCount` parameter is the value that the PWM will reset to each time it wraps around. For example, if you set `initCount` to `(0xFFFF - 99)` then the PWM will provide a period of 100 counts.

The toggleCount parameter is the value at which the PWM will switch its output from high to low. That is, it will be high when the count is less than the toggleCount and will be low when the count is greater than the toggleCount. For example, if you set the toggleCount to (0xFFFF-50) with the period set as above, then you will get a duty cycle of 50%.

You can invert the PWM output (i.e. it will start low and then transition high at the toggleCount) by setting the last parameter to 1 instead of 0.

If you want a specific clock frequency for the PWM, you must first configure the PMU\_CLK clock and then specify it in the PWM start function.

First, you have to include an additional header file:

```
#include "wiced_hal_aclk.h"
```

Then, if you (for example) want a 1 kHz clock for the PWM, you could do the following:

```
#define CLK_FREQ    (1000)

wiced_hal_aclk_enable(CLK_FREQ, ACLK1, ACLK_FREQ_24_MHZ );
wiced_hal_pwm_configure_pin (WICED_GPIO_PIN_LED_2, PWM1 );
wiced_hal_pwm_start(PWM0, PMU_CLK, toggleCount, initCount, 0);
```

Note that there is only 1 PMU clock available so if you use it, you will get the same clock frequency for all PWMs that use it as the source.

Also note that the call to wiced\_hal\_pwm\_configure\_pin is not necessary if the PWM was configured using the device configurator.

There are additional functions to enable, disable, change values while the PWM is running, get the init value, and get the toggle count. There is even a helper function called *wiced\_hal\_pwm\_params()* which will calculate the parameters you need given the clock frequency, the desired output frequency, and desired duty cycle. See the documentation for details on each of these functions.

## 2.5.6 NVRAM

There are many situations in a Bluetooth system where a non-volatile memory is required. One example of that is [Bonding](#) – which we will discuss in detail later - where you are required to save the [Link Keys](#) for future use. The BT\_20819 SDK provides an abstraction called the "NVRAM" (it is really just an area of flash memory that is set aside) for this purpose. The BSP typically allocates 4 kB to 8 kB for the NVRAM, but it is user-modifiable. The API and programming model remain the same regardless of the total NVRAM size.

The NVRAM is broken into multiple sections that are up to 512 bytes long (on the 20819 device), of which 500 are available for user data. To use the NVRAM, the application developer writes/reads to/from a number called the VSID (Virtual System Identifier). This number is not an offset or a block number within the memory. Rather, it is an ID that the API uses to locate the actual memory.

Physical addresses are not used because the NVRAM has a wear leveling scheme built in that moves the data around to avoid wearing out the memory. By using the VSID the user does not need to concern themselves with the (moving) physical location of their data. The only cost of this implementation is that reads and writes to NVRAM take a variable amount of time. Note that the scheme also has a "defragmentation" algorithm that runs during chip boot-up.

As the developer, you are responsible for managing what the VSIDs are used for in your application.

The API can be included in your project with `#include "wiced_hal_nvram.h"` which also `#defines` the first VSID to be `WICED_NVRAM_VSID_START` and last VSID to be `WICED_NVRAM_VSID_END`.

The write function for the NVRAM is:

```
uint16_t wiced_hal_write_nvram( uint16_t vs_id, uint16_t data_length, uint8_t *p_data,
                                wiced_result_t * p_status);
```

The return value is the number of bytes written. You need to pass a pointer to a `wiced_result` which will give you the success or failure of the write operation.

The read function for the NVRAM looks just like the write function:

```
uint16_t wiced_hal_read_nvram( uint16_t vs_id, uint16_t data_length, uint8_t * p_data,
                                wiced_result_t * p_status);
```

The return value is the number of bytes read into your buffer, and `p_status` tells you if the read succeeded.

Note that if you read from a VSID that has not been written to (since the device was programmed) the read will return with a failing error code. This is useful to determine if a device already has information (such as bonding information) stored.

### 2.5.7 I2C

There is an I2C master on the device which is routed by default to the Arduino header dedicated I2C pins. It is also connected to an LSM9DS1 motion sensor on the CYW920819EVB-02 kit.

#### Initialization

You must include the I2C header file to use the I2C functions:

```
#include "wiced_hal_i2c.h"
```

To initialize the I2C block you need to call the initialization function. If you want a speed other than the default of 100 kHz then you have to call the `set_speed` function after the block is initialized:

```
wiced_hal_i2c_init();  
wiced_hal_i2c_set_speed(I2CM_SPEED_400KHZ);
```

#### Read and Write Functions

There are two ways to read/write data from/to the slave. There is a dedicated read function called `wiced_hal_i2c_read()` and a dedicated write function called `wiced_hal_i2c_write()`. There is also a function called `wiced_hal_i2c_combined_read()` which will do a write followed by a read with a repeated start between them. These functions are all blocking.

The separate read/write functions require a pointer to the buffer to read/write, the number of bytes to read/write, and the 7-bit slave address. The LSM9DS1 is at address 0xD4 (write) / 0xD5 (read) and so, to generate the 7-bit address, shift 0xD4 right by 1 bit (0x6A).

For example, to write 2 bytes followed by a read of 10 bytes:

```
#define I2C_ADDRESS (0x6A)  
uint8_t TxData[2] = {0x55, 0xAA};  
uint8_t RxData[10];  
wiced_hal_i2c_write( TxData, sizeof(TxData), I2C_ADDRESS );  
wiced_hal_i2c_read( RxData, sizeof(RxData), I2C_ADDRESS );
```

If you need to write a value (e.g. a register offset value) followed by a read, you can use the `wiced_hal_i2c_combined_read()` function to do both in one function call. The function takes a pointer to the write data buffer, the number of bytes to write, a pointer to the read data buffer, the number of bytes to read, and finally, the 7-bit slave address.

For example, the same operation shown above could be:

```
#define I2C_ADDRESS (0x6A)  
uint8_t TxData[2] = {0x55, 0xAA};  
uint8_t RxData[10];  
wiced_hal_i2c_combined_read( TxData, sizeof(TxData), RxData, sizeof(RxData),  
I2C_ADDRESS );
```

## Read/Write Buffer

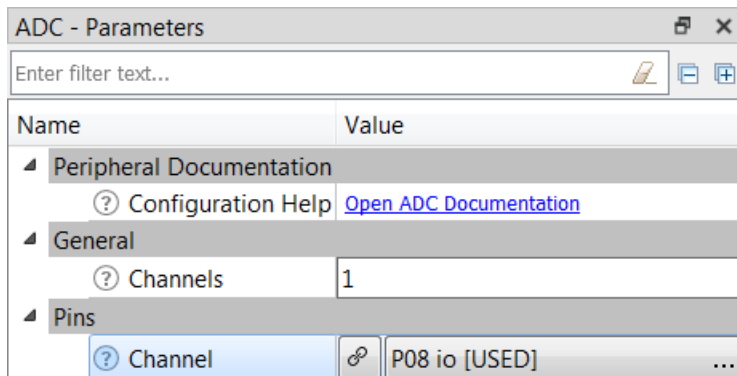
For the buffer containing the data that you want to read/write, you may want to setup a structure to map the I2C registers in the slave that you are addressing. In that case, if the structure elements are not all 32-bit quantities, you must use the packed attribute so that the non-32-bit quantities are not padded, which would lead to incorrect data. For example, if you have a structure with 3-axis 16-bit acceleration values, you could set up a buffer like this:

```
struct {  
    int16_t ax;  
    int16_t ay;  
    int16_t az;  
} __attribute__((packed)) accel_data;
```

There are two underscores before and after the word "attribute" and there are two sets of parentheses around the word "packed".

## 2.5.8 ADC

The device contains a 16-bit signed ADC (-32768 to +32767). The ADC has 32 input channels, all of which have fixed connections to pins or on-chip voltages such as Vddio. When setting up the ADC in the Device Configurator you choose the number of channels and assign each one to the desired physical pin (note that this is not routing signals on the device, merely assigning virtual channel numbers to the pins). Note that the voltage channels, like Vddio, are configured automatically for you.



You must include the ADC header file to use the ADC functions:

```
#include "wiced_hal_adc.h"
```

To initialize the ADC block you need to call the initialization function and select the input range based on your board supply. When you read a sample, you must specify which channel to read from. The BSP provides enums of the form `ADC_INPUT_*` for these channels in `wiced_hal_adc.h`. Examples of channel defines are `ADC_INPUT_P8` (ADC channel 9 is connected to pin P08 on the device) and `ADC_INPUT_VDDIO`.

There is one function that will return a count value and another function that will return a voltage value in millivolts. For example, to read the count and voltage from a sensor which is connected to GPIO WICED\_P10, you would do the following:

```
#define ADC_CHANNEL      (ADC_INPUT_P10)
wiced_hal_adc_init();
wiced_hal_adc_set_input_range( ADC_RANGE_0_3P6V );
raw_val = wiced_hal_adc_read_raw_sample( ADC_CHANNEL );
voltage_val = wiced_hal_adc_read_voltage( ADC_CHANNEL );
```



### 2.5.9 RTC (Real Time Clock)

The CYW20819 supports a 48-bit RTC timer referenced to a 32-kHz crystal (XTAL32K) LPO (low power oscillator). The LPO can be either external or internal. If an external LPO is not connected to the CYW20819, the firmware takes the clock input from the internal LPO for the RTC. The CYW20819 supports both 32-kHz and 128-kHz LPOs, but the internal defaults to 32-kHz.

The BT\_20819A1 SDK provides API functions to set the current time, get the current time, and convert the current time value to a string. By default, the date and time are set to January 1, 2010 with a time of 00:00:00 denoting HH:MM:SS.

You must include "wiced\_rtc.h" and the following code to initialize the RTC for use:

```
wiced_rtc_init();  
  
wiced_rtc_time_t newTime = { 0, 30, 12, 30, 11, 1999 }; // s,m,h,d,m,y  
  
rtc_setRTCTime( &newTime );
```

Note that the day and month are 0-based and so, for example, January is 0 and December is 11. Likewise, the first day of the month is 0.

After the RTC is initialized, you can use `wiced_rtc_get_time()` to get the time and `wiced_rtc_ctime()` to convert the result into a printable string.

## 2.6 WICED\_RESULT\_T

Throughout the SDK, a value from many of the functions is returned telling you what happened. The return value is of the type "wiced\_result\_t" which is a giant enumeration. If you right-click on wiced\_result\_t from a variable declaration, select "Open Declaration", and choose wiced\_result.h you will see this:

```
/** WICED result */
typedef enum
{
    WICED_RESULT_LIST(WICED_)
    BT_RESULT_LIST    ( WICED_BT_      ) /**< 8000 - 8999 */
} wiced_result_t;
```

To see standard return codes (WICED\_\*), right click and choose Open Declaration on WICED\_RESULT\_LIST. For Bluetooth specific return codes (WICED\_BT\_\*), right click and choose Open Declaration on BT\_RESULT\_LIST. The lists look like this:

**WICED\_\* :**

```
/** WICED result list */
#define WICED_RESULT_LIST( prefix ) \
    RESULT_ENUM( prefix, SUCCESS,          0x00 ), /**< Success */
    RESULT_ENUM( prefix, DELETED,          ,0x01 ), \
    RESULT_ENUM( prefix, NO_MEMORY,        ,0x10 ), \
    RESULT_ENUM( prefix, POOL_ERROR,        ,0x02 ), \
    RESULT_ENUM( prefix, PTR_ERROR,         ,0x03 ), \
    RESULT_ENUM( prefix, WAIT_ERROR,        ,0x04 ), \
    RESULT_ENUM( prefix, SIZE_ERROR,        ,0x05 ), \
    RESULT_ENUM( prefix, GROUP_ERROR,       ,0x06 ), \
    RESULT_ENUM( prefix, NO_EVENTS,         ,0x07 ), \
    RESULT_ENUM( prefix, OPTION_ERROR,      ,0x08 ), \
    RESULT_ENUM( prefix, QUEUE_ERROR,       ,0x09 ), \
    RESULT_ENUM( prefix, QUEUE_EMPTY,       ,0x0A ), \
    RESULT_ENUM( prefix, QUEUE_FULL,        ,0x0B ), \
    RESULT_ENUM( prefix, SEMAPHORE_ERROR,    ,0x0C ), \
    RESULT_ENUM( prefix, NO_INSTANCE,       ,0x0D ), \
    RESULT_ENUM( prefix, THREAD_ERROR,      ,0x0E ), \
    RESULT_ENUM( prefix, PRIORITY_ERROR,    ,0x0F ), \
    RESULT_ENUM( prefix, START_ERROR,       ,0x10 ), \
    RESULT_ENUM( prefix, DELETE_ERROR,      ,0x11 ), \
    RESULT_ENUM( prefix, RESUME_ERROR,       ,0x12 ), \
    RESULT_ENUM( prefix, CALLER_ERROR,      ,0x13 ), \
    RESULT_ENUM( prefix, SUSPEND_ERROR,     ,0x14 ), \
    RESULT_ENUM( prefix, TIMER_ERROR,       ,0x15 ), \
    RESULT_ENUM( prefix, TICK_ERROR,        ,0x16 ), \
```

## WICED\_BT\_\*:

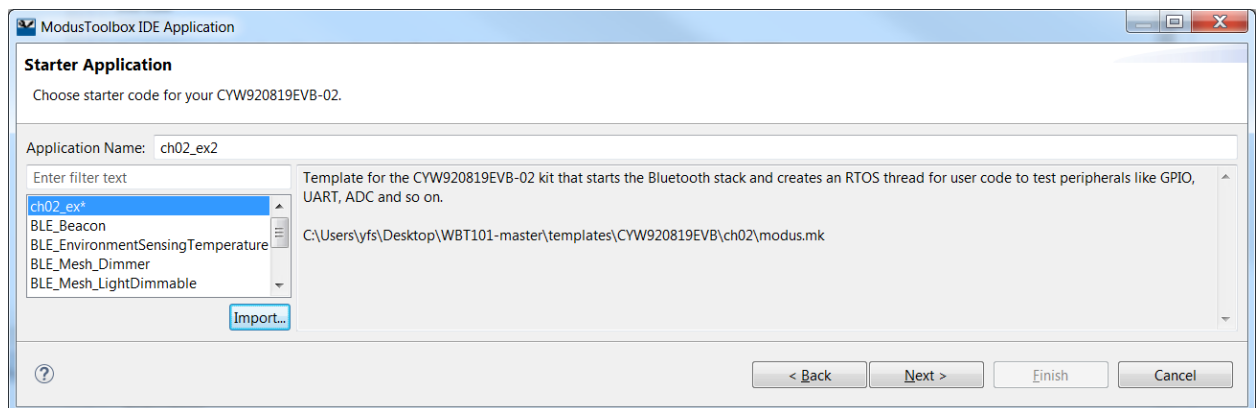
```
#define BT_RESULT_LIST( prefix ) \
    RESULT_ENUM( prefix, SUCCESS,                0 ), /**< Success */
    RESULT_ENUM( prefix, PARTIAL_RESULTS,        3 ), /**< Partial results */
    RESULT_ENUM( prefix, BADARG,                  5 ), /**< Bad Arguments */
    RESULT_ENUM( prefix, BADOPTION,               6 ), /**< Mode not supported */
    RESULT_ENUM( prefix, OUT_OF_HEAP_SPACE,       8 ), /**< Dynamic memory space exhausted */
    RESULT_ENUM( prefix, UNKNOWN_EVENT,          8029 ), /**< Unknown event is received */
    RESULT_ENUM( prefix, LIST_EMPTY,             8010 ), /**< List is empty */
    RESULT_ENUM( prefix, ITEM_NOT_IN_LIST,        8011 ), /**< Item not found in the list */
    RESULT_ENUM( prefix, PACKET_DATA_OVERFLOW,    8012 ), /**< Data overflow beyond the packet end */
    RESULT_ENUM( prefix, PACKET_POOL_EXHAUSTED,   8013 ), /**< All packets in the pool is in use */
    RESULT_ENUM( prefix, PACKET_POOL_FATAL_ERROR, 8014 ), /**< Packet pool fatal error such as per
    RESULT_ENUM( prefix, UNKNOWN_PACKET,          8015 ), /**< Unknown packet */
    RESULT_ENUM( prefix, PACKET_WRONG_OWNER,      8016 ), /**< Packet is owned by another entity */
    RESULT_ENUM( prefix, BUS_UNINITIALISED,       8017 ), /**< Bluetooth bus isn't initialised */
    RESULT_ENUM( prefix, MPAF_UNINITIALISED,      8018 ), /**< MPAF framework isn't initialised */
    RESULT_ENUM( prefix, RFCOMM_UNINITIALISED,    8019 ), /**< RFCOMM protocol isn't initialised */
    RESULT_ENUM( prefix, STACK_UNINITIALISED,     8020 ), /**< SmartBridge isn't initialised */
    RESULT_ENUM( prefix, SMARTBRIDGE_UNINITIALISED, 8021 ), /**< Bluetooth stack isn't initialised */
    RESULT_ENUM( prefix, ATT_CACHE_UNINITIALISED, 8022 ), /**< Attribute cache isn't initialised */
    RESULT_ENUM( prefix, MAX_CONNECTIONS_REACHED, 8023 ), /**< Maximum number of connections is re
    RESULT_ENUM( prefix, SOCKET_IN_USE,           8024 ), /**< Socket specified is in use */
    RESULT_ENUM( prefix, SOCKET_NOT_CONNECTED,    8025 ), /**< Socket is not connected or connecti
    RESULT_ENUM( prefix, ENCRYPTION_FAILED,       8026 ), /**< Encryption failed */
    RESULT_ENUM( prefix, SCAN_IN_PROGRESS,        8027 ), /**< Scan is in progress */
    RESULT_ENUM( prefix, CONNECT_IN_PROGRESS,     8028 ), /**< Connect is in progress */
    RESULT_ENUM( prefix, DISCONNECT_IN_PROGRESS,  8029 ), /**< Disconnect is in progress */
    RESULT_ENUM( prefix, DISCOVER_IN_PROGRESS,    8030 ), /**< Discovery is in progress */
    RESULT_ENUM( prefix, GATT_TIMEOUT,            8031 ), /**< GATT timeout occurred*/
```

## 2.7 Exercises

### Exercise - 2.1 (GPIO) Blink an LED

In this exercise, you will blink LED\_2 on the kit at 2 Hz.

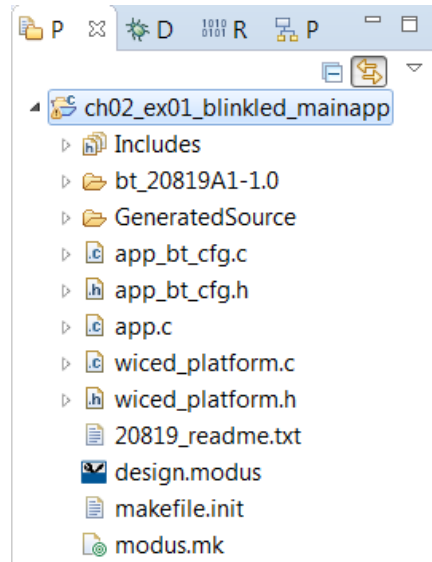
1. Launch ModusToolbox IDE and create a new workspace.
2. In the Quick Panel click "New Application".
3. Select the CYW920819EVB-02 kit and press "Next >".
4. Browse the available starter applications (although we are not going to use any of them just yet).
5. Press the "Import..." button and navigate to the course materials "templates/CYW920819EVB/ch02" folder.
6. Double-click on the modus.mk file to import the template and return to the wizard.
7. The new template will now be selected, but you may need to wait a few seconds for the dialog to refresh.



8. Change the application name to **ch02\_ex01\_blinkled**.

Note: you can call the application anything you like but it will really help if you maintain an alphabetically sortable naming scheme – you are going to create quite a few projects in this course.

9. Click "Next ". Verify the presented device, board, and example, then click "Finish". When you finish, the Project Explorer window in Eclipse should look like the following screenshot (ModusToolbox IDE adds "\_mainapp" to the project name because some applications contain more than one project):



10. Examine `app.c` to make sure you understand what it does.

All WICED BLE applications are multi-threaded (the BLE stack requires it). There is an operating system (RTOS) that gets launched from the device startup code and you can use it to create your own threads. Each thread has a function that runs almost as though it is the only software in the system – the RTOS allocates time for all threads to execute when they need to. This makes it easier to write your programs without a lot of extra code in your main loop. The details of how to use the RTOS effectively are covered in the next chapter but, in these exercises, we will show you how to create a thread and associate it with a function for the code you will write (look in `app_bt_management_callback()`).

11. Add code in the `app_task` thread function to do the following:

- a. Read the state of `WICED_GPIO_PIN_LED_1`
  - i. Hint: Go back to the section on GPIOs if you need a reminder on using pins.
- b. Drive the state of `WICED_GPIO_PIN_LED_1` to the opposite value.

12. In the Quick Panel, look in the "Launches" section and press the "ch02\_ex01\_blinkled Build + Program" link.

## Questions

1. What is the name of the first user application function that is executed? What does it do?
2. What is the purpose of the function `app_bt_management_callback`? When does the `BTM_ENABLED_EVT` case occur?
3. What controls the rate of the LED blinking?

## Exercise - 2.2 (GPIO) Add Debug Printing to the LED Blink Project

For this exercise, you will add a message that is printed to a UART terminal each time the LED changes state.

1. Repeat steps 2 through 8 from the previous exercise to create a new application called **ch02\_ex02\_blinkled\_print**.

Note: It is possible to copy-paste the prior project and rename it. The new application will build without error. However, Eclipse does not re-create programming or debugging configurations in a copied project and so you would need to manually configure a GDB connection to an OpenOCD server. Accordingly, we do not recommend copying whole projects unless you are an Eclipse and OpenOCD expert who can set up debugging configurations faster than it takes to create a new project!

2. Copy the app.c file from the previous application.

Note: The safe and quick way to do this is to highlight the file you want in the Eclipse Project Explorer, copy (Control-C), select the newly created application, and paste (Control-V). Eclipse will ask if you wish to overwrite the file (if it does not, you are probably copying to the wrong location). You can also use drag & drop but be sure to press and hold the control key so that you copy, not move, the file.

3. Add WICED\_BT\_TRACE calls to display "LED LOW" and "LED HIGH" at the appropriate times.
  - a. Hint: Go back to the section on Debug Printing if you need a refresher.
  - b. Hint: Remember to set the debug UART to WICED\_ROUTE\_DEBUG\_TO\_PUART.  
Although you will see comments in the template code encouraging you to put initialization code in the app\_bt\_management\_callback() function so that it runs when the BLE stack starts up, we recommend doing it in application\_start() instead. This is because the PUART is a special type of peripheral and you may want to print messages before even trying to start the stack!
  - c. Hint: Remember to use \n\r to create a new line so that information is printed on a new line each time the LED changes.
4. Program your application to the board.
5. Open a terminal window (e.g. PuTTY or TeraTerm) with a baud rate of 115200 and observe the messages being printed.
  - a. Hint: The PUART will be the larger number of the two WICED COM ports.
  - b. 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:
    - i. Go to the Serial tab, select the correct COM port (you can get this from the device manager under "Ports (COM & LPT)" as "WICED USB Serial Port"), and set the speed to 115200.
    - ii. Go to the session tab, select the Serial button, and click on "Open".
    - iii. If you want an automatic carriage return with a line feed in putty (i.e. add a \r for every \n) check the box next to "Terminal -> Implicit CR in every LF"

### Exercise - 2.3 (GPIO) Read the State of a Mechanical Button

In this exercise, you will control an LED by monitoring the state of a mechanical button on the kit.

1. Create another new application called **ch02\_ex03\_button** with the same template.
2. Again, copy your code that toggles the LED from the **ch02\_ex01\_blink** application
3. In the C file:
  - a. Change the thread sleep time to 100ms.
  - b. In the thread function, check the state of mechanical button input (use `WICED_GPIO_PIN_BUTTON_1`). Turn on `WICED_GPIO_PIN_LED_1` if the button is pressed and turn it off if the button is not pressed.
4. Program your project to the board and test it.

### Exercise - 2.4 (GPIO) Use an Interrupt to Toggle the State of an LED

In this exercise, rather than polling the state of the button in a thread, you will use an interrupt so that your firmware is notified every time the button is pressed. In the interrupt callback function, you will toggle the state of the LED.

1. Create another new application called **ch02\_ex04\_interrupt** with the same template. There is no need to copy source code from another application in this exercise.
2. In the app.c file:
  - a. Remove the calls to `wiced_rtos_create_thread()` and `wiced_rtos_init_thread()`.
  - b. Delete the thread function.
2. In the `BTM_ENABLED_EVT`, set up a falling edge interrupt for the GPIO connected to the button and register the callback function.
  - a. Hint: You will need to call `wiced_hal_gpio_register_pin_for_interrupt` and `wiced_hal_gpio_configure_pin`.
3. Create the interrupt callback function so that it toggles the state of the LED each time the button is pressed.
4. Program your application to the board and test it.

### Exercise - 2.5 (Timer) Use a Timer to Toggle an LED

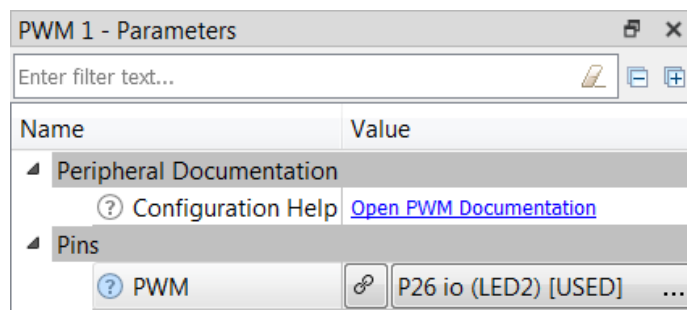
In this exercise, you will replace the thread and instead use a timer to blink an LED.

1. Create another new application called **ch02\_ex05\_timer** with the same template.
2. Copy app.c from **ch02\_ex04\_interrupt**.
3. In the C file:
  - a. Add an include for `wiced_timer.h`.
  - b. In the `BTM_ENABLED_EVT`, add calls to initialize and start a periodic timer with a 250ms interval.
  - c. Modify the interrupt callback function to serve as the timer callback.
    - i. Hint: the body of the timer function is the same as the code you wrote for the interrupt callback, but the function argument list is slightly different.
4. Program your application to the board and test it.

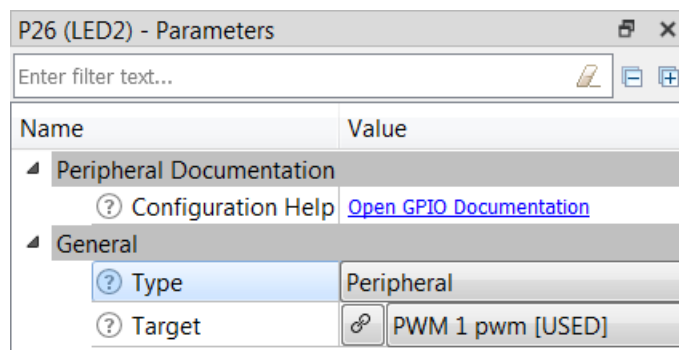
## Exercise - 2.6 (PWM) LED brightness

In this exercise, you will control an LED using a PWM instead of a GPIO. The PWM will toggle the LED too fast for the eye to see, but by controlling the duty cycle you will vary the apparent brightness of the LED.

1. Create a new application called **ch02\_ex06\_pwm** using the modus.mk file in the original location of templates/CYW920819EVB/ch02.
2. Click Configure Device from the Quick Panel or double-click the design.modus file to open it in the Device Configurator.
  - a. Find PWM0 in the list of peripherals and click the checkbox to enable it.
  - b. In the Parameters panel use the PWM drop-down menu to choose P26 (LED2) for the output.
  - c. Find P26 (LED2) in the list of peripherals.
  - d. Hint: the PWM Parameter panel should look like this. You can jump straight to the Parameters for P26 by clicking the "link" button to the left of the drop-down menu.



- e. Change the type of the pin from LED to Peripheral.
- f. Hint: The Pin Parameter panel should look like this. Note that the PWM output is already selected.



- g. Save the changes and close the configurator.
3. In the app.c file:
  - a. Add a #include for the PWM functions - "wiced\_hal\_pwm.h"
  - b. Configure PWM0 with an initial period of 100 and a duty cycle of 50%.
  - c. Hint: Use LHL\_CLK as the source clock since the exact period of the PWM doesn't matter as long as it is faster than the human eye can see (~50 Hz).



4. Update the duty cycle in the thread function so that the LED gradually cycles through intensity values from 0 to 100%.
  - a. Hint: Change the delay in the thread function to 10ms so that the brightness changes relatively quickly.
5. Program the application to the board and test it.

### Exercise - 2.7 (PWM) LED toggling at specific frequency and duty cycle

In this exercise, you will use a PWM with a period of 1 second and a duty cycle of 10% so that the LED will blink at a 1 Hz rate but will only be on for 100ms each second. In the prior exercise you configured the PWM and then the pin. This time do it by configuring the pin first.

1. Create a new application called **ch02\_ex07\_pwm\_blink** using the modus.mk file in templates/CYW920819EVB/ch02.
2. Click Configure Device from the Quick Panel or double-click the design.modus file to open it in the Device Configurator.
  - a. Find P27 in the list of Pins and change its type to Peripheral.
  - b. Choose PWM1 from the Target drop-down menu
  - c. Click the link button to jump to the PWM – notice that it is already enabled and configured.
  - d. Save the file and close the Device Configurator.
3. In the app.c file:
  - a. Hint: Don't forget to include the header files for the PWM and ACLK functions.
  - b. Initialize the aclk with a frequency of 1 kHz.
  - c. Change the PWM1 configuration to use PMU\_CLK as the source. Set the period to 1000 and set the duty cycle to 10%.
4. As you did in ex05, remove or comment out the calls to `wiced_rtos_create_thread()`, `wiced_rtos_init_thread()` and the thread function because, even though it would not affect the program functionally, it is not a good practice to write to pins that are driven by a peripheral. Program the application to the board and test it.

## Exercise - 2.8 (I2C) Read Motion Sensor Data

In this exercise, you will use the I2C master to read 3-axis acceleration data from the LSM9DS1 motion sensor that is included on the kit. The values will be printed to the UART.

1. Create a new application called **ch02\_ex08\_i2c\_motion** using the modus.mk file in templates/CYW920819EVB/ch02\_ex08\_i2c\_motion.
2. In app.c, look for TODO comments to configure the motion sensor and read values using I2C.
  - a. Hint: If you use the `wiced_hal_i2c_combined_read()` function, which writes an offset and then reads data from that location, it expects the read arguments first and the wrote arguments second.
3. The LSM9DS1 motion sensor on the kit has the following properties:
  - a. 0x6A: device I2C address
  - b. 0x20: address of configuration register to enable the accelerometer
  - c. 0x40: value for configuration register to provide 2g acceleration data at 50Hz
  - d. 0x28: address of first acceleration data register. The values in order are:
    - i. 0x28: X\_LSB
    - ii. 0x29: X\_MSB
    - iii. 0x2A Y\_LSB
    - iv. 0x2B Y\_MSB
    - v. 0x2C: Z\_LSB
    - vi. 0x2D: Z\_MSB
  - e. Hint: Search online for the LSM9DS1 datasheet if you want to explore other settings.
4. Program your application to the board and test it.

## Exercise - 2.9 (Advanced) (NVRAM) Write and Read Data in the NVRAM

In this exercise, you will store a 1-byte value in the NVRAM. Mechanical button MB1 (WICED\_GPIO\_PIN\_BUTTON\_1) will increment the value each time it is pressed. Independently, the value stored in the NVRAM will be printed to a terminal window once every second.

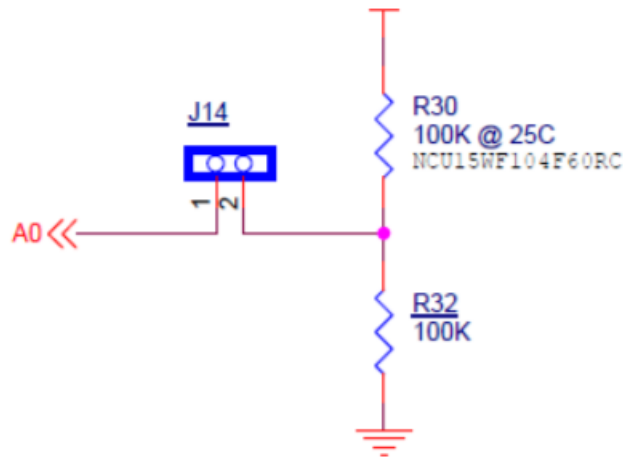
1. Create another new application called **ch02\_ex09\_nvram** using the same template from the original location of templates/CYW920819EVB/ch02 (NOT the I2C template).
2. Add a #include for the NVRAM API functions.
3. Call `wiced_set_debug_uart()` to use the PUART.
4. In the thread, instead of blinking the LED, do the following:
  - a. Read 1-byte from the NVRAM location `WICED_NVRAM_VSID_START` and save the value to a `uint8_t` variable.
  - b. Print the value to the terminal along with the number of bytes read and the status of the read operation.
  - c. Change the thread delay to 1000 ms.
5. Setup `WICED_GPIO_PIN_BUTTON_1` for a falling edge interrupt.
  - a. Hint: look at the interrupt exercise if you need a refresher on configuring the pin interrupt.
6. In the interrupt callback function:
  - a. Copy the code you wrote in the thread function to read the stored value.
  - b. Increment the value.
  - c. Write the new value to the NVRAM in location `WICED_NVRAM_VSID_START`.
  - d. Print out the number of bytes written and the status of the write operation to the terminal.
7. Open a terminal window and program the kit. Wait a few seconds and then press Button 1 a few times to observe the results.
8. Reset the kit. Notice that the previously stored value is retained.
9. Unplug the kit, plug it back in, and reset the terminal. Notice that the previously stored value is retained.

## Questions

1. How many bytes does the NVRAM read function get before you press the button the first time?
2. What is the return status value before you press the button the first time?
3. What does the return value mean?

## Exercise - 2.10 (Advanced) (ADC) Calculate the resistance of a thermistor

In this exercise you will measure the voltages, in millivolts, of Vddio (supply) and across the thermistor balance resistor (100kOhm). Then you will use that data to calculate the resistance of the thermistor. For reference, here is the thermistor circuit.



The thermistor shares pin P08 (CYW20819EVB) or P10 (CYBT213043-MESH) with A0 on the Arduino header. You need to make sure the THERMISTOR\_ENABLE jumper is attached to read the voltage.

1. Create a new application called **ch02\_ex10\_adc** using the modus.mk file in templates/CYW20819EVB/ch02.
2. Use the Configurator to enable the ADC and connect channel 1 to P08 (CYW20819EVB) or P10 (CYBT213043-MESH). Save your selections and close the Configurator.

**WARNING:** The CYW20819EVB kit BSP only supports 8 GPIOs and adding the ADC channel causes that limit to be exceeded. Since we are not using SPI in any of these exercises disable the SPI peripheral and all four of its connected pins to avoid getting the following error:

```
../GeneratedSource/cycfg_pins.c:60:3: error: 'PLATFORM_GPIO_11' undeclared here (not in a function); did you mean 'PLATFORM_GPIO_10'?
```

3. Add in the include file for the ADC functions.
4. Add the call to `wiced_set_debug_uart()` to enable printing to the PUART.
5. In the management callback function, initialize the ADC.
6. In the application thread function, use the ADC to read Vddio and the voltages across the balance resistor.
  - a. Hint: The ADC has a dedicated channel to read the supply voltage called `ADC_INPUT_VDDIO`.
7. Calculate and print the resistance of the thermistor periodically.
  - a. Hint: The formula is as follows.

$$R_{\text{therm}} = \frac{(V_{\text{ddio}} - V_{\text{meas}}) * \text{BALANCE\_RESISTANCE}}{V_{\text{meas}}}$$

8. Program the board and open a terminal window with a baud rate of 115200. Alternately place and remove your finger from the thermistor (next to the THERMISTOR\_ENABLE jumper) and observe the value displayed in the terminal.

### Exercise - 2.11 (Advanced) (UART) Send a value using the standard UART functions

In this exercise, you will use the standard UART functions to send a value to a terminal window. The value will increment each time a mechanical button on the kit is pressed.

1. Create a new application called **ch02\_ex11\_uartsend** using the modus.mk file in templates/CYW920819EVB/ch02.
2. Copy the app.c file from the **ch02\_ex04\_interrupt** application. Remember to copy, not move!
3. In app.c, remove the call to wiced\_set\_debug\_uart() if your interrupt exercise used the PUART.
4. Initialize the UART with Tx enabled, baud rate of 115200, and no flow control.
5. Modify the interrupt callback so that each time the button is pressed a variable is incremented and the value is sent out over the UART. 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.
  - a. Hint: Use a "static" variable in the callback function to remember the last number printed.
6. Program the board and open a terminal window with a baud rate of 115200. Press the button and observe the value displayed in the terminal.

### Exercise - 2.12 (Advanced) (UART) Get a value using the standard UART functions

In this exercise, you will learn how to read a value from the UART rather than sending a value like in the previous exercise. The value entered will be used to control an LED on the board (0 = OFF, 1 = ON).

1. Create a new application called **ch02\_ex112\_uartreceive** using the modus.mk file in templates/CYW920819EVB/ch02.
2. Copy the app.c file from the **ch02\_ex11\_uartsend** project. Remember to copy, not move!
3. Update the code to initialize the UART with Rx enabled, baud rate of 115200, no flow control, and an interrupt generated on every byte received.
  - a. Hint: you can remove the code for the button press and its interrupt, but you will need to register a UART Rx interrupt callback instead.
4. In the interrupt callback, read the byte. If the byte is a 1, turn on LED\_2. If the byte is a 0, turn off the LED. If you wish you can also print LED\_ON/OFF messages. Ignore any other characters.
5. Program your application to the board.
6. Open a terminal window with a baud rate of 115200.
7. Press the 1 and 0 keys on the keyboard and observe the LED turn on/off.

### Exercise - 2.13 (Advanced) (RTC) Display Time and Date Data on the UART

In this exercise you will set the time and date after reset and thereafter display a clock on the UART.

1. Create a new application called **ch02\_ex13\_rtc** using the modus.mk file in templates/CYW920819EVB/ch02\_ex13\_rtc.
2. In app.c, look for TODO comments and add code to control the RTC as follows.
  - a. Initialize the RTC.
  - b. Set the time and date. Make sure you understand how the `getDateTimeEntry()` function works and gets used.

Hint: the template code includes `ring_pop()` and `ring_push()` functions that implement a ring buffer for the UART – the UART interrupt code pushes received characters into the buffer and the code that sets the time and date pulls characters from it.

- c. Display the clock with a precision of 1s.
3. Program your application to the board and test it.