Chapter 1: Tour of Cypress Bluetooth

After completing Chapter 1 (this chapter) you will understand a top-level view of the ModusToolbox Bluetooth ecosystem components, including the chips, modules, software, documentation, support infrastructure and development kits. You will have ModusToolbox installed and working on your computer and understand how to program an existing project into a kit.

1.1   Embedded vs. Hosted Bluetooth

Many of our Bluetooth chips can be used in either embedded or hosted mode. Embedded mode means that the Bluetooth chip runs the entire Bluetooth stack. In hosted mode, the Bluetooth chip runs the lower parts of the stack while a separate host processor runs the upper parts of the stack. The two devices communicate using the Host Controller Interface (HCI). We will talk about this in much more detail later.

Most of this class focuses on embedded mode using a CYW920819 Bluetooth chip – sometimes referred to as a WICED Bluetooth chip (WICED = Wireless Internet Connectivity for Embedded Devices). We will also briefly cover using a CYW43012 chip in hosted mode with a PSoC 6 MCU as the host. The libraries, application structure and configurators are different between the two, but the API functions are identical in most cases. All descriptions of libraries, application structure and configurators in this class refer to the CYW920819 in embedded mode unless otherwise noted. The ModusToolbox 101 class covers the CYW43012 in hosted mode with a PSoC 6 MCU host in more detail.

1.2   Tour of ModusToolbox

ModusToolbox is a set of tools that allow you to develop, program, and debug applications for Cypress MCUs.

The Eclipse IDE for ModusToolbox is an Eclipse-based development environment that is included as part of the ModusToolbox software installer, but it is not the only supported environment. A command line interface (CLI) can be used interchangeably with IDE applications. In addition, users can use ModusToolbox software with their own preferred IDE, such as Visual Studio (VS) Code or IAR.

For Windows, ModusToolbox software is installed, by default, in *C:/Users/<UserName>/ModusToolbox.* Once installed, the IDE will show up in the Windows Start menu and Search feature..

Note This class focusses on Bluetooth. We will mainly use the Eclipse IDE for ModusToolbox with a little information on using the CLI. You'll also get a chance to try using VS Code with ModusToolbox Bluetooth applications in an advanced exercise if you want. Refer to the ModusToolbox 101 class for many more details, tips, and tricks for working with ModusToolbox including using other IDEs.

1.2.1 Make/Build Infrastructure

ModusToolbox uses a GNU "make" based system to create, build, program, and debug projects. The installation provides a set of tools required for all the operations to work. Those tools include:

- Gnu Make 3.81 or newer
- Git 2.20 or newer
- Python3
- GCC
- OpenOCD with supports for Cypress devices
- Configurator tools
- Library Manager
- Project Creator GUI and CLI utilities
- Modus-Shell
- Eclipse IDE for ModusToolbox

These tools can be found in the ModusToolbox installation directory under *tools_<version>,* except for the Eclipse IDE which is under *ide_<version>*.

The general application development flow is as follows. All these steps can be done using an IDE or on the command line. You can even switch back and forth seamlessly between the two.
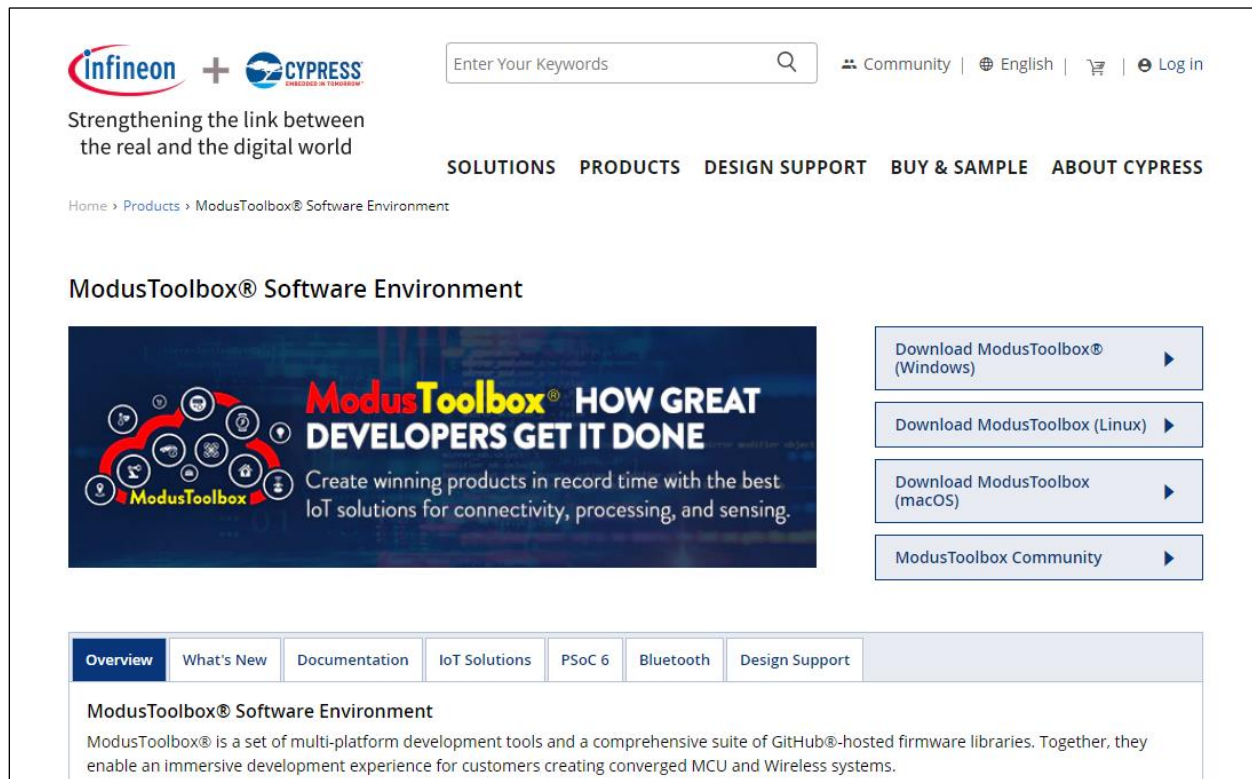
- Create the application using the Project Creator tool (this step downloads the application itself and any require libraries)
- Configure the device and its peripherals using configurators
- Write the application code
- Build/Program/Debug

1.3　ModusToolbox Online

1.3.1　Cypress.com

On the Cypress website, you can download the software, view the documentation, and access the solutions:

https://www.cypress.com/products/modustoolbox-software-environment

1.3.2 Community Page

On the ModusToolbox Community website, you can interact with other developers and access various Knowledge Base Articles (KBAs):

https://community.cypress.com/t5/ModusToolbox/ct-p/ModusToolbox
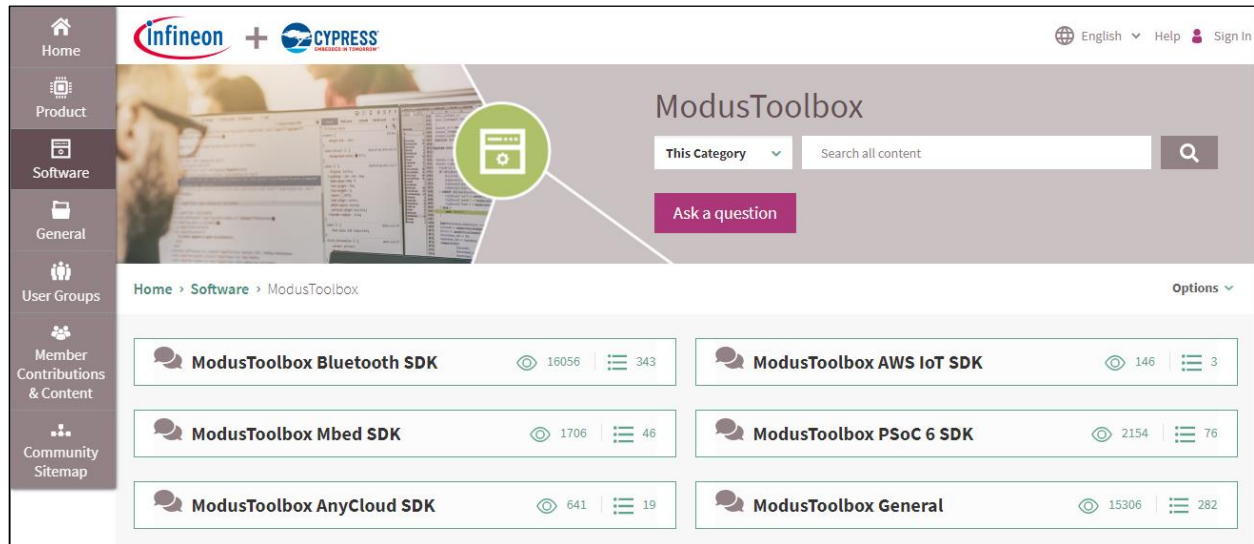
1.3.3 Github.com

The Cypress GitHub website contains all the Board Support Packages (BSPs), code examples, and libraries for use with various ModusToolbox tools.

https://github.com/cypresssemiconductorco
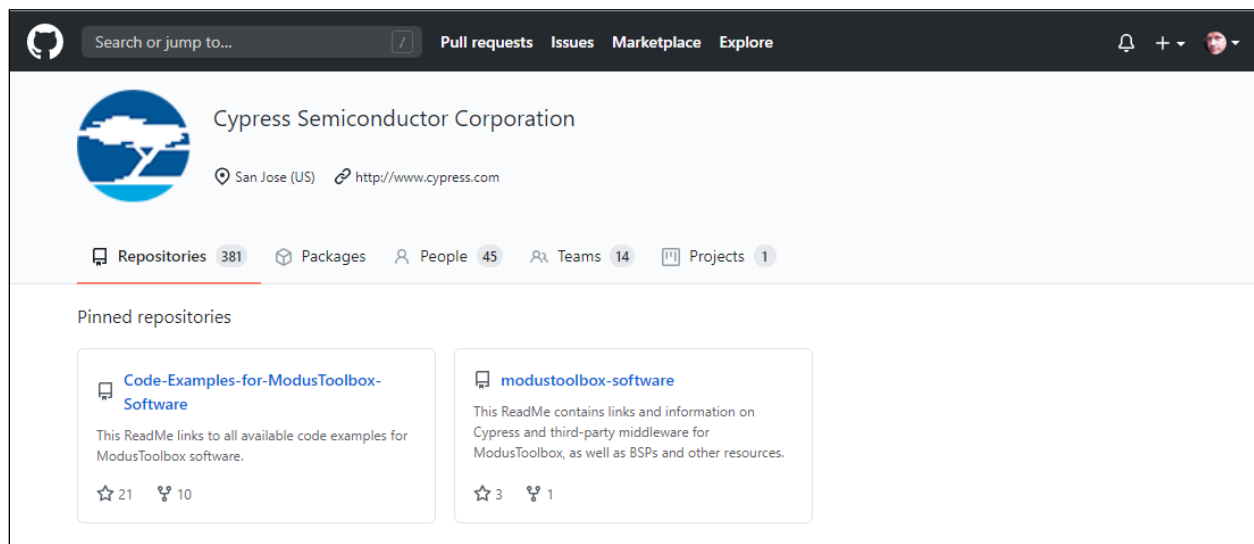
1.4   Network Considerations

The Project Creator and Library Manager tools both require internet access. Specifically, GitHub.com. Depending on your network and location, you may need to change proxy settings or otherwise work around access limitations.

1.4.1 Network Proxy Settings

When you run the Project Creator or Library Manager, the first thing it does is look for a remote manifest file. If that file isn't found, you will not be able to go forward. In some cases, it may find the manifest but then fail during the project creation step (during git clone). If either of those errors occur, it may due to one of these reasons:

- You are not connected to the internet. In this case, you can choose to use offline content if you have previously downloaded it. Offline content is discussed in the next section.
- You may have incorrect proxy settings (or no proxy settings).

To view/set proxy settings in the Project Creator or Library Manager, use the menu option **Settings > Proxy Settings…**

If your network doesn't require a proxy, choose "Direct". If your network requires a proxy choose "Manual". Enter the server name and port in the format http:/hostname:port/.
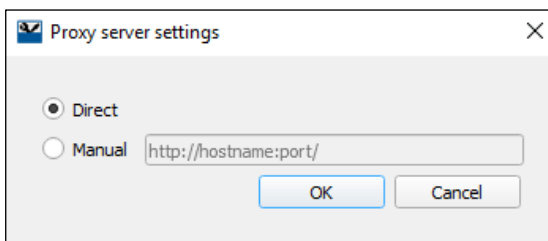
No Proxy   Manual Proxy
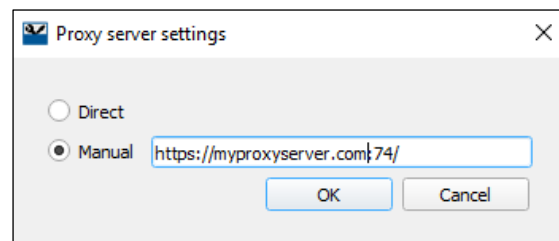
Once you set a proxy server, you can enable/disable it just by selecting Manual or Direct. There is no need to re-enter the proxy server every time you connect to a network requiring that proxy server. The tool will remember your last server name.

The settings entered in either the Project Creator or Library Manager apply to the other.

1.4.2 Offloading Manifest Files

In some locations, git clone operations from GitHub may be allowed but raw file access may be restricted. This prevents manifest files from being loaded. In that case, the manifest files can be offloaded either to an alternate server or a local location on disk. For details on how to do this, see the following knowledge base article:

https://community.cypress.com/t5/Knowledge-Base-Articles/Offloading-the-Manifest-Files-of-ModusToolbox-KBA230953/ta-p/252973

1.4.3 Offline Content

In the case where network access is just not possible, Infineon provides a zipped-up bundle with all of our GitHub repos to allow you to work offline, such as on an airplane or if for some reason you don't have access to GitHub. To set this up, you must have access to cypress.com in order to download the zip file. After that, you can work offline.

Go to https://community.cypress.com/t5/Resource-Library/ModusToolbox-offline-libraries/ta-p/252265 and follow the instructions to download and extract the offline content.

To use the offline content, toggle the setting in the Project Creator and Library Manager tools, as follows:
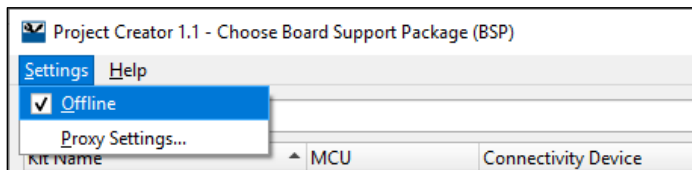
If you try to open these tools without a connection, a message will ask you to switch to Offline Mode.
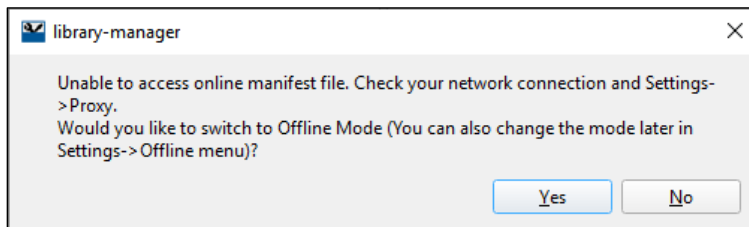
1.5   Eclipse IDE for ModusToolbox

1.5.1 First Look

ModusToolbox software includes an Eclipse-based IDE that supports Windows, MacOS, and Linux. On Windows, it is installed, by default, in *C:/Users/<UserName>/ModusToolbox.*

Once installed, Windows users can access the Eclipse IDE via the Start Menu. When you open the IDE, you will be asked for which workspace to use. The default Workspace location is *C:/Users/<UserName>/mtw*, but you can have as many Workspaces as you want, and you can put them anywhere you want. I usually put each workspace under a directory inside *C:/Users/<UserName>/mtw* such as *wbt101* for the exercises in this class.

Each time you open Eclipse IDE you will need to provide a workspace name such as the following:
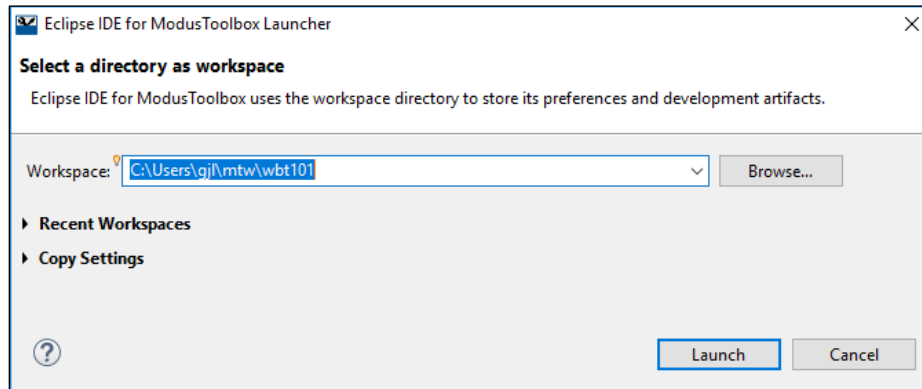
After clicking **Launch** for a new Workspace, the IDE will start with an empty ModusToolbox perspective. If you open an existing Workspace, the IDE will open to where you exited that Workspace.
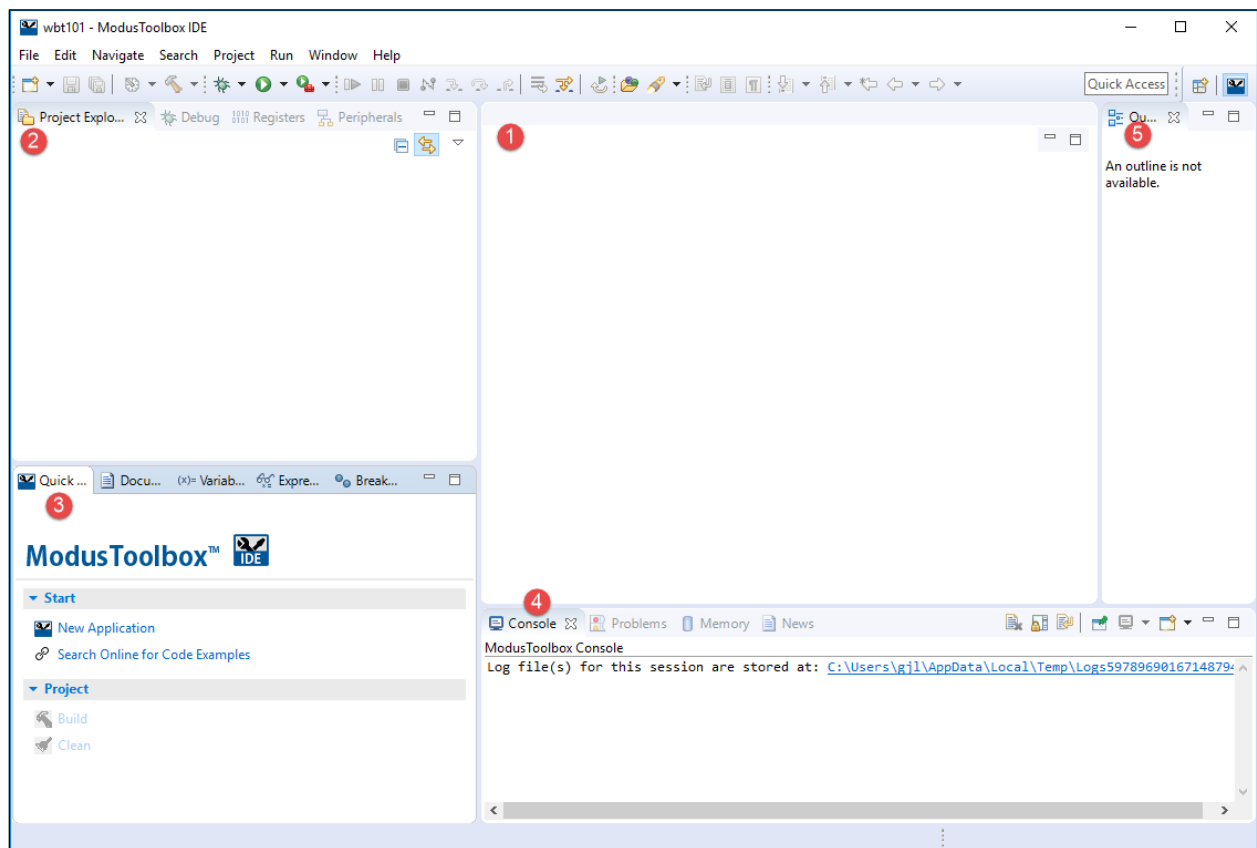
Whenever you want to switch to a different Workspace or create a new one, just use the menu item **File > Switch Workspace** and either select one from the list or select **Other…** to specify the name of an existing or new Workspace.

A perspective in Eclipse is a collection of views. The ModusToolbox perspective combines editing and debugging features. You can also create your own custom perspectives if you want a different set or arrangement of windows.

You can always get back to the ModusToolbox perspective by selecting it from the button in the upper right corner of the IDE, clicking the **Open Perspective** button and choosing **ModusToolbox**, or from **Window > Perspective > Open Perspective > Other > ModusToolbox**.

The major views of the ModusToolbox perspective are:

1. File Viewer/Editor
2. Project Explorer
3. Quick Panel
4. Console / Problems
5. Outline

If you close a view unintentionally, you can reopen it from the menu **Window > Show View**. Some of the views are under **Window > Show View > Other…**. You can drag and drop windows and resize them as you desire. If you unintentionally change something in the perspective and want to reset to the default, you can use **Window > Perspective > Reset Perspective**.

1.5.2 Customization

Eclipse is extremely flexible – you can customize almost anything if you know where to look. A good place to start for general Eclipse settings is **Window > Preferences**. One that I always turn on is **General > Editors > Text Editors > Show line numbers** (you can also access this from a pop-up menu by right clicking along the left edge of the code editor window). Most of these settings are at the Workspace level, but when you create a new Workspace there is an option under **Copy Settings** to copy various settings from an existing workspace if desired.
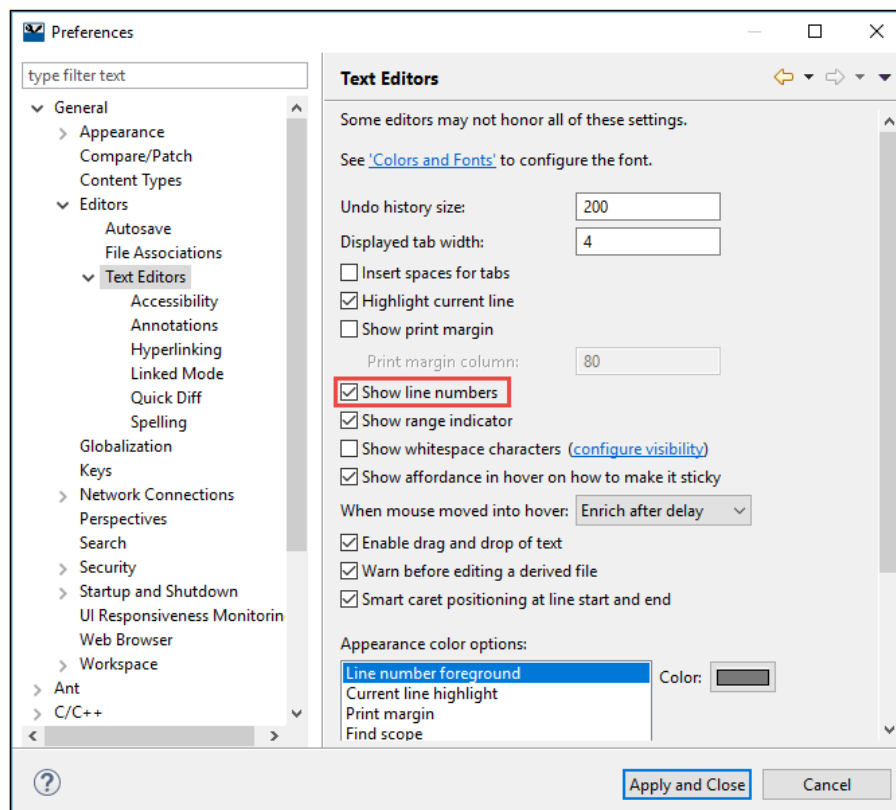
1.5.3 Project Explorer

In the Project Explorer, you will see any projects in your workspace and all their associated files.
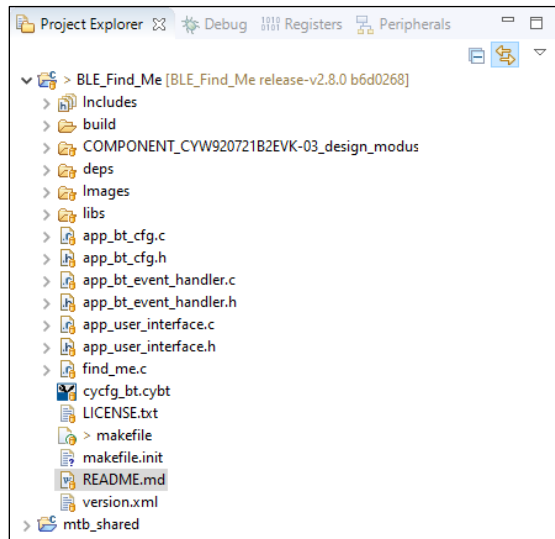
1.5.4 Quick Panel

Once you have created a new application, the Quick Panel is populated with common commands so that you don't have to hunt for them. For example, there are commands to build/program/debug the chip, links to launch other tools, and links to relevant documentation.
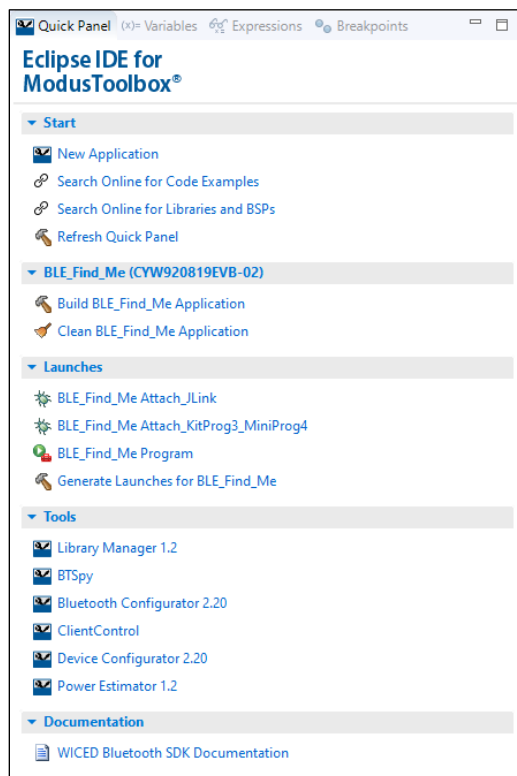
New Application

The first link in the Quick Panel is to create a new application. This launches a separate ModusToolbox tool called the Project Creator. It can also be run directly from the command line (more on that later).

Once the Project Creator opens, you must select a starting BSP and then one or more starter applications. They can be imported if you have your own custom BSP or template application. Once you have completed the selections, the tool creates the application(s) for you. When you run Project Creator from inside the Eclipse IDE, it will import them into Eclipse once they have been created.

More details on the Project Creator can be found in [Project Creator](#).

Build

The second section in the Quick Panel provides links to build and clean the selected application.

Launches

The launches section is used to program the kit or to attach to the kit with the debugger. Different launches are provided for different hardware such as KitProg3 (which is included on most Cypress kits) and J-Link. There may be other launch configs besides the common ones in the Quick Panel that can be accessed from **Run > Debug Configurations…** or **Run > External Tools > External Tool Configurations…**

The final link in this section to **Generate Launches** can be used if you change settings such as the target device, application name, or compiler optimization settings. In those cases, the existing launch configurations will point to old build files, so it is necessary to regenerate them to point to the new build location.

The launch configurations are stored in the project under the sub-directory *.mtbLaunchConfigs*. In some cases (e.g. manually renaming an application) these files may point to incorrect locations or may have multiple copies with different settings. If so, it is safe to remove them and regenerate them.

Tools

The tools section in the Quick Panel has links to various tools to allow configuration of the device (e.g. Device Configurator, Bluetooth Configurator, Library Manager) and to analyze Bluetooth messages (e.g. Client Control, BTSpy).

1.5.5 Eclipse IDE Tips & Tricks

Eclipse has several quirks that new users may find hard to understand at first. Here are a few tips to make the experience less difficult:

- If your code has IntelliSense issues, use **Program > Rebuild Index** and/or **Program > Build**.
- Sometimes when you import a project, you don't see all the files. Right-click on the project and select **Refresh**.
- Various menus and the Quick Panel show different things depending on what you select in the Project Explorer. Make sure you click on the project you want when trying to use various commands.
- Right-click in the column on the left- side of the text editor view to pop up a menu to:

- o   Show/hide line numbers
- o   Set/clear breakpoints

Refer also to our Eclipse Survival Guide for more tips and tricks.

1.6   Project Directory Organization

Once created, an embedded Bluetooth project will contain various files and directories.
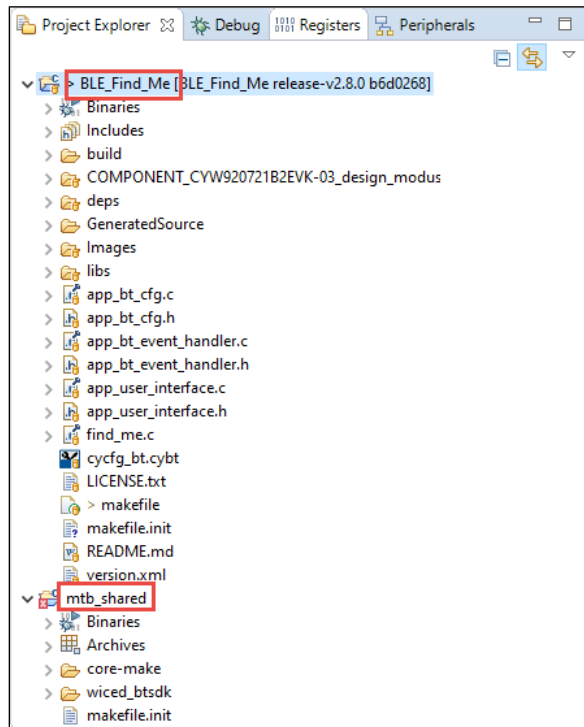
There are two top-level directories associated with most applications: the application project itself
(*BLE_Find_Me* in this example) and a location that contains shared library source code (*mtb_shared*).

1.6.1 Application Organization

The application project contains the source code files for the application, Bluetooth configuration
information, local libraries and information on shared libraries. The key parts of a project are:

- A directory with the name of the project
- Readme files (*README.md*)
- Application configurator files (e.g. *cycfg_bt.cybt* for Bluetooth)
- A *GeneratedSource* directory with the output files from the application level configurators
- Directories called *deps* and *libs*
- A *Makefile*
- Stack configuration files (e.g. *wiced_bt_cfg.c* or *app_bt_cfg.c*)
- Application C source files

README.md

The first file to open in the file editor window will be the readme file included with the application, if there is one. This gives general information about the platform or the application that you started with.

Application Configurator Files

Files for configurators are distinguished by their file extension. For example, any file that ends with *.cybt* is a Bluetooth configurator file.

GeneratedSource Directory

When you save from a configurator, the tool generates firmware in the *GeneratedSource* directory. The following are the most interesting/useful files (e.g. *cycfg_gatt_db.c* and *cycfg_gatt_db.h* for BLE).

Remember that the Device Configuration files (generated from *design.modus*) go in the BSP so they will NOT be in the *GeneratedSource* directory in the application unless you override the location.

deps

This directory contains *mtb* files that specify where the `make getlibs` command finds the libraries directly included by the project. Note that libraries included via *mtb* files may have their own library dependencies listed in the manifest files. We'll talk more about dependencies, *mtb* files and manifests when we discuss library management.

As you will see, the source code for libraries either go in the shared repository (for shared libraries) or in the *libs* directory inside the application (for libraries that are not shared).

libs

The *libs* directory contains source code for local libraries (i.e. those that are not shared) and *mtb* files for indirect dependencies (i.e. libraries that are included by other libraries). Most libraries are placed in a shared repo and are therefore not in the *libs* directory. The *libs* directory also includes a file called *mtb.mk* that specifies which shared libraries should be included in the application and where they can be found. This will be discussed in detail when we discuss library management.

As you will see, the *libs* directory only contains items that can be recreated by running `make getlibs`. Therefore, the *libs* directory should not typically be checked into a source control system.

In the case of the BTSDK, the file *index.html* contains the link to the API documentation which is what causes it to appear in the Documentation section of the quick panel.

Makefile

The *Makefile* is used in the application creation process – it defines everything that ModusToolbox needs to know to create/build/program the application. This file is interchangeable between the IDE and Command Line Interface so once you create an application, you can go back and forth between an IDE and CLI at will.

Various build settings can be set in the *Makefile* to change the build system behavior. These can be "make" settings or they can be settings that are passed to the compiler. Some examples are:

- Target Device (`TARGET=CYW920819EVB-02`)
- Method to generate the Bluetooth Device Address (`BT_DEVICE_ADDRESS?=default`)
- Enable OTA Firmware Upgrade (`OTA_FW_UPGRADE?=1`)
- Compiler Setting to Enabling Debug Trace Printing (`CY_APP_DEFINES+= -DWICED_BT_TRACE_ENABLE`)
- Compiler Optimization setting (`CONFIG=Debug`)
- Method to optionally include or exclude directories starting with the keyword COMPONENTS from the build (`COMPONENTS += bsp_design_modus`, `DISABLE_COMPONENTS+=bsp_design_modus`)

You will get to experiment with some of these settings in later chapters.

Note: `TARGET` and `CONFIG` are used in the launch configurations (program, attach, etc.). So, if you change either of these variables manually in the *Makefile*, <u>you must update or regenerate the Launch configs inside the Eclipse IDE</u>. Otherwise, you will not be programming/debugging the correct firmware.

Note: The `TARGET` board should have a *.mtb* file in the *deps* directory (if it is a standard BSP) and must be in the source file search path if it is in the shared location. Therefore, if you change the value of `TARGET` manually in the *Makefile*, <u>you must add the *.mtb* file for the new BSP</u> and then run `make getlibs` to update the search path in the *libs/mtb.mk* file.

Because of the reasons listed above, it is better to use the Library Manager to update the TARGET BSP since it makes all of the necessary changes to the project. If you are working exclusively from the command line and don't want to run the Library Manager, you can regenerate the *libs/mtb.mk* file by running `make getlibs` once the appropriate *.mtb* files are in place.

Stack Configuration Files

Many of the templates you will use in this class 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 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.

Application C file

All starter applications include at least one C source file that starts up the application (from the `application_start` function) and then implements other application functionality. The actual name of this file varies according to the starter application used to create the application. It is usually the same name as the template but there are exceptions to that rule. For example, in the templates provided for this class this top-level file is always called *app.c*.

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 header files can be found in the SDK under *wiced_btsdk/dev-kit/baselib/20819A1/<version>/include*.

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, 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 *wiced_btsdk/dev-kit/baselib/20819A1/include/wiced_bt_dev.h*.

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

```c
#include "app_bt_event_handler.h"
#include "app_bt_cfg.h"
#include "sparcommon.h"
#include "wiced_bt_trace.h"
#include "wiced_bt_stack.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;
}
```

1.6.2 Shared Library Organization

For embedded Bluetooth projects, a collection of libraries under *mtb_shared*/*wiced_btsdk* is of particular interest. These are the base libraries for all the supported devices (*wiced_btsdk/dev-kit/baselib*), BSPs for all the supported boards (*wiced_btsdk/dev-kit/bsp*), and Bluetooth tools such as ClientControl and BTSpy which you will use in later chapters (*wiced_btsdk/tools*).
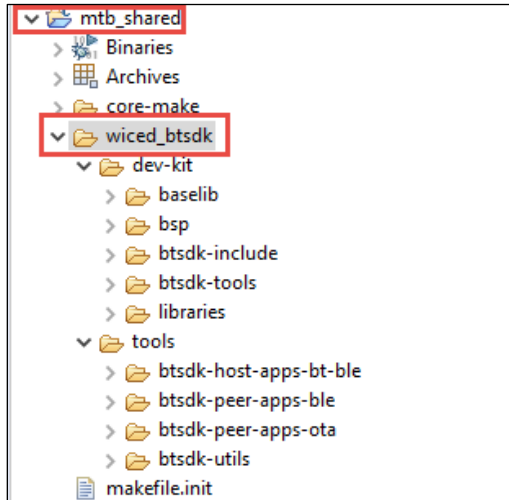
1.7 Library Management

A ModusToolbox project is made up of your application files plus libraries. A library is a related set of code, either in the form of C-source or compiled into archive files. These libraries contain code which is used by your application to get things done. These libraries range from low-level drivers required to boot the chip, to the configuration of your development kit (called Board Support Package) to Graphics or RTOS or CapSense, etc.

1.7.1 Library Classification

The way libraries are used in an application can be classified in several ways:

Shared vs. Local

Source code for libraries can either be stored locally in the application directory structure, or they can be placed in a location that can be shared between all the applications in a workspace.

Direct vs. Indirect

Libraries can be referenced directly by your application (i.e. direct dependencies) or they can be pulled in as dependencies of other libraries (i.e. indirect dependencies). In many applications, the only direct dependency is a BSP. The BSP will include everything that it needs to work with the device such as the HAL and PDL. In other applications there will be additional direct dependencies such as Wi-Fi libraries or Bluetooth libraries.

Fixed vs. Dynamic Versions

You can specify an exact version of a library to use in your application, or you can specify a major release version with the intent that you will get the latest compatible version.

1.7.2 Library Management Flow

Beginning with the ModusToolbox 2.2 release, we've developed a new way of structuring applications, called the MTB flow. Using this flow, applications can share BSPs and libraries. If necessary, different applications can use different versions of the same shared library. Sharing resources reduces the number of files on your computer and speeds up subsequent application creation time. Shared BSPs, libraries and versions are stored in a new *mtb_shared* directory adjacent to your application directories.

You can easily switch a shared BSP or library to become local to a specific application, or back to being shared.

By default, the source code for all libraries will be put in a shared directory. (You'll see how to make them local in the

Library Manager Library Manager section.) The default directory name is *mtb_shared* and its default location is parallel to the application directories (i.e. in the same application root path). The name and location of the shared directory can be customized using the *Makefile* variables `CY_GETLIBS_SHARED_NAME` and `CY_GETLIBS_SHARED_PATH`.

Source code for local libraries will be placed in the *libs* directory inside the application.

ModusToolbox knows about a library in your project in one of two ways depending on whether the library is a direct dependency or an indirect dependency that is included by another library.

Direct Dependencies

For direct dependencies, there will be one or more *mtb* files somewhere in the directories of your project (typically in the deps directory but could be anywhere except the libs directory). An *mtb* file is simply a text file with the extension *.mtb* that has three fields separated by #:

- A URL to a Git repository somewhere that is accessible by your computer such as GitHub
- A Git Commit Hash or Tag that tells which version of the library that you want
- A path to where the library should be stored in the shared location (i.e. the directory path underneath *mtb_shared*).

A typical *mtb* file looks like this:

```
https://github.com/cypresssemiconductorco/TARGET_CY8CKIT-062S2-43012/#latest-
v1.X#$$ASSET_REPO$$/TARGET_CY8CKIT-062S2-43012/latest-v1.X
```

The variable `$$ASSET_REPO$$` points to the root of the shared location - it is specified in the application's *Makefile*. If you want a library to be local to the app instead of shared you can use `$$LOCAL$$` instead of `$$ASSET_REPO$$` in the *mtb* file before downloading the libraries. Typically, the version number is excluded from the path for local libraries since there can only be one local version used in a given application.

Using the above example, a library local to the app would normally be specified like this:

```
https://github.com/cypresssemiconductorco/TARGET_CY8CKIT-062S2-43012/#latest-
v1.X#$$LOCAL$$/TARGET_CY8CKIT-062S2-43012
```

Indirect Dependencies

Indirect dependencies for each library are found using information that is stored in a manifest file. For each indirect dependency found, the Library Manager places an *mtb* file in the *libs* directory in the application.

Once all the *mtb* files are in the application, the `make getlibs` process (either called directly from the command line or by the Library Manager) finds the *mtb* files, pulls the libraries from the specified Git repos and stores them in the specified location (i.e. *mtb_shared/* for shared libraries and *libs/* for local libraries). Finally, a file called *mtb.mk* is created in the application's *libs* directory. That file is what the build system uses to find all the shared libraries required by the application.

Since the libraries are all pulled in using `make getlibs`, you don't typically need to check them in to a revision control system - they can be recreated at any time from the *mtb* files by re-running

`make getlibs`. This includes both shared libraries (in *mtb_shared*) and local libraries (in *libs*) - they all get pulled from Git when you run `make getlibs`.

The same is true for the *mtb* files for indirect references and the *mtb.mk* file which are also stored in the *libs* directory. In fact, the default *.gitignore* file in our code examples excludes the entire libs directory since you should not need to check in any files from that directory.

In summary, the default locations of files relative to the application root directory for the MTB flow are:

	direct / shared	direct / local	indirect / shared
.mtb file	./deps/	./deps/	./libs/
library source code	../mtb_shared/	./libs/	../mtb_shared/

1.8   Manifests

Manifests are XML files that tell the Project Creator and Library Manager how to discover the list of available boards, example projects, and libraries. There are several manifest files.

- The "super manifest" file contains a list of URLs that software uses to find the board, code example, and middleware manifest files.
- The "app-manifest" file contains a list of all code examples that should be made available to the user.
- The "board-manifest" file contains a list of the boards that should be presented to the user in the new project creation wizards as well as the list of BSPs that are presented in the Library Manager tool.
- The "middleware-manifest" file contains a list of the available middleware (libraries). Each middleware item can have one or more versions of that middleware available.

To use your own examples, BSPs, and middleware, you need to create manifest files for your content and a super-manifest that points to your manifest files.

Once you have the files created an in a Git repo, place a *manifest.loc* file in your *<user_home>/.modustoolbox* directory that specifies the location of your custom super-manifest file (which in turn points to your custom manifest files). For example, a *manifest.loc* file may have:

```
# This points to the IOT Expert template set
https://github.com/iotexpert/mtb2-iotexpert-manifests/raw/master/iotexpert-super-
manifest.xml
```

Note that you can point to local super-manifest and manifest files by using file:/// with the path instead of https://. For example:

```
file:///C:/MyManfests/my-super-manifest.xml
```

To see examples of the syntax of super-manifest and manifest files, you can look at the Cypress provided files on GitHub as listed below. Each repo may hold several manifest file versions – be sure to look at the latest version.

- Super Manifest: https://github.com/cypresssemiconductorco/mtb-super-manifest

- App Manifest: https://github.com/cypresssemiconductorco/mtb-ce-manifest
- Board Manifest: https://github.com/cypresssemiconductorco/mtb-bsp-manifest
- Middleware Manifest: https://github.com/cypresssemiconductorco/mtb-mw-manifest

1.9 Board Support Packages

Each project is based on a target set of hardware. This target is called a "Board Support Package" (BSP). It contains information about the chip(s) on the board, how they are programmed, how they are connected, what peripherals are on the board, how the device pins are connected, etc. A BSP directory starts with the keyword "TARGET". By default, embedded Bluetooth design BSPs are placed under *mtb_shared/wiced_btsdk/dev-kit/bsp*.

1.9.1 BSP Directory Structure
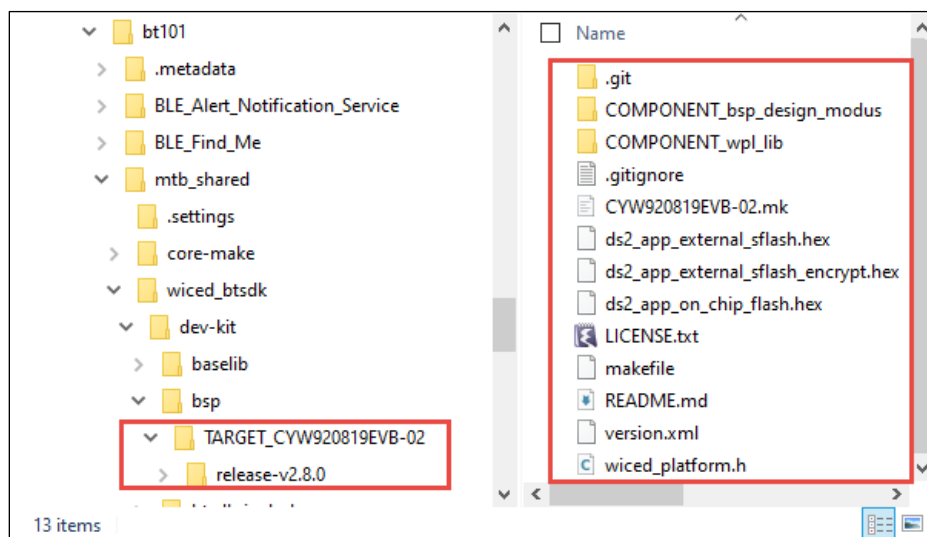
README.md

This file contains useful documentation on the BSP.

Makefile and <Boardname>.mk

These contain information used by the build system to properly use the various BSP files.

wiced_platform.h

Platform definitions for board-level peripherals. Note that many peripheral pins are also defined in the *deisgn.modus* file using the Device Configurator tool, usually with different names. For example, *wiced_platform.h* has `WICED_GPIO_PIN_LED_1` while the configurator file has `LED1` for the same pin. Either name can be used, but the one in the device configurator is typically shorter.

The *wiced_platform.h* file may also contain other useful macros such as `LED_STATE_OFF`, `LED_STATE_ON`, `BTN_PRESSED`, and `BTN_OFF`.

COMPONENT_bsp_design_modus

This directory contains the default configuration for the kit. In some cases, this file will be modified or over-ridden for a single application. The methods to do that are described next.

Hint The definitions from the Device Configurator are often not included by default in an application. To get access to these, just include the file *cycfg.h* in your source code.

1.9.2 Modifying the BSP Configuration for a Single Application

Since the wiced_btsdk libraries are shared among all applications in a workspace, any changes to files contained in the SDK will affect all applications in that workspace. This is especially important for the BSP since it contains the Device Configurator files.

Therefore, if you want to use the Device Configurator to change settings for an application without affecting other applications, you can do one of three things: (1) override the Device Configurator files for a single application; (2) create a custom BSP; or (3) relocate the BSP to be local to the application. The first two options are good when a single application requires a different configuration while the second option is better if you want a single custom BSP that you can use for multiple applications. Method 3 is not typically recommended since it results in modifications to the BSP library itself, making it more difficult to update versions later but it is simple to do using the Library Manager for experimenting before creating a custom BSP.

Override Device Configuration

The steps to override the Device Configurator files for a single application are:

1. Copy the *COMPONENT_bsp_design_modus* directory and its contents from the BSP to the application's project directory (the directory with the top-level source code). Change the name to something unique that still starts with "COMPONENT_". For example:

    COMPONENT_custom_design_modus

2. Disable the configurator files from the BSP and include your custom configurator files by modifying the `COMPONENTS` variable in the *Makefile* to change *bsp_design_modus* to whatever name you chose. For example:

    ```
    COMPONENTS += custom_design_modus
    ```

3. Make sure your project is selected and open the Device Configurator from the Quick Panel. Make the required changes and save/exit the configurator.
    a. It is a good idea to look at the banner in the configurator to verify it is opening the file from the application, not from the SDK.

Creating your own BSP

If you want to change more than just the configuration from the *COMPONENT_BSP_DESIGN_MODUS* directory (such as for your own custom hardware or for different linker options), you can create a full BSP based on an existing one. To create your own custom BSP, do the following:

1. Locate the closest-matching BSP to your intended custom BSP and set that as the default `TARGET` for the application in the *Makefile*.

2. In the application directory, run the `make bsp` target.

   Specify the new board name by passing the value to the `TARGET_GEN` variable. For example:

   ```
   make bsp TARGET_GEN=MyBSP
   ```

   This command creates a new BSP in the root of the application project.

   It also creates *.mtbx* files for all the BSP's dependences. The Project Creator tool uses these files when you import your custom BSP into that tool. These files can also be used with the `make import_deps` command if you manually include the custom BSP in a new application.

   Note The BSP used as your starting point may have library references that are not needed by your custom BSP. You can delete these from the BSP. Be sure to remove the corresponding *.mtbx* files as well.

3. Update the application's *Makefile* `TARGET` variable to point to your new BSP. <u>You must do this BEFORE running the Device Configurator so that it opens the correct file</u>.

4. Update the application's `SUPPORTED_TARGETS` variable to include your new BSP.

5. Open the Device Configurator to customize settings in the new device's *design.modus* file for pin names, clocks, power supplies, and peripherals as required. Address any issues that arise.

6. If using an IDE, regenerate the configuration settings and launches to reflect the new BSP. Use the appropriate command(s) for the IDE(s) that are being used. For example: `make vscode`. For Eclipse, you can use the **Generate Launches** link in the Quick Panel.

7. When you want to use a custom BSP in a new application, the easiest method to include it is to use the Import functionality in Project Creator. You can also use other manual library management techniques if you prefer.

If you want to re-use a custom BSP on multiple applications, you can copy it into each application, or you can put it into a version control system such as Git. See [Manifests](#) for information on how to create a manifest to include your custom BSPs and their dependencies if you want them to show up as standard BSPs in the Project Creator and Library Manager. Note that if you put your custom BSP in a shared location instead of local to the application, it will need to be included in the `SEARCH` path in the *Makefile* unless it is included in a manifest to get it to show up as a standard BSP.

1.10 Tools

The ModusToolbox installer includes many tools that can be used along with the IDE, or stand-alone. These tools include configurators that are used to configure hardware blocks, as well as utilities to create projects without the IDE or to manage BSPs and libraries. All the tools can be found in the installation directory. The default path (Windows) is:

*C:/Users/<user-name>/ModusToolbox/tools_2.2*:
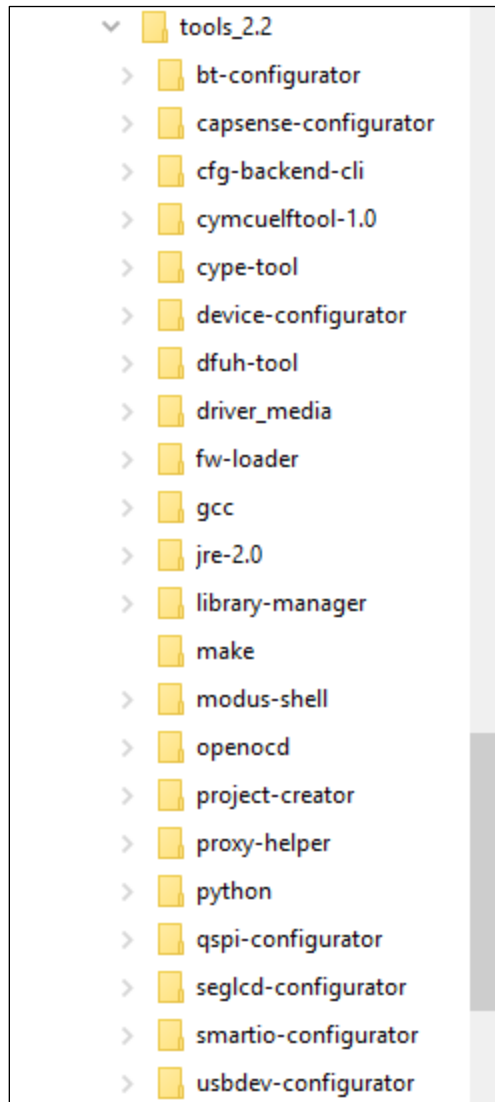
1.10.1 Project Creator

ModusToolbox software includes a project-creator tool that you can open from the IDE or as a stand-alone tool. The tool sets up new applications with the required target hardware support, wireless 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 the **New ModusToolbox Application** item in the **File > New** menu.

The first thing the Project Creator tool does is read the configuration information from the GitHub site so that it knows all the BSPs etc. that we support. It then asks you to select your **Target Hardware** from that list. The kit used in this course is the CYW920819EVB-02.
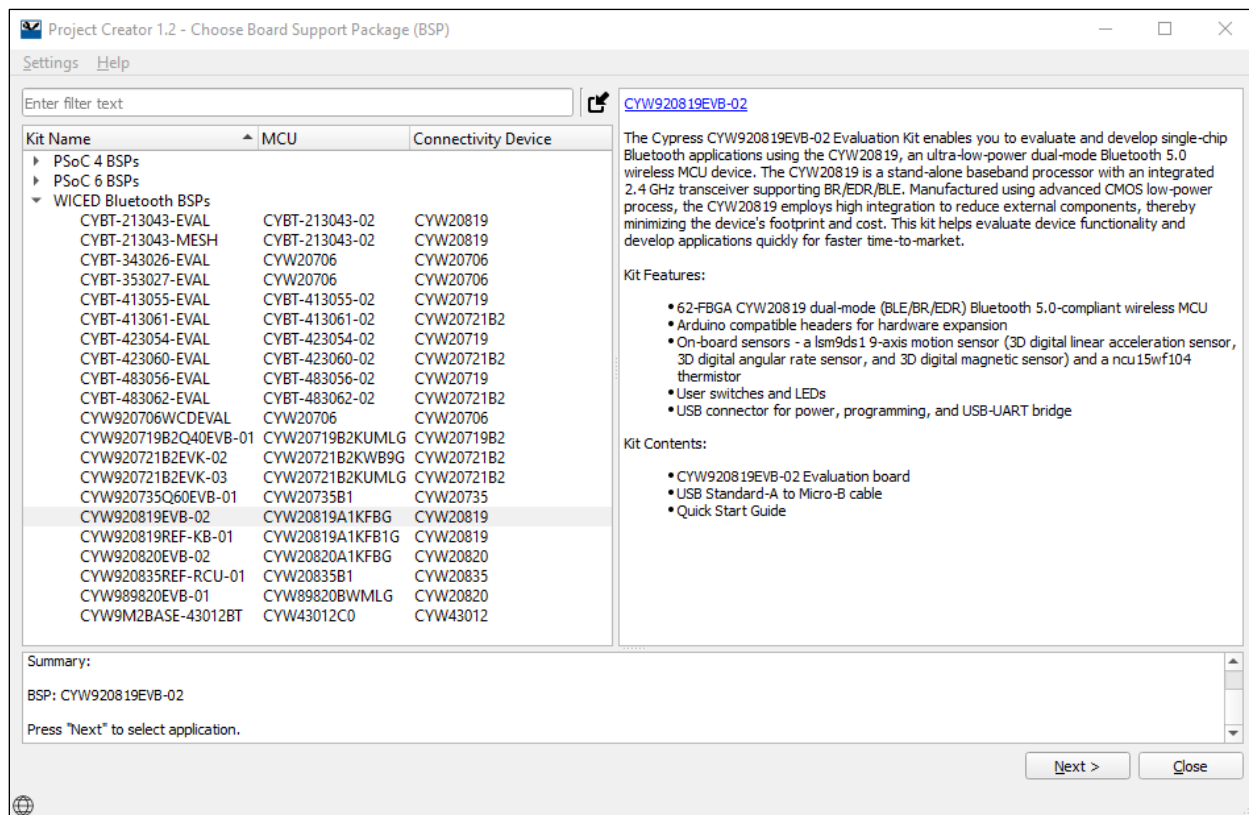
Note that there is an **Import** button so you can specify your own custom BSP if you have one. We show you how to create your own BSP in the Board Support Packages section of this chapter.

Clicking **Next >** will present you with all the code examples supported by the BSP you chose. Pick one or more applications to create and give them names. You can specify a different **Application Root Path** if you don't want to use the default value. A directory with the name of your application(s) will be created inside the specified Application Root Path. The value for the Application Root Path is remembered from the previous invocation, so by default it will create applications in the same location that was used previously.
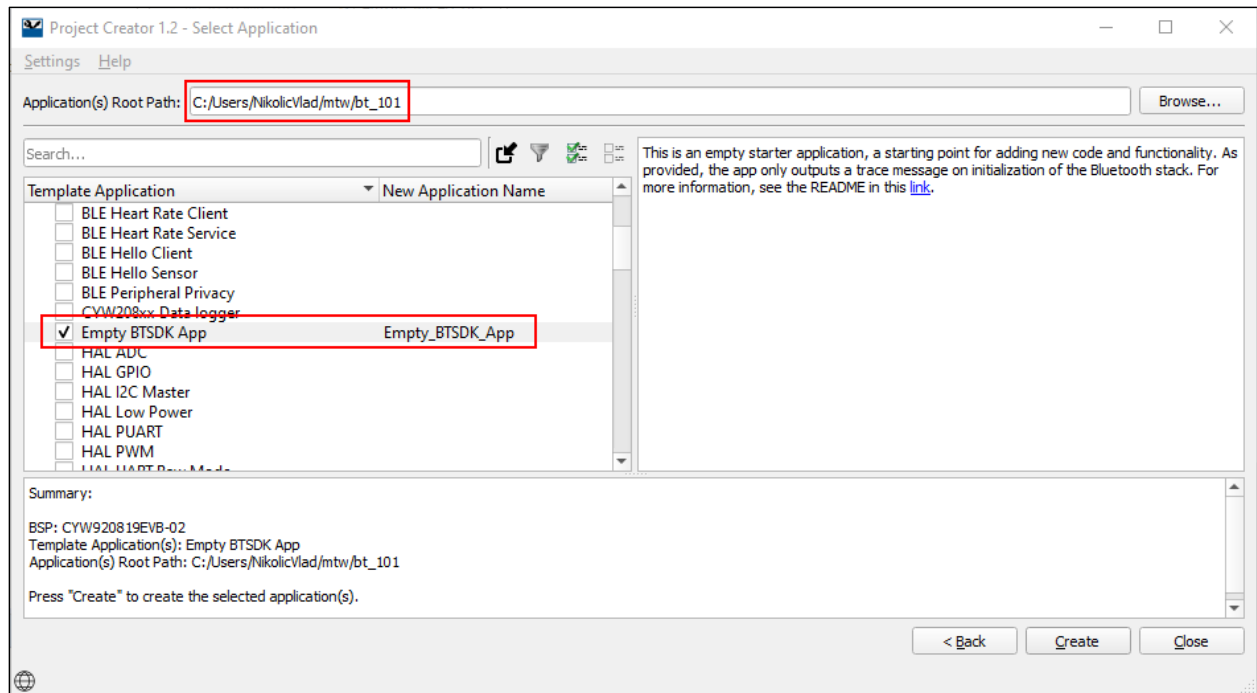
You can use the "Search…" box to enter a string to match from the application names and their descriptions. For example, if you enter "ble," you may see a list as shown in the following image. The mesh applications match because their descriptions have "ble" in them. (The exact list will change over time as new code examples are created):
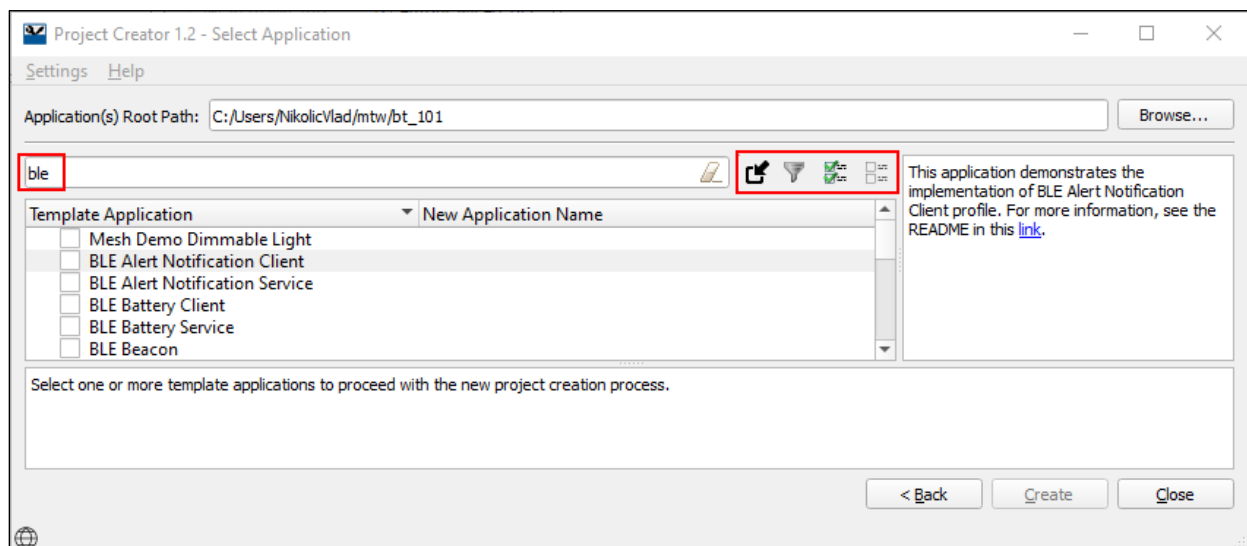
If desired, you can select all the applications in the search result by using the **Select All** button or you can unselect them with the **Unselect All** button.

The **Filter** button can be used to list only the applications which are currently checked. This can be useful if you have a large number of applications checked and want to see them all listed together.

If you want to start with your own local project or template instead of one of the code examples that we provide, click the **Import** button. This allows you to browse files on your computer and then create a new application based on an existing one. Make sure the path you select is to the directory that contains the *Makefile* (or one above it – more on that in a minute), otherwise the import will fail.

Once you select an application directory, it will show up at the top of the list of applications.
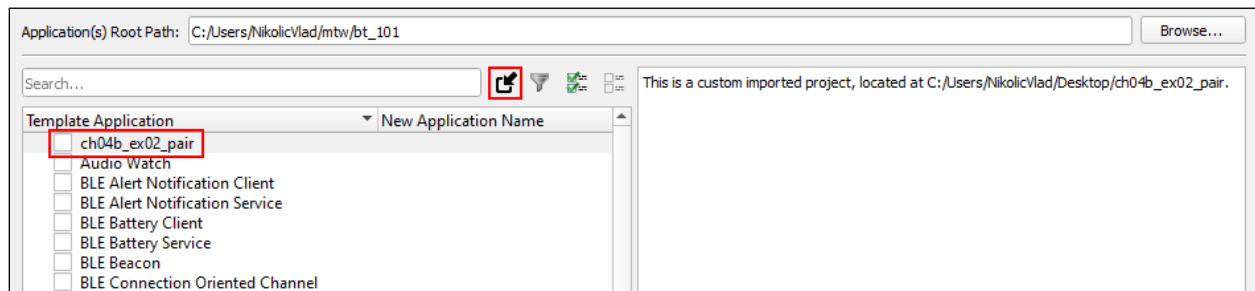
Note that the existing project can be one that is in your workspace or one that is located somewhere else. It does <u>not</u> need to be a full Eclipse project. At a minimum, it needs a *Makefile* and source code files, but it may contain other items such as configurator files and *mtb* files which are a mechanism to include dependent libraries in an application.

If you specify a path to a directory that is above the *Makefile* directory, the hierarchy will be maintained, and the path will be added to each individual application name inside Eclipse. This can be useful if you have a directory containing multiple applications in sub-directories and you want to import them all at once. For example, if you have a directory called *myapps* containing the 2 subdirectories *myapp1* and *myapp2*, when you import from the *myapps* level into Eclipse, you will get a project called *myapps.myapp1* and *myapps.myapp2*.

Once you have selected the application(s) you want to create (whether it's one of our code examples or one of your own applications), click **Create** and the tool will create the application(s) for you.

The tool runs the `git clone` operation and then the `make getlibs` operation. The project is saved in the directory you specified previously. When finished, click **Close**.

1.10.2 Library Manager

ModusToolbox provides a GUI for helping you manage library files. You can run this from the Quick panel link **Library Manager**. It is also available outside the IDE from *ModusToolbox/tools_2.2/library-manager*. If you are using the command line, you can launch it from an application directory using the command `make modlibs`.

The Library Manager knows where to find libraries and their dependencies by using manifest files. Therefore, *mtb* (or *lib* files for the LIB flow) included in your application that are not in the manifest file will NOT show up in the Library Manager. This may include your own custom libraries or libraries that you got from another source. You can create and include one or more custom manifest files for your custom libraries so that you can use the Library Manager, or you can manage them manually.

The Library Manager GUI looks like this when using the MTB flow. The tabs and windows will be slightly different for the LIB flow, but we will focus on the MTB flow here.
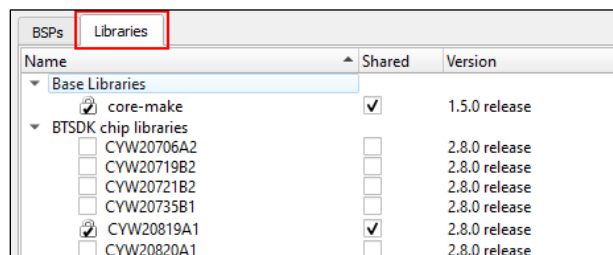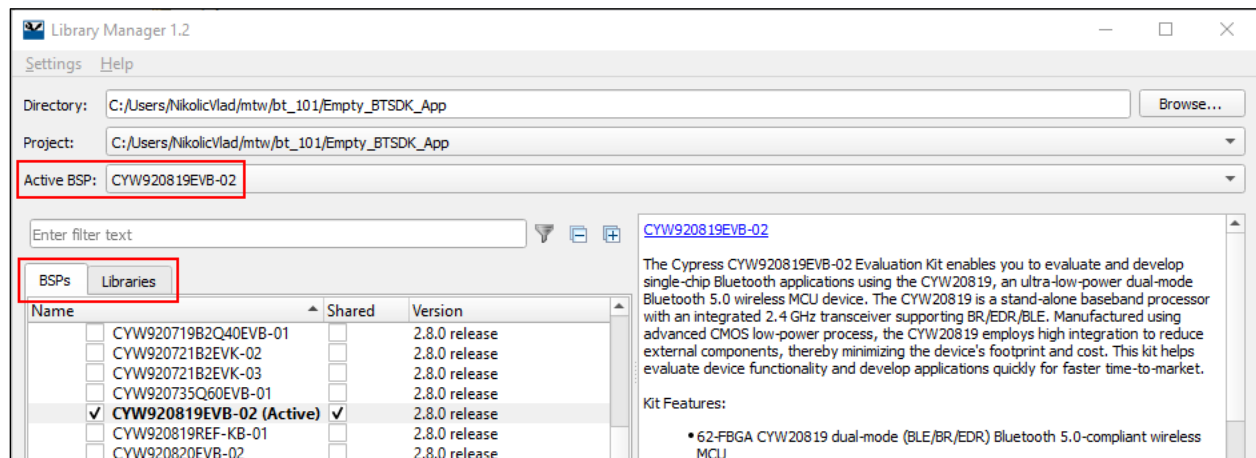
As you can see, the Library Manager has two tabs: *BSPs* and *Libraries*.

The BSPs tab provides a mechanism to add/remove BSPs from the application and to specify which BSP (and version) should be used when building the application. An application can contain multiple BSPs, but a build from either the IDE or command line will build for and program to the "Active BSP" selected in the Library Manager. The Libraries tab allows you to add and remove libraries, as well as change their versions.

Shared Column: Both BSPs and Libraries can be "Shared" or not (i.e. "Local"). By default, most libraries are shared, meaning that the source code for the libraries is placed in the workspace's shared location (e.g. *mtb_shared*). If you uncheck the Shared box for a given library, its source code will be copied to the libs directory in the application itself. Note that this column only exists for the MTB flow.

Version Column: You can select a specific fixed version of a library for your application or choose a dynamic tag that points to a major version but allows `make getlibs` to fetch the latest minor version (e.g. Latest 1.X). The drop-down will list all available versions of the library.

Locked Items: A library shown with a "lock" symbol is an indirect dependency (meaning its *mtb* file is in the *libs* directory). An indirect dependency is included because another library requires it, so you cannot remove it from your application unless you first remove the library that requires it. You can change an indirect dependency from shared to local or you can change its version, but if you do it is converted to a direct dependency (meaning its *mtb* file is moved to the *deps* directory). This allows the local/version information to be retained by the application even if the *libs* directory is removed or is not checked into version control.

Behind the scenes, the Library Manager reads/creates/modifies *mtb* and *lib* files, creates/modifies the *mtb.mk* file, and then runs `make getlibs`. For example, when you change a library version and click **Update**, the Library Manager modifies the version specified in the corresponding *mtb* or *lib* file and then runs `make getlibs` to get the specified version.

Active BSP: When you change the Active BSP and then click **Update**, the Library Manager edits the `TARGET` variable in the *Makefile*, creates the *deps/<bsp>.mtb* file if it is a newly added library, updates the *libs/mtb.mk* file and then updates any Eclipse launch configs so that they point to the new BSP.

1.10.3 ModusToolbox Configurators

ModusToolbox software provides graphical applications called configurators that make it easier to configure a hardware block. For example, instead of having to search through all the documentation to configure a serial communication block as a UART with a desired configuration, open the appropriate configurator to set the baud rate, parity, stop bits, etc. Upon saving the hardware configuration, the tool generates the C code to initialize the hardware with the desired configuration.

Many configurators do not apply to all types of projects. So, the available configurators depend on the project/application you have selected in the Eclipse IDE Project Explorer. Configurators are independent of each other, but they can be used together to provide flexible configuration options. They can be used stand alone, in conjunction with other configurators, or within an IDE. Everything is bundled together as part of the installation. Each configurator provides a separate guide, available from the configurator's Help menu. Configurators perform tasks such as:

- Displaying a user interface for editing parameters
- Setting up connections such as pins and clocks for a peripheral
- Generating code to configure middleware

Device Configurator

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 pins and peripherals on the device that you can set up. As mentioned earlier, the *design.modus* file is shared by all applications in a workspace that use a given BSP. Keep that in mind if you chose to edit the *design.modus* file.

As described earlier, you can override the *design.modus* for a single application. You will use this method in the exercises in the next chapter.

To launch the Device Configurator, click on the link in the Quick Panel or enter `make config` from the command line in your application directory. As you can see there are sections for Peripherals and Pins on the left. When you enable a peripheral or pin, the upper right-hand panel allows you to select configuration options and to open the documentation for that element.

It is a good idea to look at the path at the top of the configurator window to verify you are editing the file from the expected location (either in the BSP or in the application).
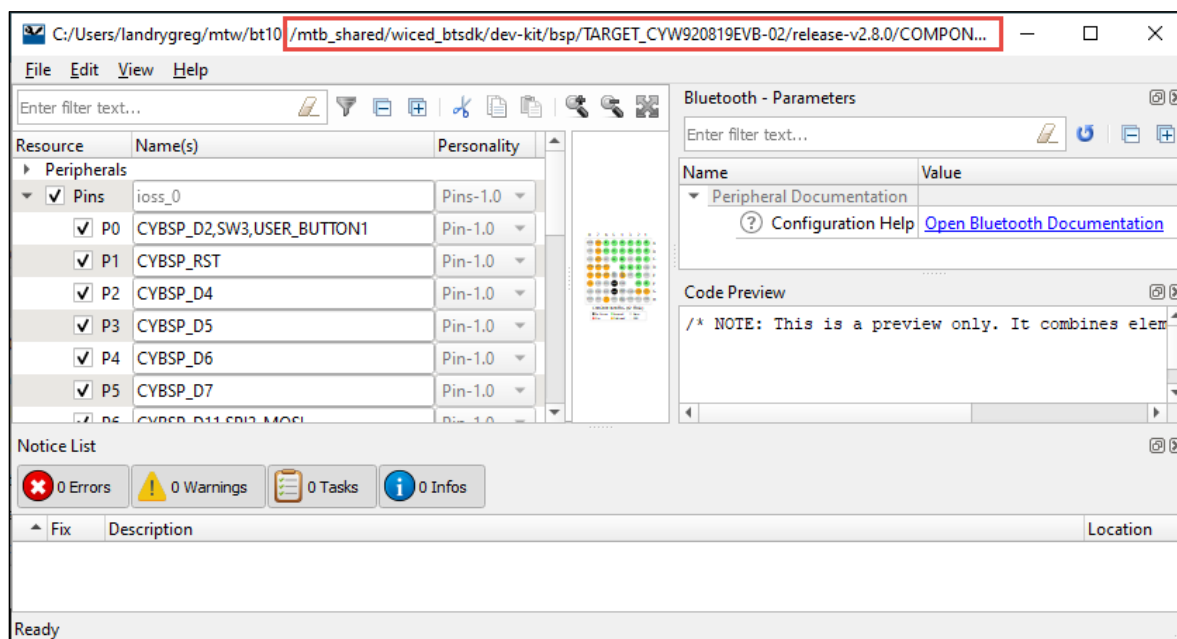
Hint The **+** and **-** buttons can be used to expand/contract all categories and the **filter** button can be used to show only items that are selected. This is particularly useful on the Pins tab since the list of pins is sometimes quite long.

Once you save the configurator information (**File > Save**) it creates/updates the Generated Source files in the project BSP or the local copy if you are using the override method or if you have a custom BSP.

Bluetooth Configurator

The Bluetooth Configurator is a tool that will build a semi-customized GATT database and device configuration for Bluetooth Low Energy applications. It will be covered in detail in later chapters.

1.10.4 Command Line Interface (CLI)

In addition to the Eclipse IDE for ModusToolbox, there is a command line interface that can be used for all operations including downloading applications, running configurators, building, programming, and even debugging.

The make/build system works the same in all environments so once you have an application setup in an IDE, you can go back and forth between the command line and IDE at will.

If you start from the Eclipse IDE, the application is automatically capable of command line operations. If you start from the command line, you can use the *ModusToolbox Application Import* command from inside Eclipse to import it. In either case, once it is set up, you can use both methods interchangeably.

Windows

To run the command line, you need a shell that has the correct tools (such as git, make, etc.). This could be your own Cygwin installation or Git Bash but it is recommended to use "modus shell", which is based on Cygwin and is installed as one of the ModusToolbox tools. To run it, search for modus-shell (it is in the ModusToolbox installation under *ModusToolbox/tools_2.2/modus-shell*). It is also listed in the Start menu under **ModusToolbox 2.2**.
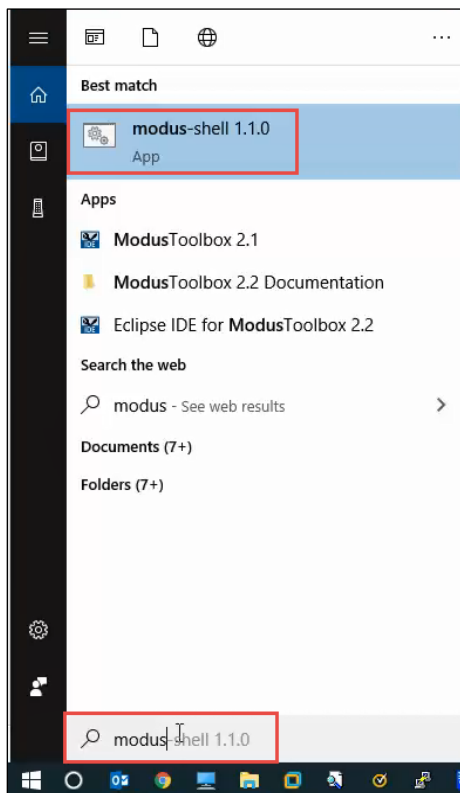
macOS / Linux

To run the command line on macOS or Linux, just open a terminal window.

Make Targets (CLI Commands)

To run commands, you need to be at the top level of a ModusToolbox project where the *Makefile* is located.

The following table lists a few helpful make targets (i.e. commands). Refer also to the [ModusToolbox User Guide](#) document.

Make Command	Description
`make help`	This command will print out a list of all the make targets. To get help on a specific target type `make help CY_HELP=getlibs` (or whichever target you want help with).
`make getlibs`	Process all the *.mtb* files and bring all the libraries into your project.
`make debug`	Build your project, program it to the device, and then launch a GDB server for debugging.
`make program`	Build and program your project.
`make qprogram`	Program without building.
`make config`	This command will open the Device Configurator.
`Make config_bt`	This command will open the Bluetooth Configurator.
`make get_app_info`	Prints all variable settings for the app.
`make get_env_info`	Prints the tool versions that are being used.
`make printlibs`	Prints information about all the libraries including Git versions

The help make target by itself (e.g. `make help`) will print out top level help information. For help on a specific variable or target use `make help CY_HELP=<variable or target>`. For example:

```
make help CY_HELP=build
```
or
```
make help CY_HELP=TARGET
```

1.11   Visual Studio Code (VS Code)

One alternative to using the Eclipse-based IDE is a code editor program called VS Code. This tool is quickly becoming a favorite editor for developers. The ModusToolbox command line knows how to make all the files required for VS Code to edit, build, and program a ModusToolbox program.

To use VS Code, first create an application using Project Creator. Then, from the command line enter the following command inside the application's top-level directory:

```
make vscode
```

The output will look like the following:

The message at the bottom of the command window tells you the next steps to take. However, for steps 2, 4, and 5, it is better to instead open the workspace file so that you will see the application as well as the shared library files from inside VS Code. From the command line just enter `code application-name>.code-workspace` to start VS Code and load the workspace all in one step.

1.12 Tour of Documentation

1.12.1 In the Eclipse IDE for ModusToolbox

In the Eclipse IDE Help menu there are 2 items of particular interest:
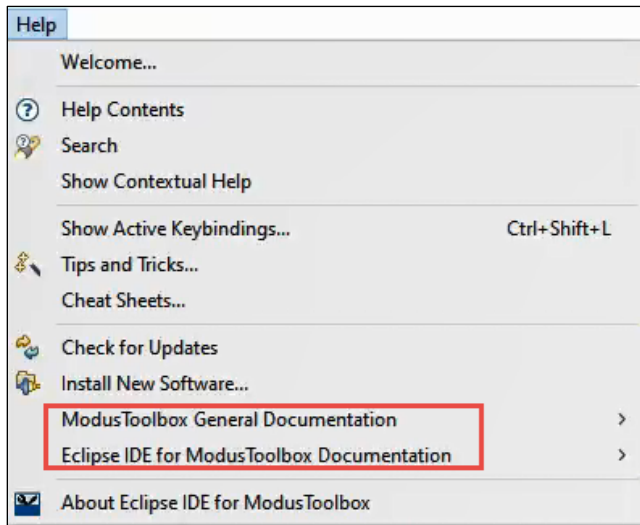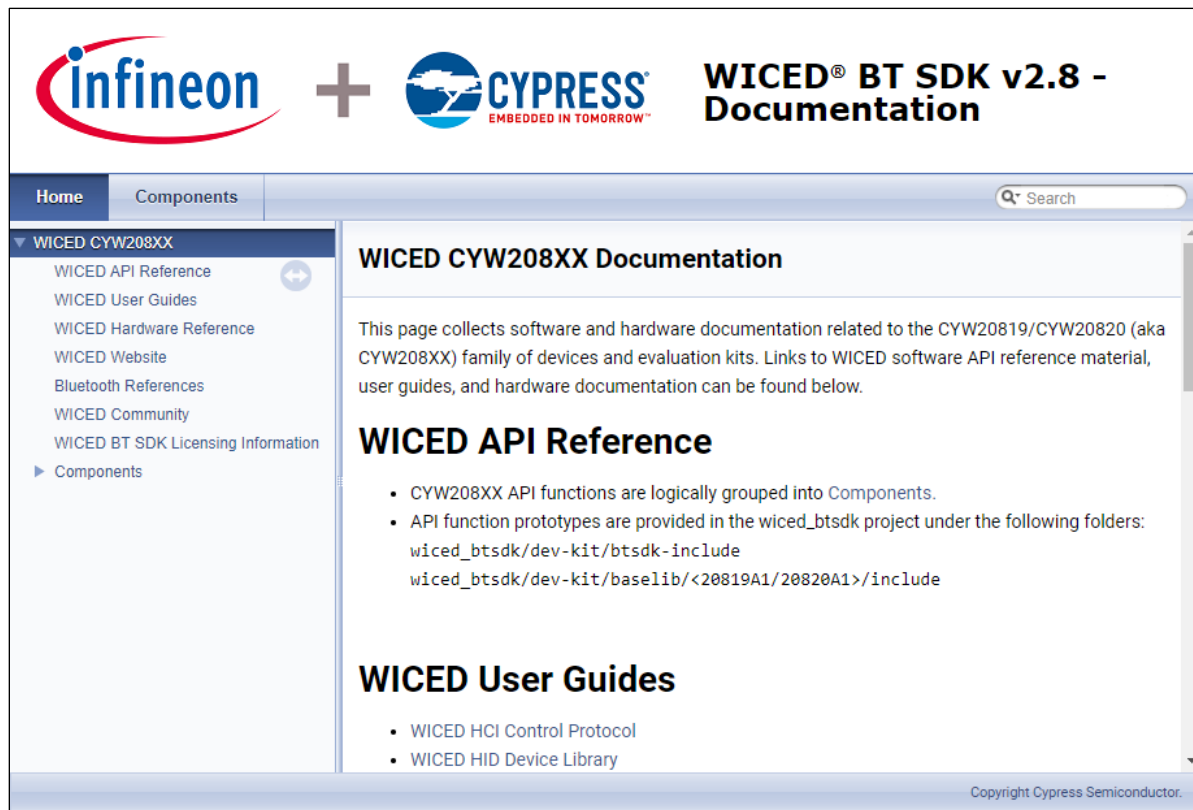

ModusToolbox General Documentation

The first item has (among other things) a link to the ModusToolbox Documentation Index which is a web page that looks like this:
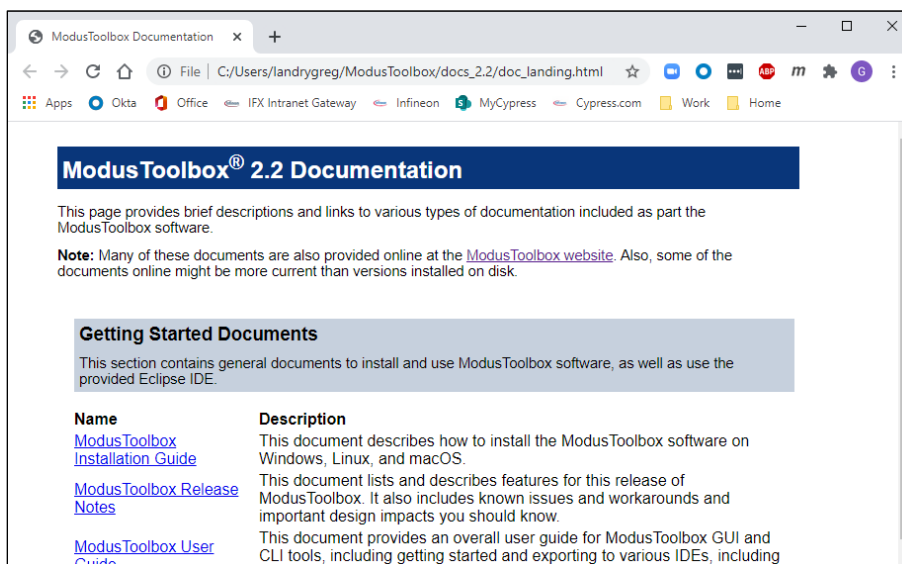
At the bottom of that page you will find a link to the BT SDK API guide.
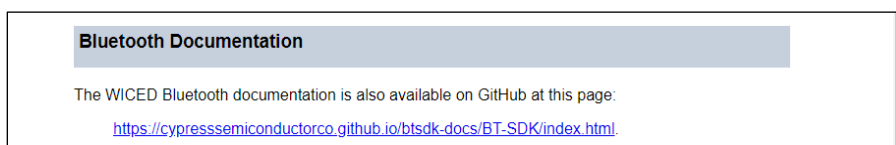
The Bluetooth Documentation link takes you to a page where you select from a list of Bluetooth devices. This page is also available directly from a link in the Quick Panel. Once you select your device, you will get to this page:

The links on the right are useful for specific topics while the section on the left is invaluable for understanding the API functions, data types, enumerations, macros, etc.

Eclipse IDE for ModusToolbox Documentation

The second item on the Help Menu of interest contains documents for:

- Quick Start Guide
- User Guide
- Eclipse IDE Survival Guide

The last item contains tips and tricks on using the Eclipse IDE with Cypress devices intended for those who have relatively little experience with Eclipse.

1.12.2 On the Web

Navigating to **www.cypress.com > Design Support > Community** will take you to the following site (the direct link is https://community.cypress.com/):
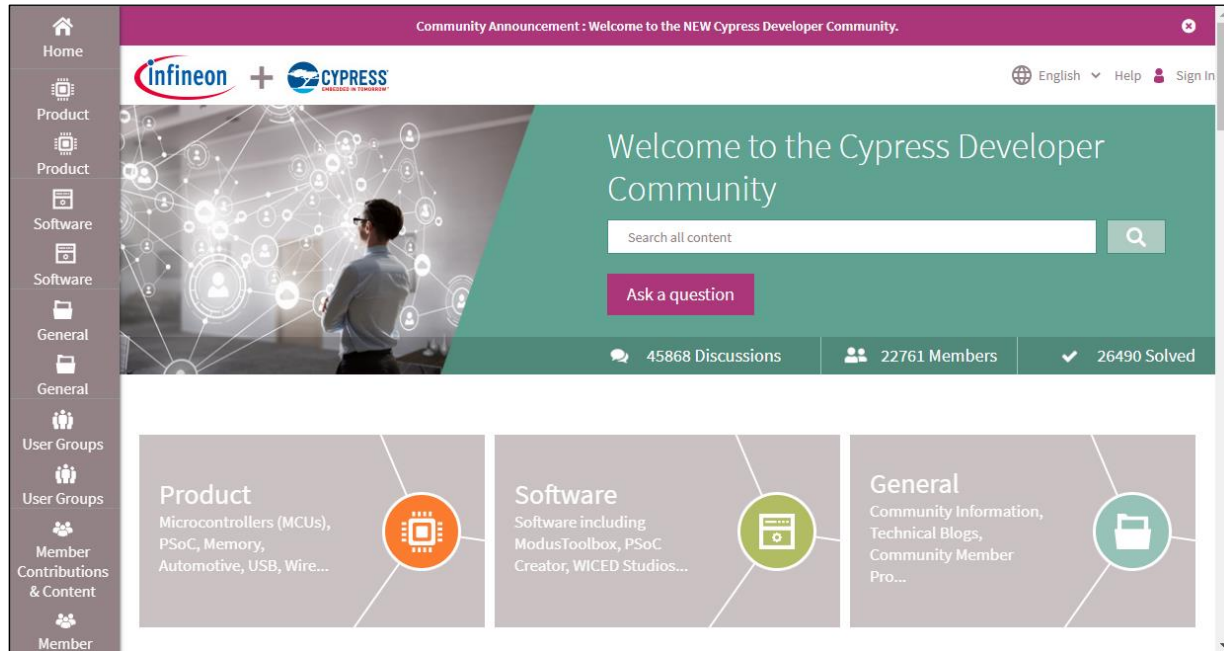
There are links to forums for Wireless, ModusToolbox, ModusToolbox Bluetooth SDK, etc. There are also links to Knowledge Base Articles, User Groups, Blogs, Code Examples, and a search tool.
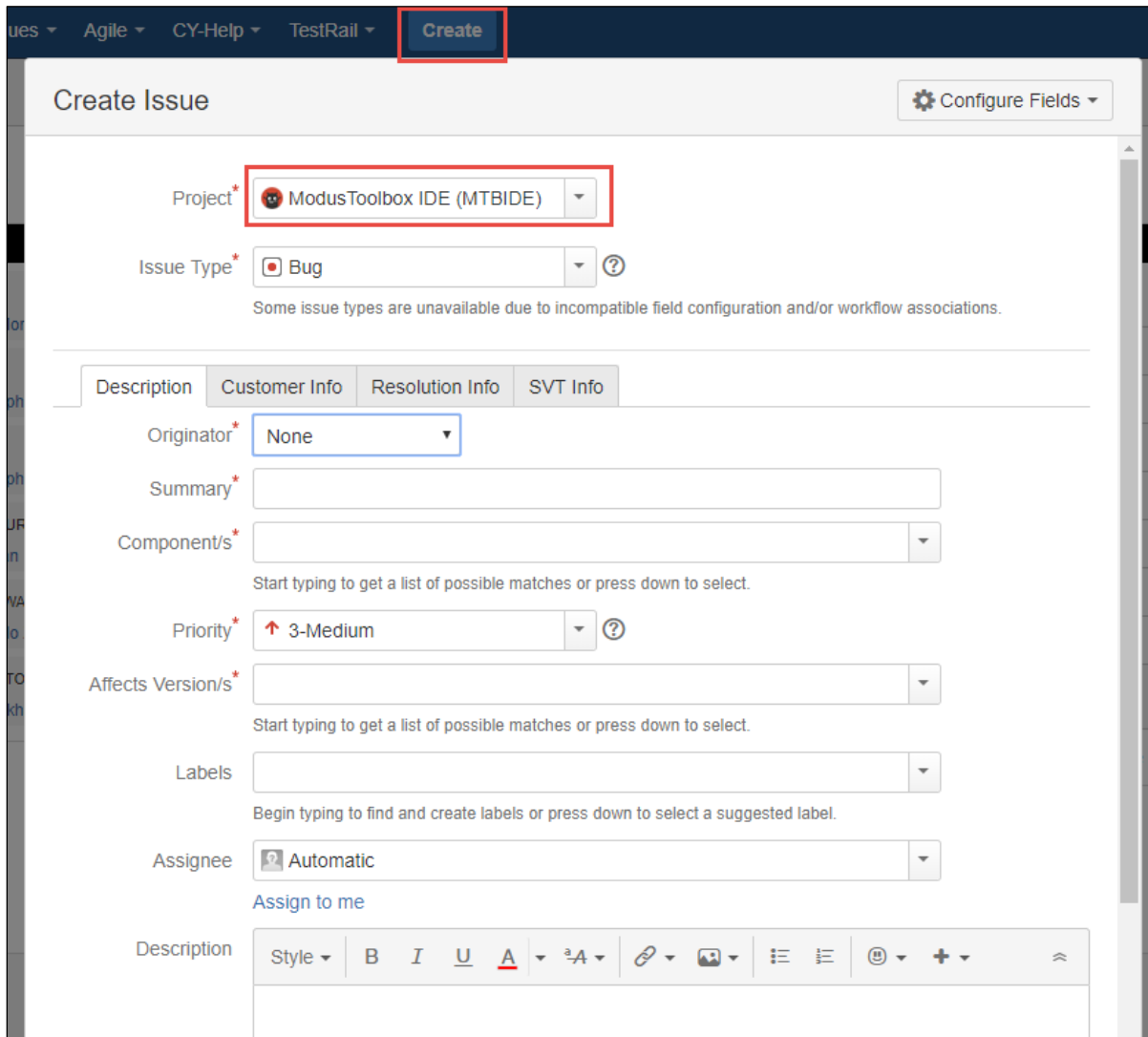
If you register or login, you can post questions to the various forums when you need help.

1.13   Reporting Issues

If you are an Infineon employee and you find an issue in ModusToolbox (bug, missing or confusing documentation, enhancement request), please use a "JIRA" to report it:

jira.cypress.com

Click on Create to start submitting a JIRA. Use the project type of "MTBIDE" or "BTSDK", depending on the issue and then fill in as many details as you can to report the issue.

Non-Cypress employees can ask questions and report issues in the developer community.

1.14 Tour of Bluetooth

Bluetooth is a short-range wireless standard that runs on the 2.4 GHz ISM (Industrial, Scientific, and Medical) band modulation. It is controlled by the Bluetooth Special Interest Group (SIG).

Discussions about Bluetooth are typically divided into *Classic Bluetooth* and *Bluetooth Low Energy*.

1.14.1 The Bluetooth Special Interest Group (SIG)

The Bluetooth Special Interest Group is an industry consortium that owns the specifications for Bluetooth. All the Bluetooth documentation is available at www.bluetooth.org. You can register for an account on that website.
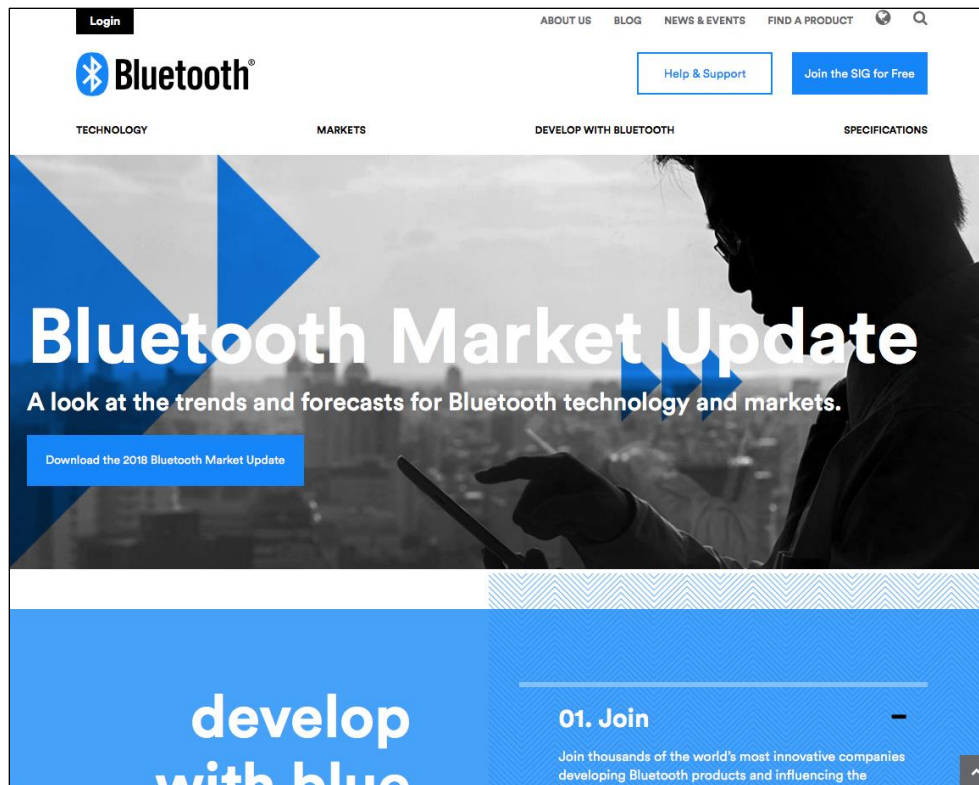
The current Bluetooth Specification is Version 5.1 is a ~3000 page long document that can be downloaded from the Bluetooth SIG website at https://www.bluetooth.com/specifications/bluetooth-core-specification

1.14.2 Classic Bluetooth

Classic Bluetooth uses 79 channels with a channel spacing of 1 MHz. It has three main speeds – Basic Rate (BR) and two Extended Data Rates (EDR). Each of these uses a different modulation scheme.

Mode	Speed	Modulation
Basic Rate	1 Mbps	GFSK (Gaussian Frequency Shift Keying)
Extended Data Rate	2 Mbps	π/4 DQPSK (Differential Quadrature Phase Shift Keying)
Extended Data Rate	3 Mbps	8DPSK (Octal Differential Phase Shift Keying)

The range is dependent on the transmission power which is divided into four classes:

Class	Max Permitted Power		Typical Range
	(mW)	(dBm)	(m)
1	100	20	100
2	2.5	4	10
3	1	0	1
4	0.5	-3	0.5

1.14.3 Bluetooth Low Energy

Bluetooth Low Energy (BLE) uses 40 channels with a channel spacing of 2 MHz (and so it shares the same range of frequencies with Bluetooth Classic). It provides much lower power consumption than Classic Bluetooth. Lower power is not achieved by reducing range (i.e. transmission power) but rather by staying actively connected for short bursts and being idle most of the time. This requires devices to agree on a connection interval. This connection interval can be varied to trade off the frequency of data transmitted vs. power. Therefore, BLE is excellent for data that can be sent in occasional bursts such as sensor states (i.e. temperature, state of a door, state of a light, etc.) but is not good for continuous streaming of data such as audio. BLE typically transmits data up to 1 Mbps, but 2 Mbps can be achieved in Bluetooth version 5 with shorter range.

1.14.4 Bluetooth History

Bluetooth Spec	Year	Major Features
1.0	1999	Initial standard.
1.1	2002	Many bug fixes. Addition of RSSI and non-encrypted channels.
1.2	2003	Faster connection and discovery. Adaptive Frequency Hopping (AFH) Host Control Interface (HCI) Addition of flow control and retransmission.
2.0 + EDR	2004	Addition of EDR (up to 3 Mbps).
2.1 + EDR	2007	Addition of Secure Simple Pairing (SPP) and enhanced security. Extended Inquiry Response (EIR).
3.0 + HS	2009	Addition of HS which uses Bluetooth for negotiation and establishment, then uses an 802.11 link for up to 24 Mbps. This is called Alternative MAC/PHY (AMP). Addition of Enhanced Retransmission Mode (ERTM) and Streaming Mode (SM) for reliable and unreliable channels.
4.0 + LE	2010	Addition of BLE. Addition of Generic Attribute Profile (GATT). Addition of Security Manager (SM) with AES encryption.
4.1	2013	Incremental software update.
4.2	2014	LE secure connections with data packet length extension. Link Layer privacy. Internet Protocol Support Profile (SPP) version 6.
5	2016	LE up to 2 Mbps for shorter range, or 4x range with lower data rate. LE increased packet lengths to achieve 8x data broadcasting capacity.
5.1	2019	Mesh-based model hierarchy Angle of Arrival (AoA) and Angle of Departure (AoD) fo rtracking Advertising Channel Index GATT Cacheing

1.15   Tour of Chips

The following shows a subset of the Bluetooth devivces available. See the cypress.com website for a comprehensive list of products.

| Device	Key Features	Software
CYW20706	• Bluetooth BR, EDR and LE 5.x • ARM Cortex-M3 • 848 kB ROM • 352 kB RAM (data and patches) • 2 kB NVRAM	WICED Studio
CYW20719	• Bluetooth BR, EDR and LE 5.x • 2 Mbps LE v5 • 96 MHz ARM Cortex-M4 • Single Precision FPU • 2 MB ROM • 1 MB On-Chip Flash • 512 kB RAM	ModusToolbox
CYW20819	• Bluetooth BR, EDR and LE 5.x • BR/EDR 2 Mbps and 3Mbps • LE 2Mbps • Ultra-low power • 96 MHz ARM Cortex-M4 • 1 MB ROM • 256 kB On-Chip Flash • 176 kB RAM	ModusToolbox
PSoC 4 BLE	• BLE 4.2 • 48 MHz ARM Cortex-M0 • 256 kB On-Chip Flash • 32 kB RAM	PSoC Creator
PSoC 6 BLE	• BLE 5.x • 150 MHz ARM Cortex-M4 & M0+ • 1MB or 2 MB On-Chip Flash • 288 KB RAM	ModusToolbox PSoC Creator
PSoC 6 + 43012 BT/WiFi	• Wi-Fi + Bluetooth Combo • Wi-Fi 802.11 ac-friendly • Bluetooth BR, EDR and LE 5.0	ModusToolbox

This class focusses on the CYW20819 Bluetooth SoC device with a minimal amount of information on the PSoC 6 + 43012 solution. It does not cover PSoC 4 BLE or PSoC 6 BLE devices.

The Cypress CYW20819 is an ultra-low power (ULP), highly integrated, and dual-mode Bluetooth wireless MCU. By leveraging the all-inclusive development platform ModusToolbox, it allows you to implement the industry's smallest-footprint, lowest-power Bluetooth Low Energy (BLE) and dual mode Bluetooth applications quickly. CYW20819 is a Bluetooth 5.x compliant SoC with support for Bluetooth Basic Rate (BR), Enhanced Data Rate (EDR), and BLE.

The CYW20819 employs the highest level of integration to eliminate all critical external components, thereby minimizing the device's footprint and the costs associated with implementing Bluetooth solutions. A 96 MHz CM4 CPU coupled with 256 kB on-chip flash and 1 MB ROM for stack and profiles

offers significant processing power and flash space to customers for their applications. CYW20819 is the optimal solution for a range of battery-powered single/dual mode Bluetooth internet of things applications such as home automation, HID, wearables, audio, asset tracking, and so on.

1.16 Tour of Partners
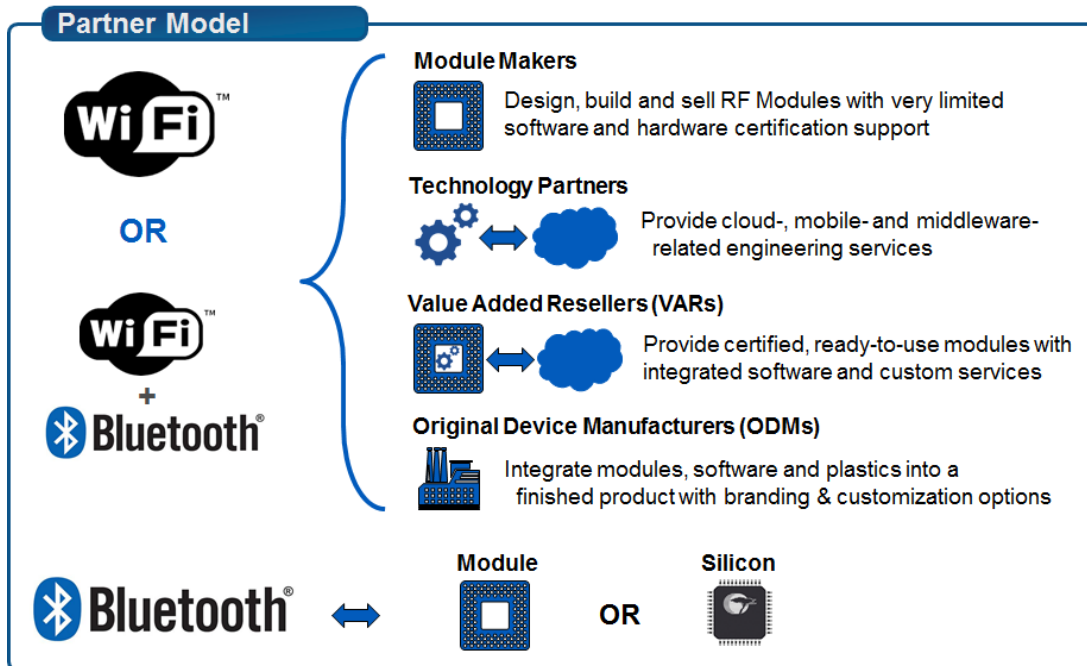
An IoT Selector Guide including partner modules available can be found in the Community at:

https://community.cypress.com/docs/DOC-3021

1.17 Tour of Development Kits

1.17.1 Cypress CYW920819EVB-02

- Bluetooth 5.x plus 2 Mbps LE from v5
- 96 MHz ARM Cortex-M4
- Integrated transceiver
- 1 MB ROM, 256 kB On-Chip Flash, 176 kB SRAM
- 1 User Button, 2 User LEDs
- USB JTAG Programmer/Debugger

1.17.2 Cypress CYBT-213043-MESH

- Bluetooth Mesh kit with 20819 module
- Each kit contains 4 boards to evaluate mesh networks
- 1 User Button, RGB LED, ambient light sensor, PIR motion sensor

1.17.3 Cypress CYW920719Q40EVB-01

- Bluetooth 5.x plus 2 Mbps LE from v5
- 96 MHz ARM Cortex-M4
- Integrated transceiver
- 2 MB ROM, 1 MB On-Chip Flash, 512 kB SRAM
- 1 User Button, 2 User LEDs
- USB JTAG Programmer/Debugger

1.17.4 Cypress CY8CKIT-062S2-43012

- Murata 1LV Module containing the CYW43012 Wi-Fi + Bluetooth Combo Chip
- PSoC 62 MCU with 150 MHz Cortex-M4 and 100 MHz Cortex M3
- 2 MB On-Chip Flash, 1 MB SRAM
- 2 User Buttons, 5 User LEDs
- USB JTAG Programmer/Debugger

1.18   Exercise(s)

Exercise - 1.1 Create a Forum Account

1. Go to https://community.cypress.com/welcome

2. Click **Log in** from the top right corner of the page and login to your Cypress account. If you do not have an account, you will need to create one first.

3. Once you are logged in, click the **Wireless** icon and then explore.

Exercise - 1.2 Start the Eclipse IDE for ModusToolbox, and Explore the Documentation

1. Run the Eclipse IDE and create a new workspace.

2. *Explore the different documents available in the Help menu such as the Eclipse IDE for ModusToolbox Help, Quick Start Guide, User Guide* and *Eclipse IDE Survival Guide*.

3. Open and explore the *ModusToolbox Documentation Index* page. Be sure to look at the **CYW920819EVB-02 Kit** link and the **Bluetooth Documentation > CYW208XX API Reference**.
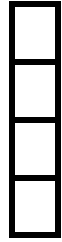
Questions to answer:

1. Where is the WICED API documentation for the PWM located?

Exercise - 1.3 Program a Simple Application

In this exercise, you will create a new application, and then build/program it to a kit.

1. In the Quick Panel tab click **New Application**.

2. Select the CYW920819EVB-02 kit and click **Next >**.

3. Check the box next to the "Empty BTSDK App" application template.

4. Change the application name to **ch01_ex01_led** and click **Create**.
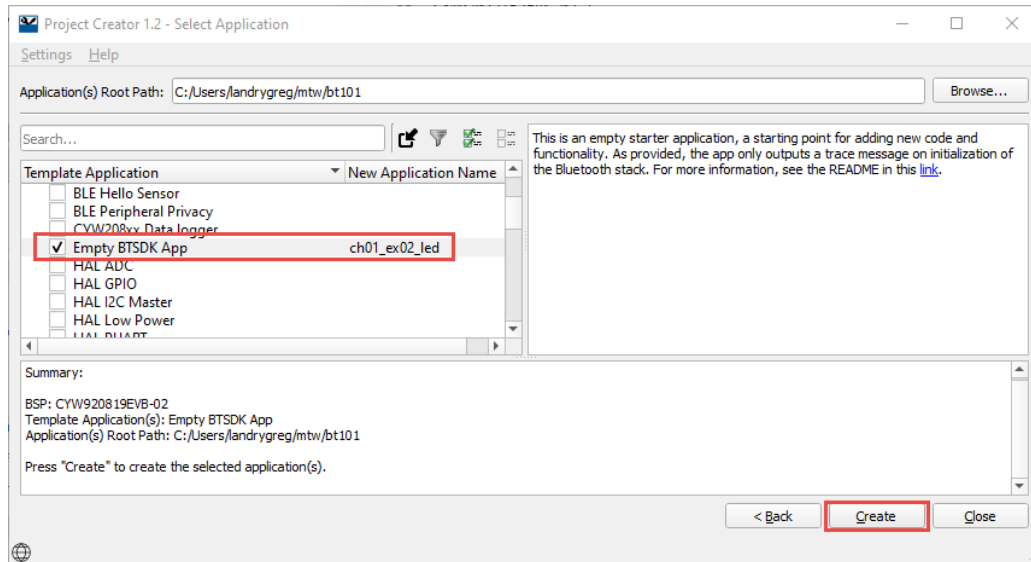
Note You can call the application anything you like but it will really help if you maintain an alphabetically sortable naming scheme – you are going to create quite a few projects.
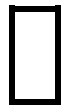
5. Once the ch01_ex01_led application has been created, it will be imported into the Eclipse IDE.

When that's done, the Project Explorer window in Eclipse should look like the following:
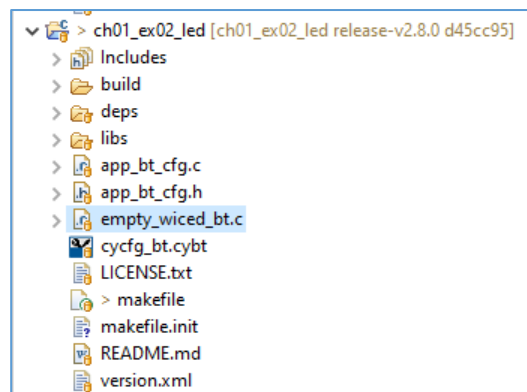
6. Open the top-level C file which in this case is called *empty_wiced_bt.c*.

7. Add the header to get access to the Device Configurator GeneratedSource:

   #include "cycfg.h"

8. Add the following line in the `application_start` function where it reads:
   `/* TODO your app init code */`:

   wiced_hal_gpio_set_pin_output(LED1, *LED_STATE_ON*);

9. Connect your CYW920819EVB-02 kit to a USB port on your computer.

10. In the Quick Panel, click the **ch01_ex01_led Program** link under "Launches."

11. Once the build and program operations are done, you should see "Programming complete" in the Console window and LED1 (the yellow user LED) should be on.

Exercise - 1.4 Use the Command Line

In this exercise you will use the command line to modify/build/program the same application as the previous exercise.

1. Open modus-shell (Windows) or a Terminal (MacOS and Linux).

2. Go to the directory containing the application from the previous exercise.

3. Edit the file *empty_wiced_bt.c* with a file editor of your choice to change to turn on LED2 (the red user LED).

4. Run the appropriate make command to build and program your kit.

Exercise - 1.5 (Advanced) Use Visual Studio Code

In this exercise you will use Visual Studio Code to modify/build/program the same application as the previous exercise.

1. Install VS Code if you haven't already.

2. Open VS Code and install the C/C++ and Cortex-Debug extensions is you haven't already.

3. Open ModusShell (Windows) or a Terminal (MacOS and Linux).

4. Go to the directory containing the application from the previous exercise.

5. Run the `make vscode` command to create the VS Code files.

6. Open the workspace in VS Code.

7. Edit the file *empty_wiced_bt.c* to change the LED behavior somehow.

8. Run the appropriate build task to build and program your kit.