

# Chapter 1: Tour

Time 1 Hour

After completing this chapter, you will understand a top-level view of the PSoC 6 ecosystem components, including the chips, modules, software, documentation, support infrastructure and development kits.

<b>1.1</b>	<b>OVERVIEW .....</b>	<b>3</b>
1.1.1	MAKE/BUILD INFRASTRUCTURE .....	3
<b>1.2</b>	<b>MODUSTOOLBOX ONLINE.....</b>	<b>4</b>
1.2.1	CYPRESS.COM .....	4
1.2.2	COMMUNITY PAGE.....	5
1.2.3	GITHUB.COM .....	5
1.2.4	NETWORK CONSIDERATIONS/NETWORK PROXY SETTINGS .....	6
1.2.5	OFFLOADING MANIFEST FILES .....	6
1.2.6	OFFLINE CONTENT .....	7
<b>1.3</b>	<b>ECLIPSE IDE FOR MODUSTOOLBOX.....</b>	<b>8</b>
1.3.1	FIRST LOOK.....	8
1.3.2	CUSTOMIZATION .....	10
1.3.3	PROJECT EXPLORER .....	10
1.3.4	QUICK PANEL.....	11
1.3.5	INTEGRATED DEBUGGER.....	12
1.3.6	DOCUMENTATION .....	13
1.3.7	ECLIPSE IDE TIPS & TRICKS .....	14
<b>1.4</b>	<b>PROJECT DIRECTORY ORGANIZATION .....</b>	<b>14</b>
1.4.1	BINARIES.....	15
1.4.2	INCLUDES .....	15
1.4.3	BUILD .....	15
1.4.4	DEPS .....	15
1.4.5	IMAGES.....	15
1.4.6	LIBS .....	15
1.4.7	MAIN.C .....	15
1.4.8	LICENSE.....	15
1.4.9	MAKEFILE .....	16
1.4.10	MAKEFILE.INIT .....	17
1.4.11	README.MD .....	17
<b>1.5</b>	<b>LIBRARY MANAGEMENT .....</b>	<b>17</b>
1.5.1	LIBRARY CLASSIFICATION .....	17
1.5.2	LIBRARY MANAGEMENT FLOWS .....	18
<b>1.6</b>	<b>MANIFEST FILES.....</b>	<b>20</b>
<b>1.7</b>	<b>BOARD SUPPORT PACKAGES .....</b>	<b>22</b>
1.7.1	BSP DIRECTORY STRUCTURE.....	22
1.7.2	BSP DOCUMENTATION.....	24
1.7.3	MODIFYING THE BSP CONFIGURATION (E.G. DESIGN.MODUS) FOR A SINGLE APPLICATION .....	25
1.7.4	CREATING YOUR OWN BSP.....	26
<b>1.8</b>	<b>TOOLS.....</b>	<b>28</b>
1.8.1	PROJECT CREATOR.....	29
1.8.2	LIBRARY MANAGER .....	32
1.8.3	MODUSTOOLBOX CONFIGURATORS .....	33
1.8.4	BSP CONFIGURATORS .....	34

1.8.5	APPLICATION CONFIGURATORS.....	38
1.8.6	COMMAND LINE .....	39
<b>1.9</b>	<b>VISUAL STUDIO CODE.....</b>	<b>41</b>
<b>1.10</b>	<b>REPORTING ISSUES.....</b>	<b>42</b>
<b>1.11</b>	<b>TOUR OF WI-FI .....</b>	<b>42</b>
<b>1.12</b>	<b>TOUR OF CHIPS .....</b>	<b>43</b>
<b>1.13</b>	<b>TOUR OF PARTNERS.....</b>	<b>44</b>
<b>1.14</b>	<b>TOUR OF DEVELOPMENT KITS.....</b>	<b>45</b>
1.14.1	CYPRESS CY8CKIT-062S2-43012.....	45
1.14.2	CYPRESS CY8CKIT-062-WiFi-BT .....	45
1.14.3	CYPRESS CYW943907AEVAL1F .....	45
1.14.4	CYPRESS CYW94343WWCD1_EVB EVALUATION AND DEVELOPMENT KIT .....	46
1.14.5	FUTURE NEBULA IoT DEVELOPMENT KIT .....	46
1.14.6	ARROW QUADRO IoT Wi-Fi KIT .....	46
1.14.7	ARROW QUICKSILVER IoT KIT.....	47
1.14.8	AVNET BCM4343W IoT STARTER KIT.....	47
1.14.9	ADAFRUIT FEATHER .....	47
1.14.10	ELECTRIC IMP.....	47
1.14.11	PARTICLE PHOTON .....	48
1.14.12	SPARKFUN WITH PARTICLE PHOTON MODULE .....	48
1.14.13	INVENTEK .....	48
<b>1.15</b>	<b>EXERCISE(S) .....</b>	<b>49</b>
1.15.1	EXERCISE 1: CREATE A FORUM ACCOUNT.....	49
1.15.2	EXERCISE 2: INSTALL MODUSTOOLBOX.....	49
1.15.3	EXERCISE 3: OPEN THE DOCUMENTATION .....	50

## 1.1 Overview

This chapter provides a brief tour of the ModusToolbox websites, Eclipse IDE, and various tools as well as a tour of Wi-Fi in general and our chips, partners, and development kits.

ModusToolbox is a set of Reference Flows, Products, and Solutions that enable an immersive development experience for customers creating converged MCU and Wireless systems.

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. There is a command line interface (CLI) that can be used interchangeably with IDE applications. In addition, users can use ModusToolbox software with their own preferred IDE, such as Visual Studio Code or IAR.

For Windows, ModusToolbox software is installed, by default, in *C:/Users/<UserName>/ModusToolbox*. Once installed, the IDE will show up in Windows under **Start > All Programs > ModusToolbox <version>**.

**Note** This class focusses on the AnyCloud Wi-Fi solution. ModusToolbox is introduced since you will use it for creating Wi-Fi applications, but many details are not covered. Refer to the ModusToolbox 101 class for many more details, tips, and tricks for working with ModusToolbox including using other IDEs such as Visual Studio Code.

### 1.1.1 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
- Configurator tools
- Library manager
- New project GUI and CLI utilities
- Modus-Shell
- Eclipse IDE for ModusToolbox

These tools can be found in the ModusToolbox installation folder 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.

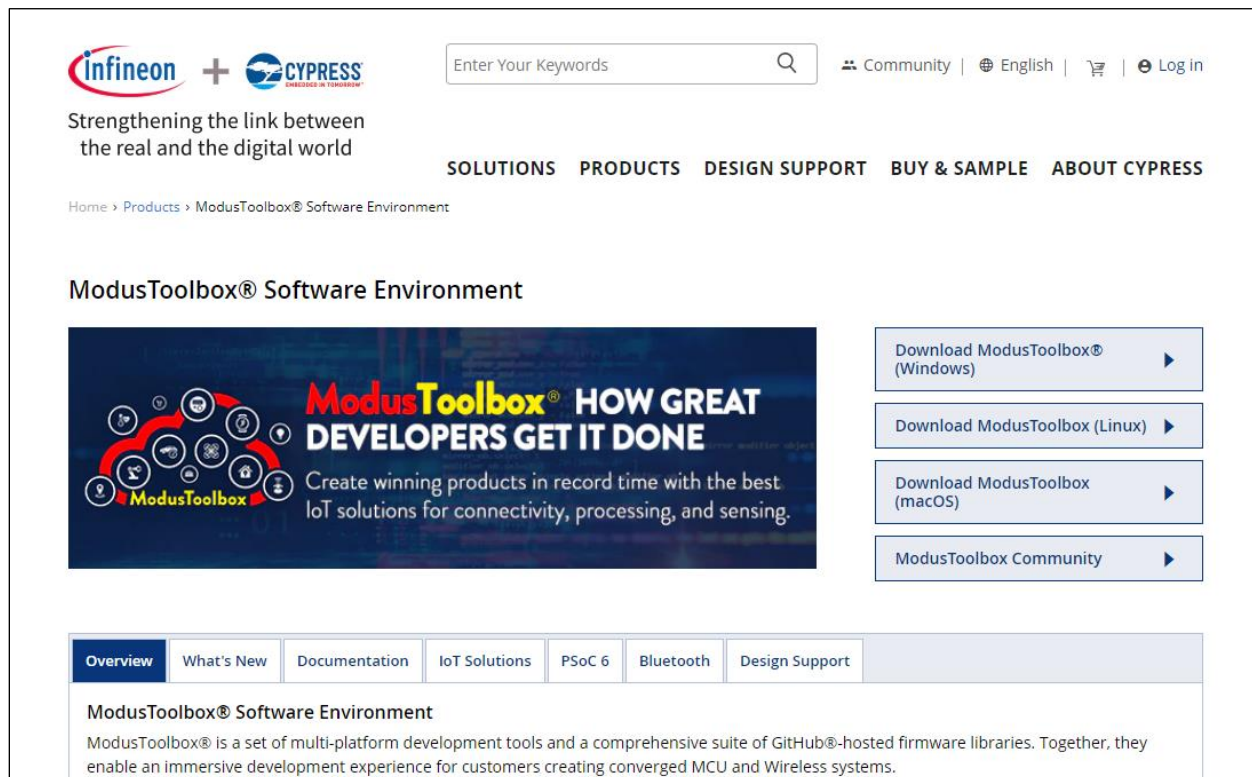
- Create the application using the project creator tool (this step downloads the application itself and any required libraries)
- Configure the device and its peripherals using configurators
- Write the application code
- Build/Program/Debug

## 1.2 ModusToolbox Online

### 1.2.1 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>



The screenshot shows the Cypress website's page for the ModusToolbox Software Environment. At the top, there is a navigation bar with the Infineon and Cypress logos, a search bar, and links for Community, English, and Log in. Below the navigation bar, the page title "ModusToolbox® Software Environment" is displayed. A large banner image features the text "ModusToolbox® HOW GREAT DEVELOPERS GET IT DONE" and "Create winning products in record time with the best IoT solutions for connectivity, processing, and sensing." To the right of the banner are four buttons: "Download ModusToolbox® (Windows)", "Download ModusToolbox (Linux)", "Download ModusToolbox (macOS)", and "ModusToolbox Community". Below the banner is a tabbed interface with tabs for Overview, What's New, Documentation, IoT Solutions, PSoc 6, Bluetooth, and Design Support. The Overview tab is selected, showing a description of the ModusToolbox Software Environment.

Strengthening the link between the real and the digital world

SOLUTIONS PRODUCTS DESIGN SUPPORT BUY & SAMPLE ABOUT CYPRESS

Home > Products > ModusToolbox® Software Environment

**ModusToolbox® Software Environment**

**ModusToolbox® HOW GREAT DEVELOPERS GET IT DONE**

Create winning products in record time with the best IoT solutions for connectivity, processing, and sensing.

Download ModusToolbox® (Windows)

Download ModusToolbox (Linux)

Download ModusToolbox (macOS)

ModusToolbox Community

Overview What's New Documentation IoT Solutions PSoc 6 Bluetooth Design Support

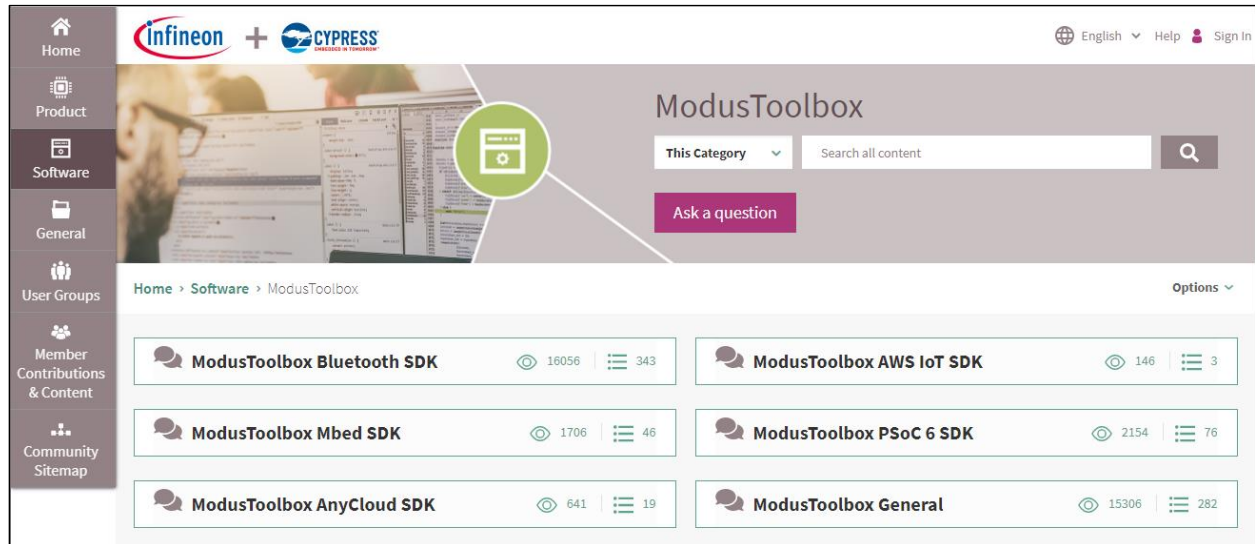
**ModusToolbox® Software Environment**

ModusToolbox® is a set of multi-platform development tools and a comprehensive suite of GitHub®-hosted firmware libraries. Together, they enable an immersive development experience for customers creating converged MCU and Wireless systems.

## 1.2.2 Community Page

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

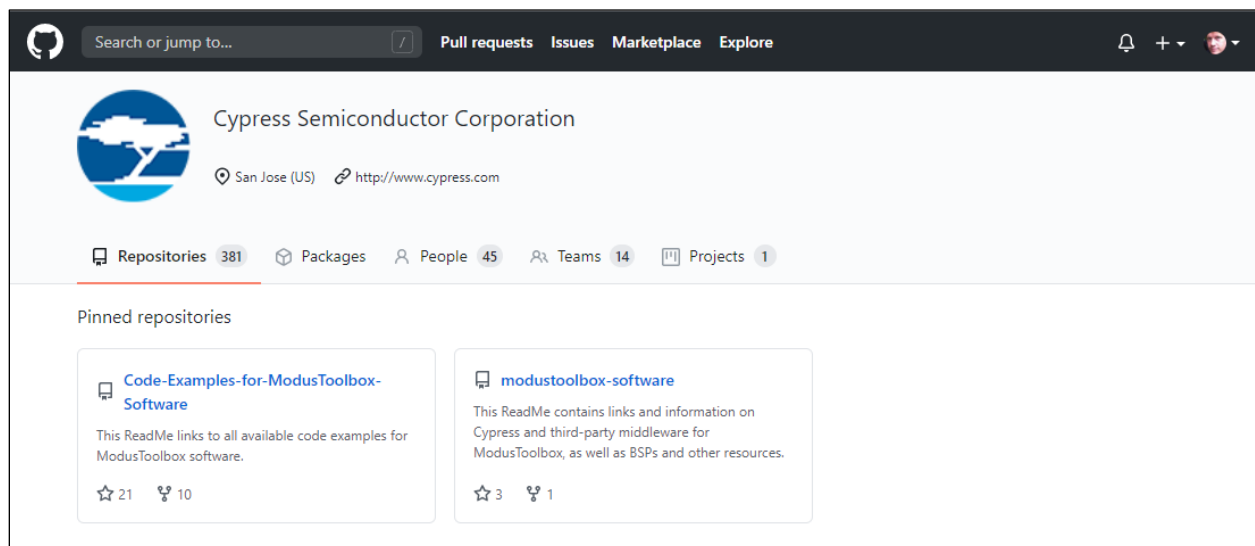
<https://community.cypress.com/t5/ModusToolbox/ct-p/ModusToolbox>



## 1.2.3 Github.com

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

<https://github.com/cypresssemiconductorco>



### 1.2.4 Network Considerations/Network Proxy Settings

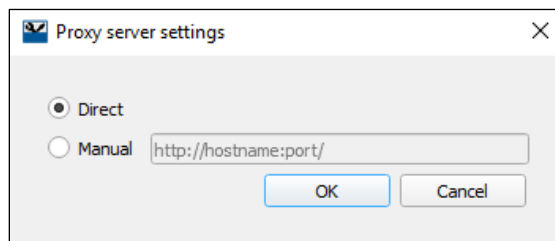
When you run the Project Creator or Library Manager, the first thing it does is look for a remote manifest file. If that file isn't found, you will not be able to go forward. In some cases, it may find the manifest but then fail during the project creation step (during git clone). If either of those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

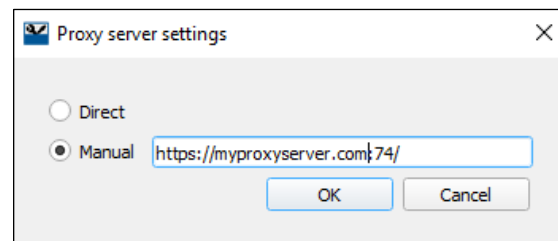
To view/set proxy settings in the Project Creator or Library Manager, use the menu option **Settings > Proxy Settings...**

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 time you connect to a network requiring that proxy server. The tool will remember your last server name.

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

### 1.2.5 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>

## 1.2.6 Offline Content

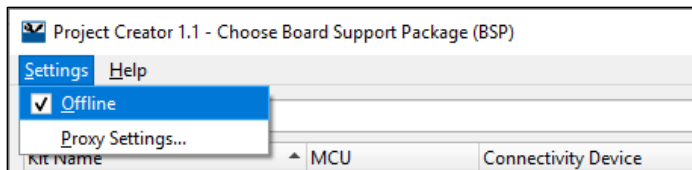
Beginning with ModusToolbox version 2.1, Cypress provides a zipped-up bundle of the GitHub repos to allow you to work offline, such as on an airplane or if for some reason you don't have access to GitHub. To set this up, you must have access to [cypress.com](https://www.cypress.com) in order to download the zip file. After that, you can work offline.

Go to [cypress.com/modustoolbox-offline-content](https://www.cypress.com/modustoolbox-offline-content) and download the *modustoolbox-offline-content-x.x.x.<build>.zip* file.

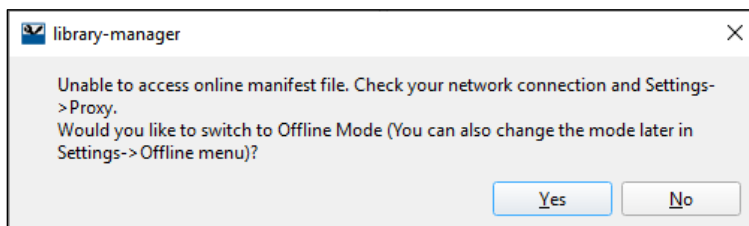
Extract the "offline" directory under the location where you have ModusToolbox installed in a hidden directory named *.modustoolbox*. In most cases, this is in your User Home directory. After extracting, the path should be similar to this:

*C:\Users\<username>\.modustoolbox\offline*

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.

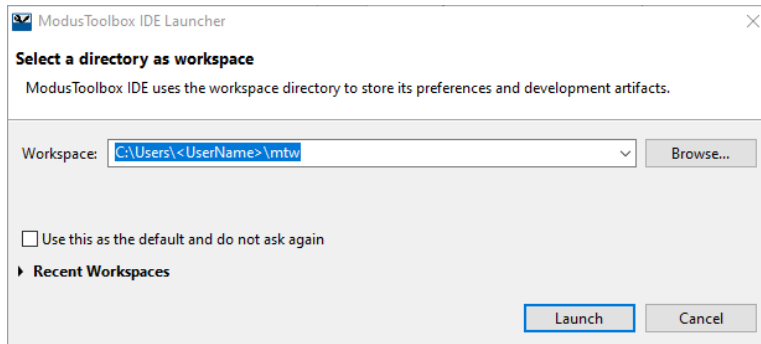


## 1.3 Eclipse IDE for ModusToolbox

### 1.3.1 First Look

ModusToolbox software includes an Eclipse based IDE. It 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.

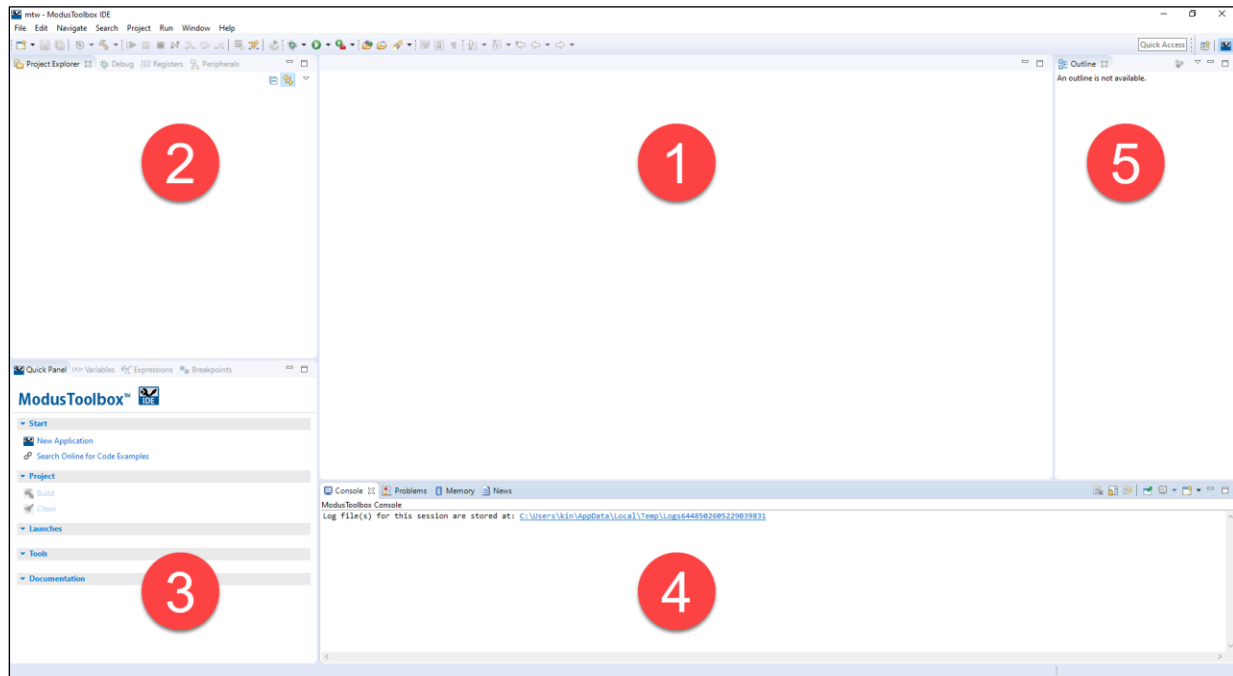


**Note** Each version of ModusToolbox installs alongside previous versions; therefore, each one can be used independently. Be aware that the default workspace location (*C:\Users\<UserName>\mtw* on Windows) is the same in all versions of the Eclipse IDE. If you plan to use more than one version, you should specify different workspace names for each installation.

If you want to switch to a different workspace, you can use the menu item **File > Switch Workspace**. You can select a recent workspace from the menu or use **Other...** to specify a different path.

After clicking **Launch**, the IDE will start. When you open a new workspace, the first window you see will be the empty ModusToolbox perspective.





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**.

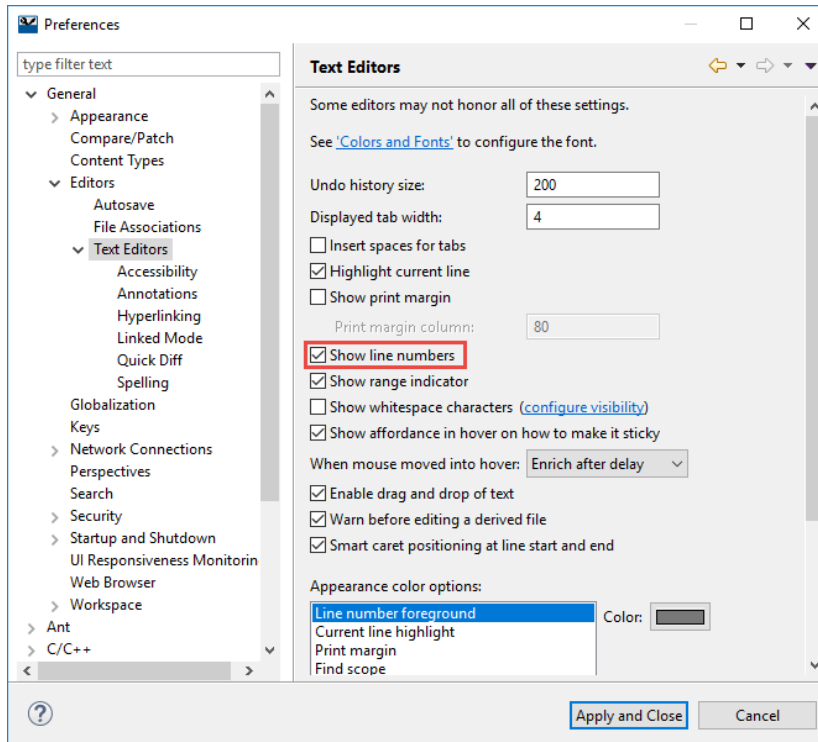
The major views are:

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

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

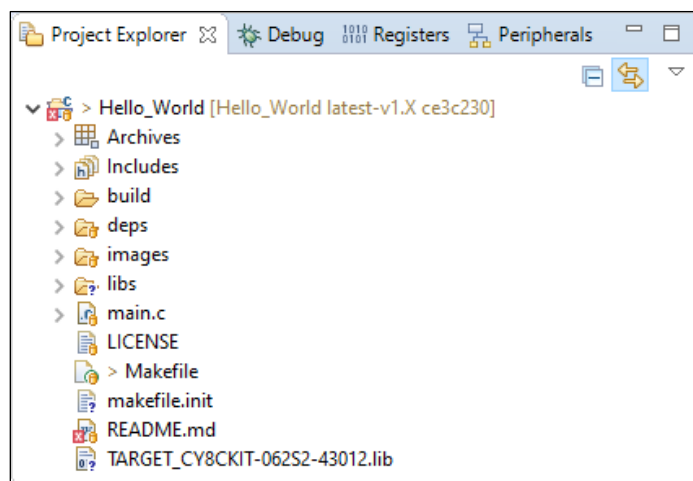
### 1.3.2 Customization

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**. A setting you may want to turn on is **Show line numbers** via **General > Editors > Text Editor**. (You can also enable line numbers by right-clicking along the left edge of the file editor window). Most settings are saved at the workspace level. For more information you can read the [Eclipse IDE Survival Guide](#).



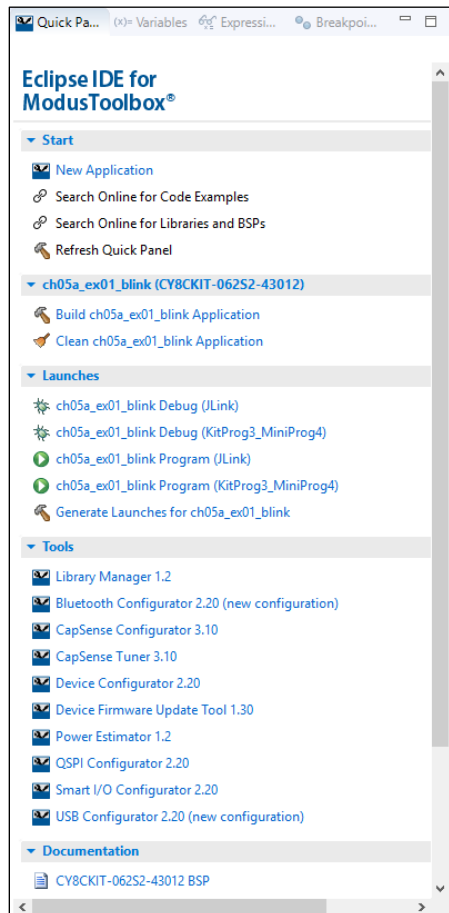
### 1.3.3 Project Explorer

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



### 1.3.4 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 in the menus. There are top level commands, application level commands, launches, 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 board support package (BSP) and then one or more starter applications. One or both of them can be imported if you have your own custom BSP or template application. Once the selections are done, 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 (such as Erase) in the Quick Panel that can be accessed from **Run > Run 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 and peripherals. Some of these will be covered in [Tools](#).

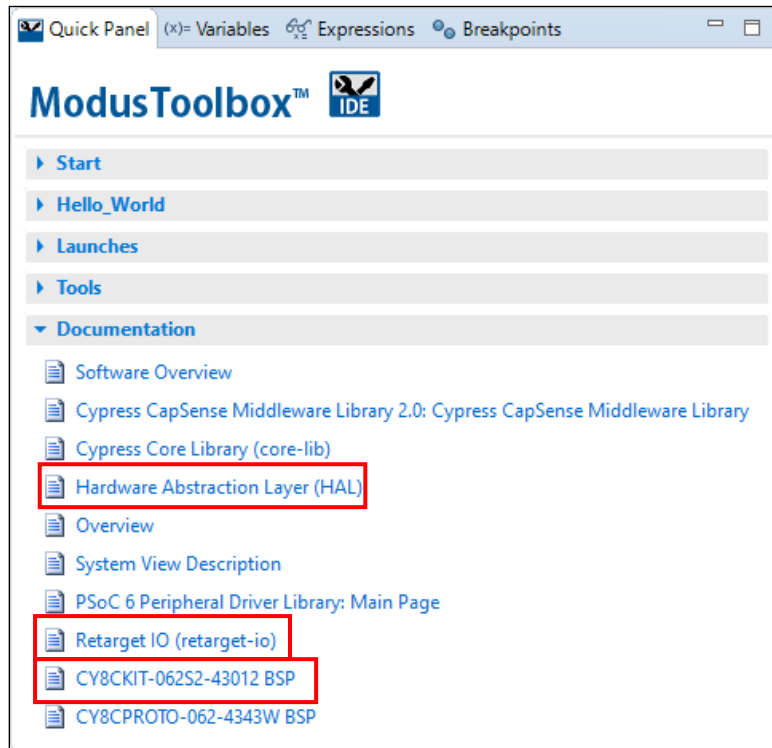
### 1.3.5 Integrated Debugger

The Eclipse IDE provides an integrated debugger using either the KitProg3 (OpenOCD) on the PSoC 6 development kit or using a separate MiniProg4. It also allows you to debug using a Segger J-Link debug probe.

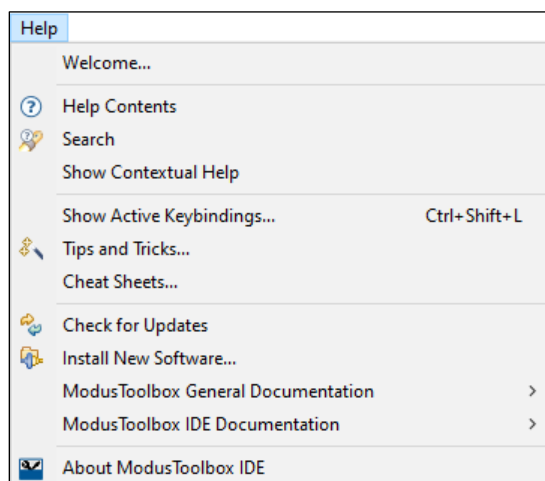


### 1.3.6 Documentation

After creating a project, assorted links to documentation are available directly from the Quick Panel, under the “Documentation” section. This will update to include documentation for new libraries as you add them to the application. The HAL, BSP, and various library documentation links are particularly useful for finding API usage. Most of them include a quick start section and a code snippets to get you going.



The Help menu provides links to general documentation, such as the [ModusToolbox User Guide](#) and [Release Notes](#), as well as IDE-specific documentation, such as the [Eclipse IDE for ModusToolbox Quick Start Guide](#).



### 1.3.7 Eclipse IDE Tips & Tricks

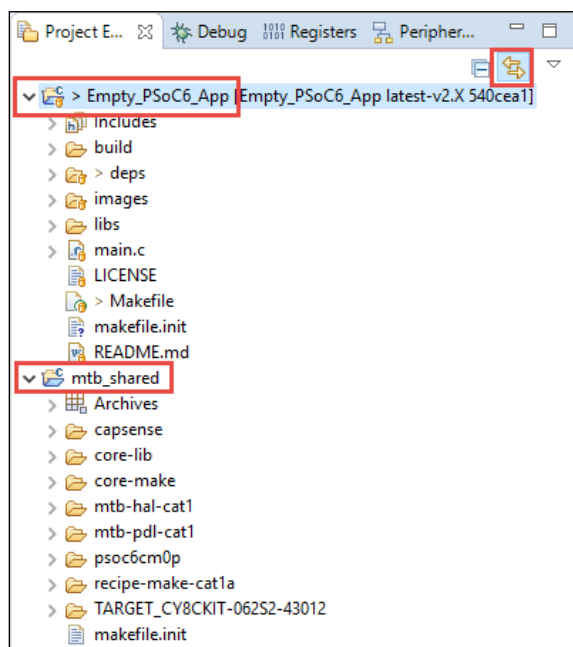
Eclipse has several quirks that new users may find hard to understand at first. Here are a few tips to make the experience less difficult:

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

Refer also to our [Eclipse Survival Guide](#) for more tips and tricks.

## 1.4 Project Directory Organization

Once created, a typical ModusToolbox PSoC 6 project contains various files and directories.



**Hint** if you are going back and forth between files from two different applications (e.g. to copy code from one application to another), switching between files can be greatly speeded up by turning off the "Link with Editor" feature. This prevents the active project from being changed every time you switch between files. It can be turned off using the icon with two arrows shown inside the red box above.

There are two top-level directories associated with most applications - the application project itself (Empty\_PSoC6\_App in this example) and a location that contains shared library source code (mtb\_shared).

The directories in the application's project include:

#### 1.4.1 Binaries

Virtual directory that points to the elf file from the build. This directory will not appear until a build is done.

#### 1.4.2 Includes

Virtual directory that shows the include paths used to find header files.

#### 1.4.3 build

This directory contains build files, such as the *.elf* and *.hex* files. It will not appear until a build is done.

#### 1.4.4 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).

#### 1.4.5 images

This directory contains artwork images used by the documentation.

#### 1.4.6 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.

#### 1.4.7 main.c

This is the primary application file that contains the project's application code.

#### 1.4.8 LICENSE

This is a software license agreement file.

### 1.4.9 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 Eclipse IDE and the Command Line Interface (CLI) so once you create an application, you can go back and forth between the 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=**)

```
31# Target board/hardware (BSP).
32# To change the target, use the Library manager ('make modlibs' from command line).
33# If TARGET is manually edited, ensure TARGET_<BSP>.lib with a valid URL exists
34# in the application, and run 'make getlibs' to fetch BSP contents.
35 TARGET=CY8CKIT-062S2-43012
```

#### Build Configuration (**CONFIG=**)

```
49# Default build configuration. Options include:
50#
51# Debug -- build with minimal optimizations, focus on debugging.
52# Release -- build with full optimizations
53# Custom -- build with custom configuration, set the optimization flag in CFLAGS
54
55 CONFIG=Debug
--
```

**Note** TARGET and CONFIG are used in the launch configurations (program, debug, etc.) so if you change either of these variables manually in the *Makefile*, you must update or regenerate the Launch configs inside the Eclipse IDE. 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*, you must add the .mtb file for the new BSP 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.



## Adding Components (`COMPONENTS=`)

```

61 #####
62 # Advanced Configuration
63 #####
64
65 # Enable optional code that is ordinarily disabled by default.
66 #
67 # Available components depend on the specific targeted hardware and firmware
68 # in use. In general, if you have
69 #
70 #     COMPONENTS=foo bar
71 #
72 # ... then code in directories named COMPONENT_foo and COMPONENT_bar will be
73 # added to the build
74 #
75 COMPONENTS=
76
77 # Like COMPONENTS, but disable optional code that was enabled by default.
78 DISABLE_COMPONENTS=

```

### 1.4.10 Makefile.init

This file contains variables used by the make process. This is a legacy file that is not used anymore.

### 1.4.11 README.md

Almost every code example / starter application includes a *README.md* (mark down) file that provides a high-level description of the project.

## 1.5 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.

### 1.5.1 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 either 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.

### 1.5.2 Library Management Flows

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 Board Support Packages (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.

Looking ahead, most example applications will use the new MTB flow. However, there are still various applications that use the previous flow, now called the LIB flow, and these applications generally do not share BSPs and libraries. ModusToolbox fully supports both flows.

Note that the flow selection for a single application is a binary choice. If an application uses the MTB flow, only *mtb* files will be processed, and any *lib* files will be ignored.

### MTB flow

First let's discuss the flow using *mtb* files. In this flow, the source code for all libraries will be put in a shared directory by default. (You'll see how to make them local using in the [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`.

The `CY_GETLIBS_SHARED_PATH` variable will most commonly be changed for bundled apps. For most applications, the path is set to `../` to locate the directory one level up from the application directory (i.e. parallel to it). In the case of bundled applications, you will have one or more additional levels of hierarchy. If you have one extra level of hierarchy, the path would typically be set to `../../` so that the shared directory is in the usual place in the workspace.

Source code for local libraries will be placed in `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.

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 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
<b>.mtb file</b>	./deps/	./deps/	./libs/
<b>library source code</b>	../mtb_shared/	./libs/	../mtb_shared/

## LIB flow

Now, let's discuss the flow using *lib* files for completeness since there are still some applications that use this flow. In the LIB flow, all libraries are local to the app by default. ModusToolbox knows about a library in your project by having a lib file somewhere in the directories of your project (typically in the deps directory).

A *lib* file looks just like an *mtb* file except that the file extension is *.lib* and the path to the shared location is not specified. Instead it just has 2 fields:

- 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 typical library file will look something like this:

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

The `make getlibs` process (either called directly from the command line or by the Library Manager) pulls the libraries from the specified Git repo, but in this case, it searches for *lib* files instead of *mtb* files. The other difference is that all the library source code goes in the *libs* directory in the application itself instead of in a shared location. The *lib* files are found and processed recursively so that if a library that gets pulled in has dependencies, those dependencies are pulled in during the next pass.

Again, 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 lib files by re-running `make getlibs`.

## 1.6 Manifest Files

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 BSP packages 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 and 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:/MyManifests/my-super-manifest.xml
```

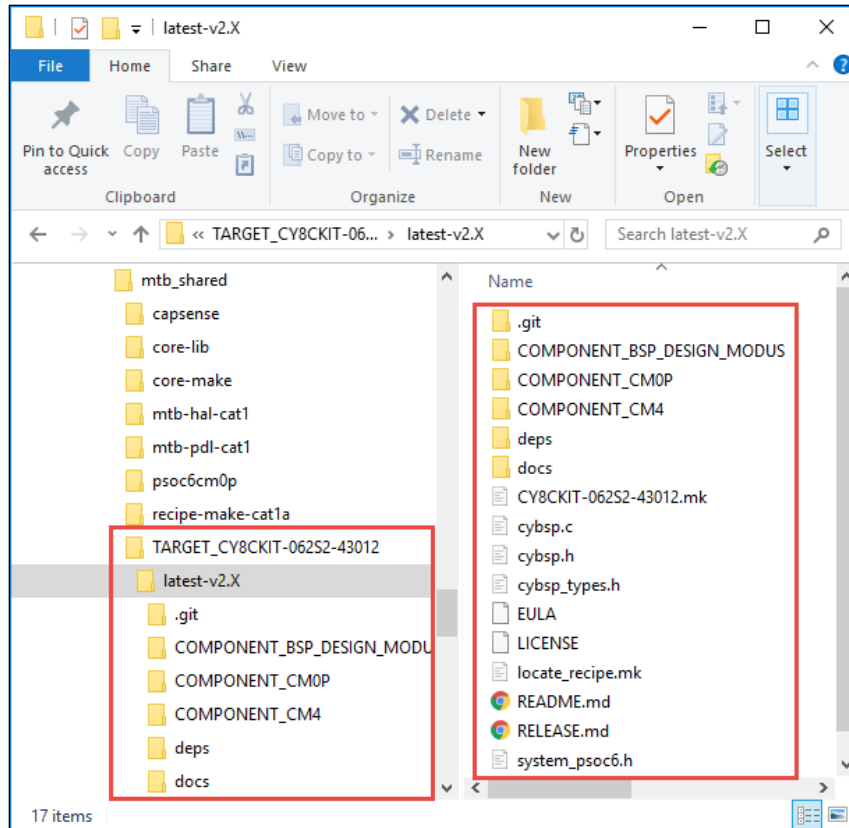
To see examples of the syntax of super-manifest and manifest files, you can look at the GitHub repos listed below. Each repo may hold several manifest file versions – be sure to look at the latest version.

- 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>

## 1.7 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” and can be seen in the *mtb\_shared* directory (by default if using the MTB flow):

### 1.7.1 BSP Directory Structure

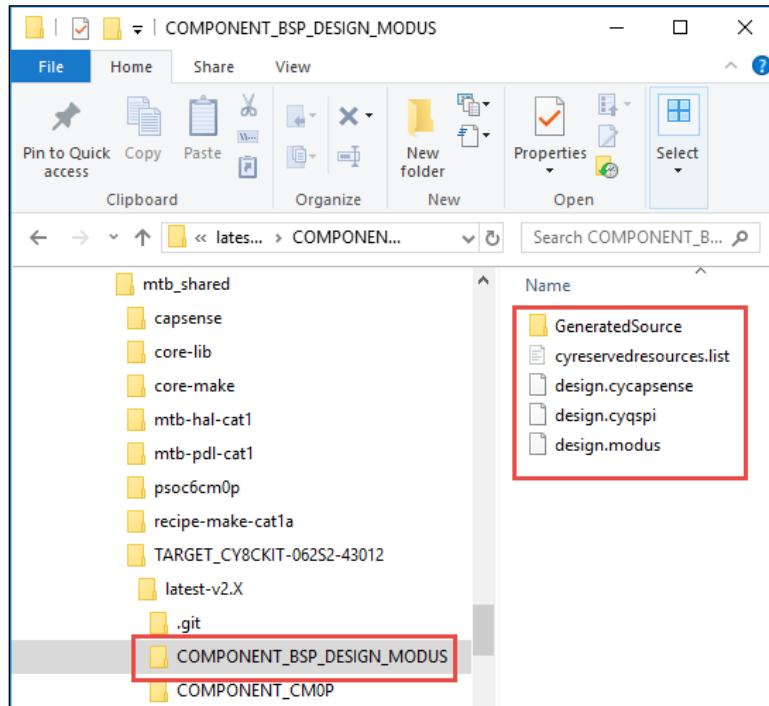


#### COMPONENT\_BSP\_DESIGN\_MODUS

This directory contains the configuration files (such as *design.modus*) for use with various BSP configurator tools, including the Device Configurator, QSPI Configurator, and CapSense Configurator. At the start of a build, the build system invokes these tools to generate the source files in the *GeneratedSource* directory.

Note that since these files are part of the BSP, if you use a configurator to modify them you will be modifying them for all applications that use the BSP (if it is shared). In addition, your changes will cause the BSP's repo to become dirty which means it cannot be updated to newer versions. Therefore, it is not recommended that you change any settings that are in the BSP. Instead, you can create a custom configuration for a single application (see [Modifying the BSP Configuration \(e.g. design.modus\) for a Single Application](#)) or you can create a custom BSP (see [Creating your Own BSP](#)).

A typical COMPONENT\_BSP\_DESIGN\_MODUS directory looks like this:



## COMPONENT\_CM4 and COMPONENT\_CM0P

These directories contain startup code and linker scripts for all supported toolchains for each of the two cores - the CM4 and the CM0+.

## docs

The *docs* directory contains the HTML based documentation for the BSP. See [BSP Documentation](#).

## deps

The *deps* directory contains *lib* files for libraries that the BSP requires as dependencies. These are only used for the application flow that uses *lib* files. For the flow that uses *mtb* files, the dependency information is contained in a manifest file and the *lib* files are ignored.

## cybsp\_types.h

The *cybsp\_types.h* file contains the aliases (macro definitions) for the board resources. It also contains comments for standard pin names so that they show up in the BSP documentation.

## cybsp.h / cybsp.c

These files contain the API interface to the board's resources.

You need to include only *cybsp.h* in your application to use all the features of a BSP. Call the `cybsp_init` function from your code to initialize the board (that function is defined in *cybsp.c*).

## <targetname>.mk

This file defines the DEVICE and other BSP-specific make variables such as COMPONENTS.

## locate\_recipe.mk

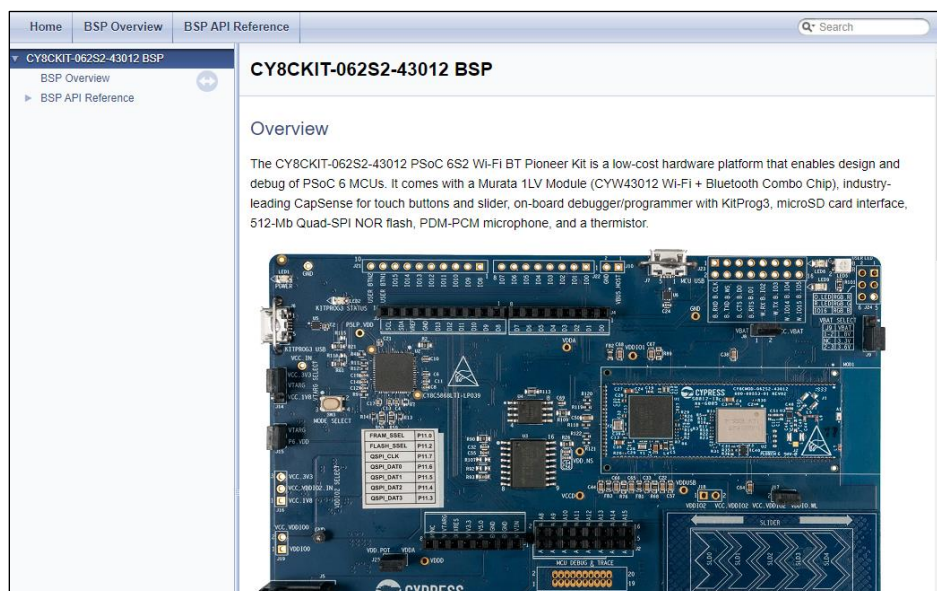
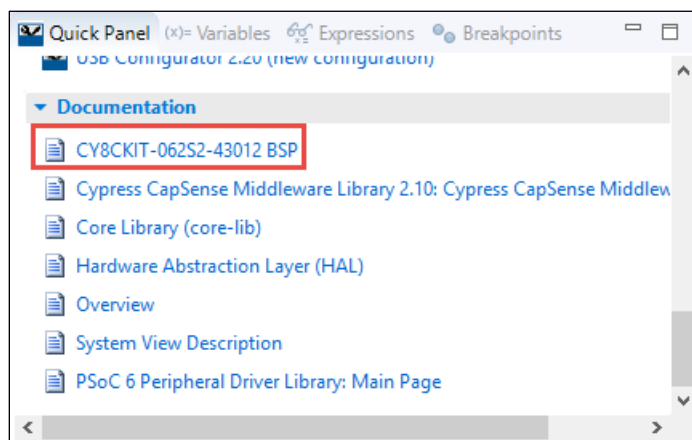
This file provides the path to the core and recipe make files that are needed by the build system.

## system\_psoc6.h

This file contains device system header information.

### 1.7.2 BSP Documentation

Each BSP provides HTML documentation that is specific to the selected board. It also includes an API Reference and the BSP Overview. After creating a project, there is a link to the BSP documentation in the IDE Quick Panel. As mentioned previously, this documentation is located in the BSP's *docs* directory.





### 1.7.3 Modifying the BSP Configuration (e.g. `design.modus`) for a Single Application

If you want to modify the BSP configuration for a single application (such as different pin or peripheral settings), you should not modify the BSP directly since that results in changes to the BSP library. This will affect other applications in the same workspace if the BSP is shared, and it will prevent you from updating the BSP repository in the future. Instead, use the following process to create a custom set of configuration files for a specific application:

1. Create a directory at the root of the application to hold any custom BSP configuration files. For example:

`Hello_World/COMPONENT_CUSTOM_DESIGN_MODUS`

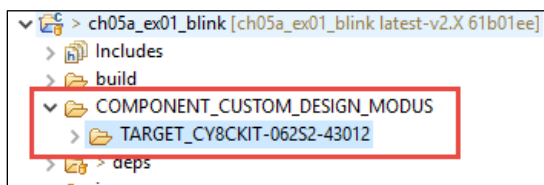
This is a recommended best practice. In an upcoming step, you will modify the *Makefile* to include files from that directory instead of the directory containing the default configuration files in the BSP.

2. Create a subdirectory for each target that you want to support in your application. For example:

`Hello_World/COMPONENT_CUSTOM_DESIGN_MODUS/TARGET_CY8CKIT-062S2-43012`

The subdirectory name must be `TARGET_<board name>`. Again, this is a recommended best practice. If you only ever build with one BSP, this directory is not required, but it is safer to include it.

The build system automatically includes all source files inside a directory that begins with `TARGET_`, followed by the target name for compilation, when that target is specified in the application's *Makefile*. The file structure appears as follows. In this example, the `COMPONENT_BSP_DESIGN_MODUS` directory for this application is overridden for just one target: `CY8CKIT-062S2-43012`.



3. Copy the *design.modus* file and other configuration files (that is, everything from inside the original BSP's `COMPONENT_BSP_DESIGN_MODUS` directory), and paste them into the new directory for the target.
4. In the application's *Makefile*, add the following lines. For example:

```
DISABLE_COMPONENTS += BSP_DESIGN_MODUS
COMPONENTS += CUSTOM_DESIGN_MODUS
```

**Note** The *Makefile* already contains blank `DISABLE_COMPONENTS` and `COMPONENTS` lines where you can add the appropriate names.

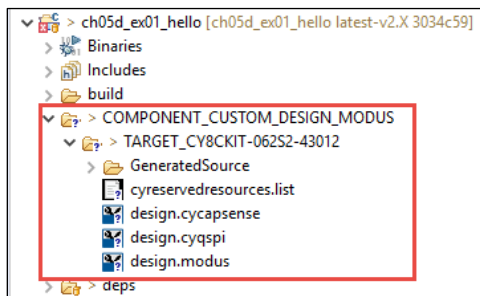
The first line disables the configuration files from the original BSP since they are now in different directory.

The second line is required to specify the new directory and include your custom configuration files so that the `init_cycfg_all` function is still called from the `cybsp_init` function. The `init_cycfg_all` function is used to initialize the hardware that was set up in the configuration files.

5. Customize the configuration files as required, such as using the Device Configurator to open the *design.modus* file and modify appropriate settings.

**Note** When you first create a custom configuration for an application, the Eclipse IDE Quick Panel entry to launch the Device Configurator may still open the *design.modus* file from the original BSP instead of the custom file. To fix this, click the **Refresh Quick Panel** link.

When you save the changes in the *design.modus* file, the source files are generated and placed under the *GeneratedSource* directory. The file structure appears as follows:



6. When finished customizing the configuration settings, you can build the application and program the device as usual.

#### 1.7.4 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*. If you change this variable, you will need to add the BSP for your new `TARGET` using the Library Manager.
2. In the application directory, run the `make bsp` target.

Specify the new board name by passing the value to the `TARGET_GEN` variable. Optionally you may specify a new device (`DEVICE_GEN`) and additional devices (`ADDITIONAL_DEVICES_GEN`) if they are different from the BSP that you started from. For example:

```
make bsp TARGET_GEN=MyBSP DEVICE_GEN=CY8C624ABZI-S2D44  
ADDITIONAL_DEVICES_GEN=CYW4343WKUBG
```

This command creates a new BSP in the root directory of the application project. It automatically copies the relevant startup and linker scripts into the newly created BSP, based on the device specified by the `DEVICE_GEN` option.

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 (for example, `capsense.lib` or `udb-sdio-whd.lib`) 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.

**Note** If you want your custom BSP to support only the LIB flow, then you should manually remove the `.mtbx` files from the BSP after creating it.

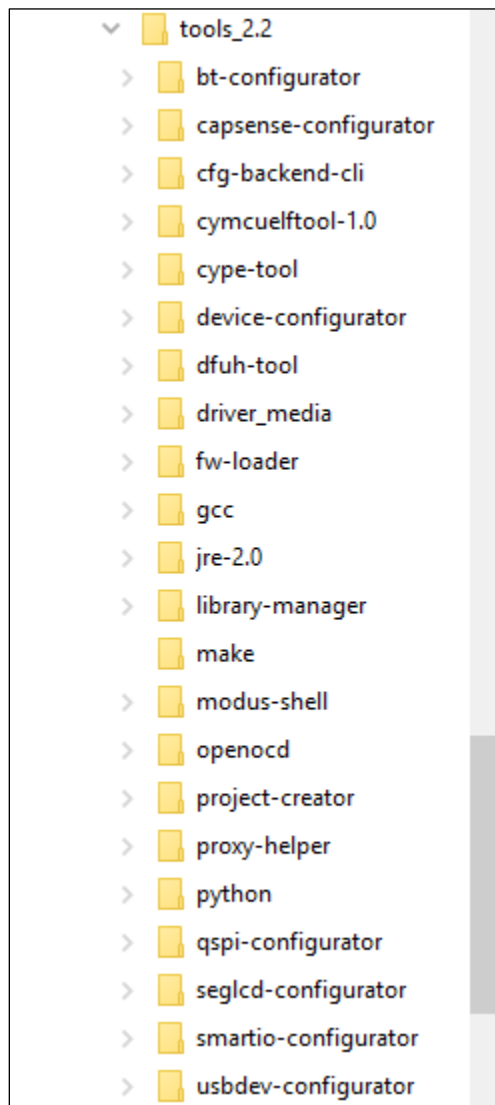
3. Update the application's *Makefile* `TARGET` variable to point to your new BSP. For example, `TARGET=MyBSP`. You must do this BEFORE running the Device Configurator so that it opens the correct file.
4. Open the Device Configurator to customize settings in the new device's *design.modus* file for pin names, clocks, power supplies, and peripherals as required. Address any issues that arise.
5. If using an IDE, regenerate the configuration settings to reflect the new BSP. Use the appropriate command(s) for the IDE(s) that are being used. For example: `make vscode`.
6. 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; see [Library Management](#).

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 [Manifest Files](#) 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.

## 1.8 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:

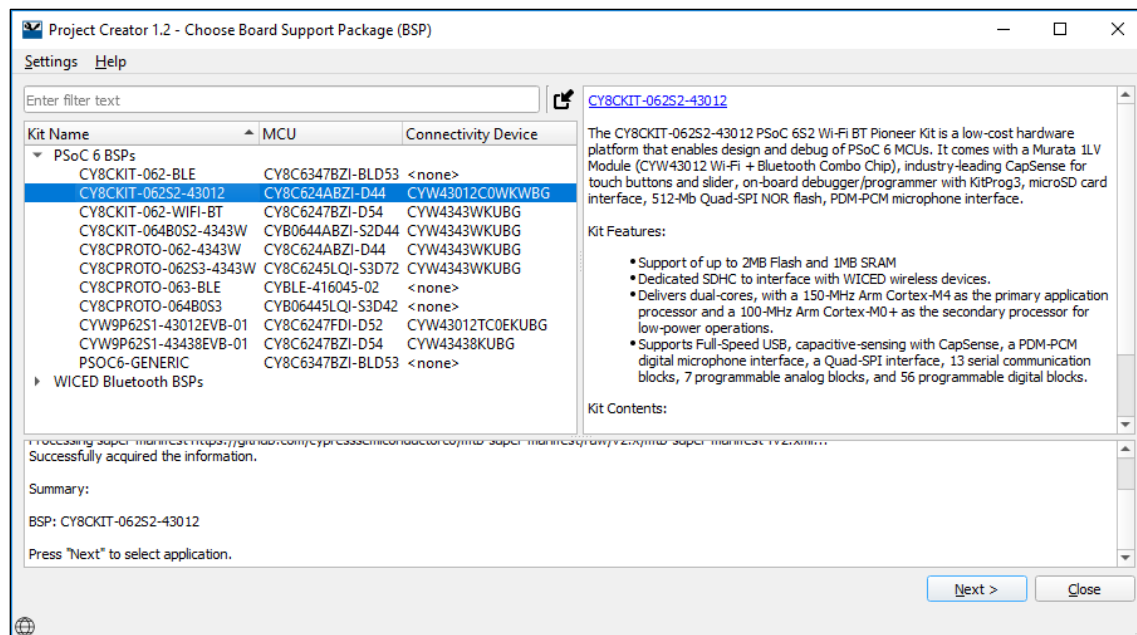
*C:/Users/<user-name>/ModusToolbox/tools\_2.2:*




## 1.8.1 Project Creator

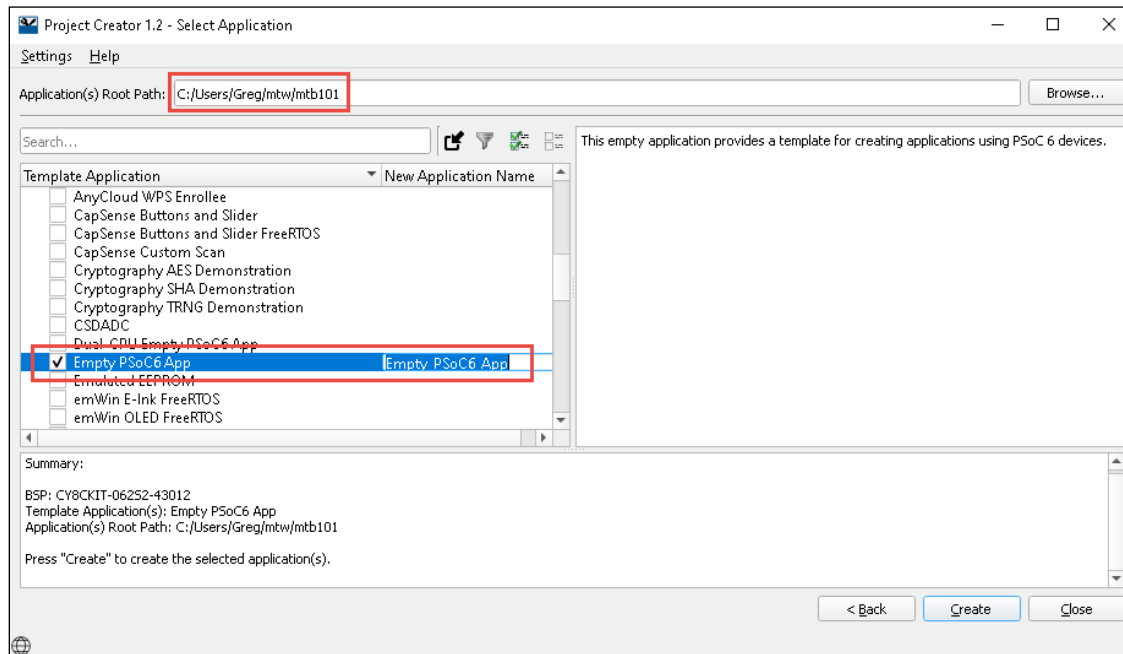
ModusToolbox software includes a project-creator tool that you can open from the IDE or as a stand-alone tool. The tool sets up new applications with the required target hardware support, wireless configuration code, middleware libraries, build, program and debug settings, and a "starter" application. 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 Cypress 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 [CY8CKIT-062S2-43012](#), along with the [CY8CKIT-028-TFT](#) display shield.

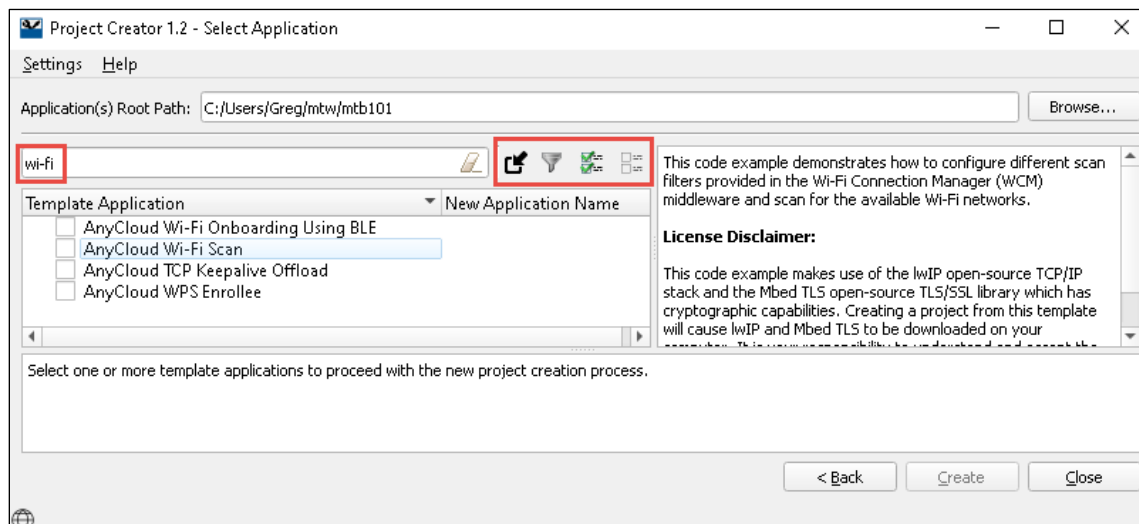


Note that there is an **Import** button  so you can specify your own custom BSP if you have one. We show you how to create your own BSP in the [Creating your Own BSP](#) section of this chapter.

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 wi-fi, you may see a list like this. The last 2 match because their descriptions have "wi-fi" in them. (The exact list will change over time as new code examples are created):

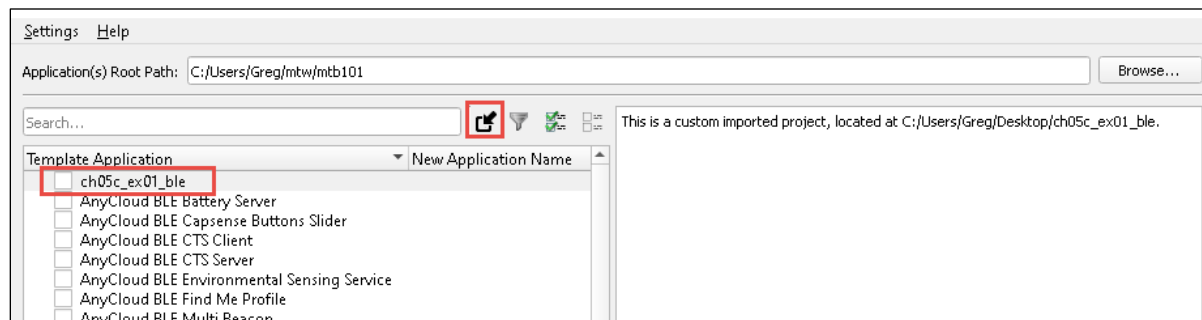


If desired, you can select all the applications in the search result by using the **Select All** button or you can unselect them with the **Unselect All** button.

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 not 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.

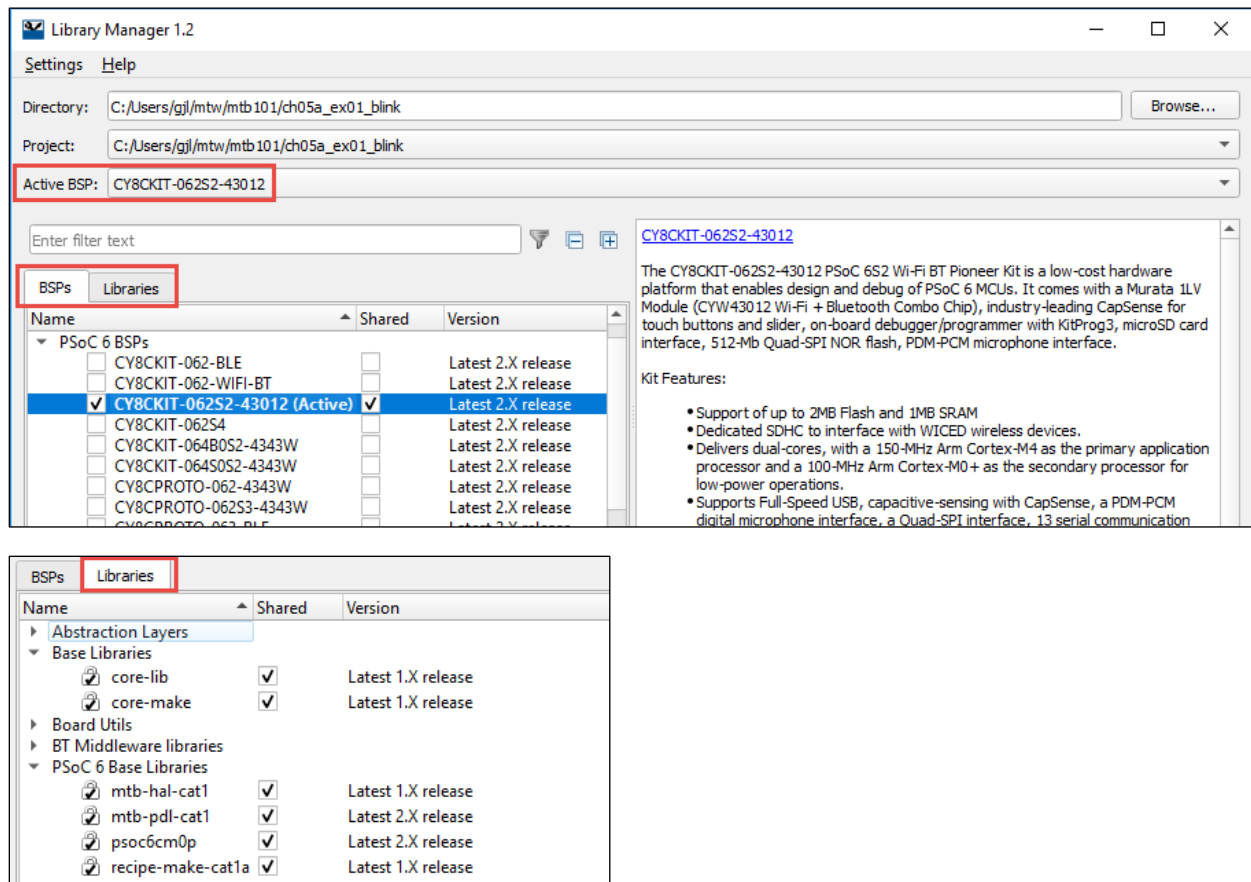
The tool runs the `git clone` operation and then the `make getlibs` operation. The project is saved in the directory you specified previously. When finished, click **Close**.

## 1.8.2 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* (Board Support Packages) 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, 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

### 1.8.3 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 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

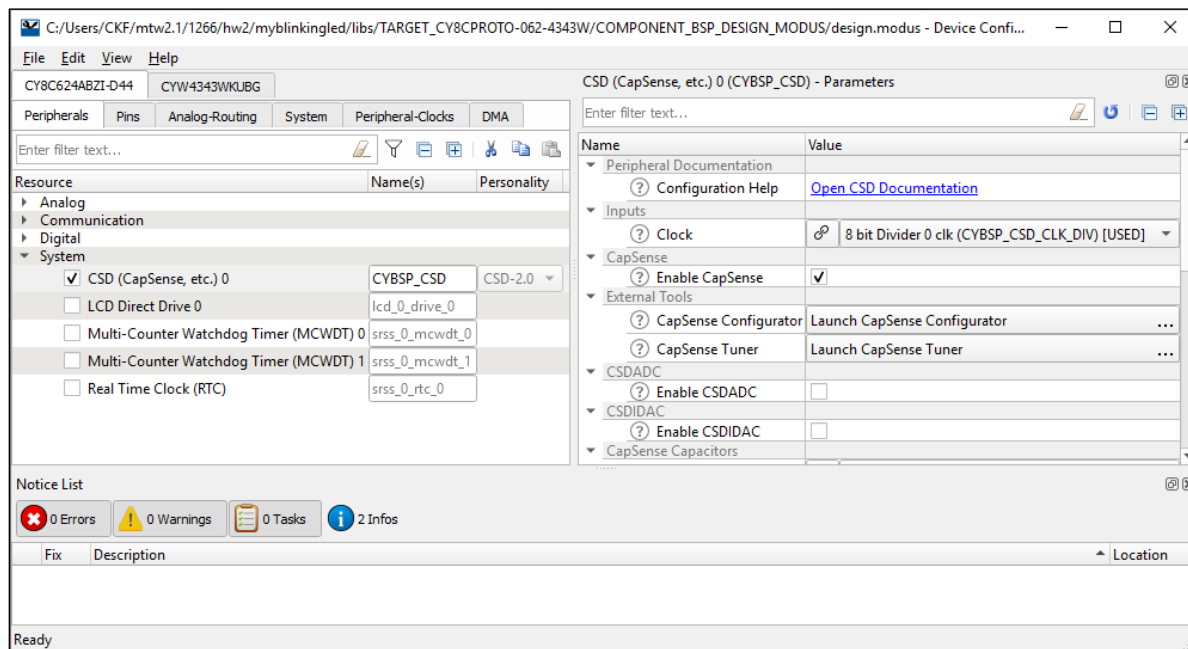
Configurators are divided into two types: Board Support Package (BSP) Configurators and Application Configurators.

### 1.8.4 BSP Configurators

BSP Configurators are closely related to the hardware resources on the board such as CapSense sensors, external Flash memory, etc. As described earlier, since these files are normally part of the BSP, if you use a configurator to modify them you will be modifying them for all applications that use the BSP (if it is shared). In addition, your changes will cause the BSP's repo to become dirty which means it cannot be updated to newer versions. Therefore, it is not recommended that you change any settings that are in the BSP. Instead, you can create a custom configuration for a single application (see [Modifying the BSP Configuration \(e.g. design.modus\) for a Single Application](#)) or you can create a custom BSP (see [Creating your Own BSP](#)).

### Device Configurator

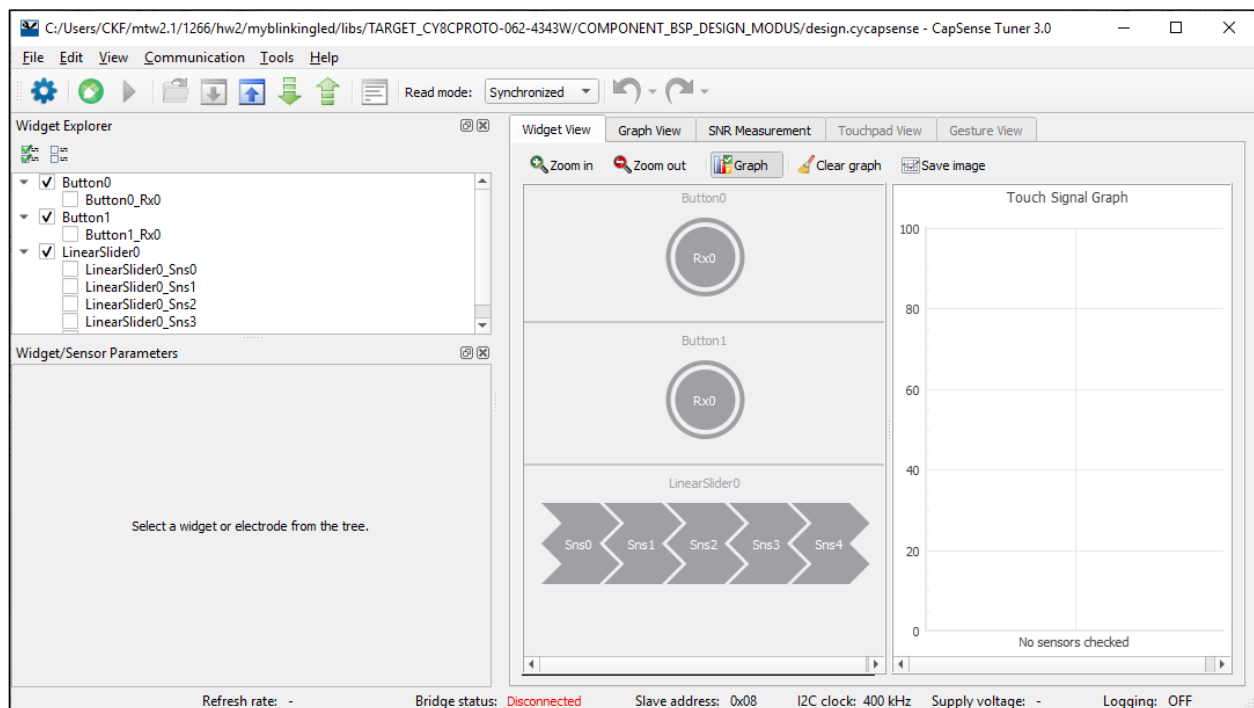
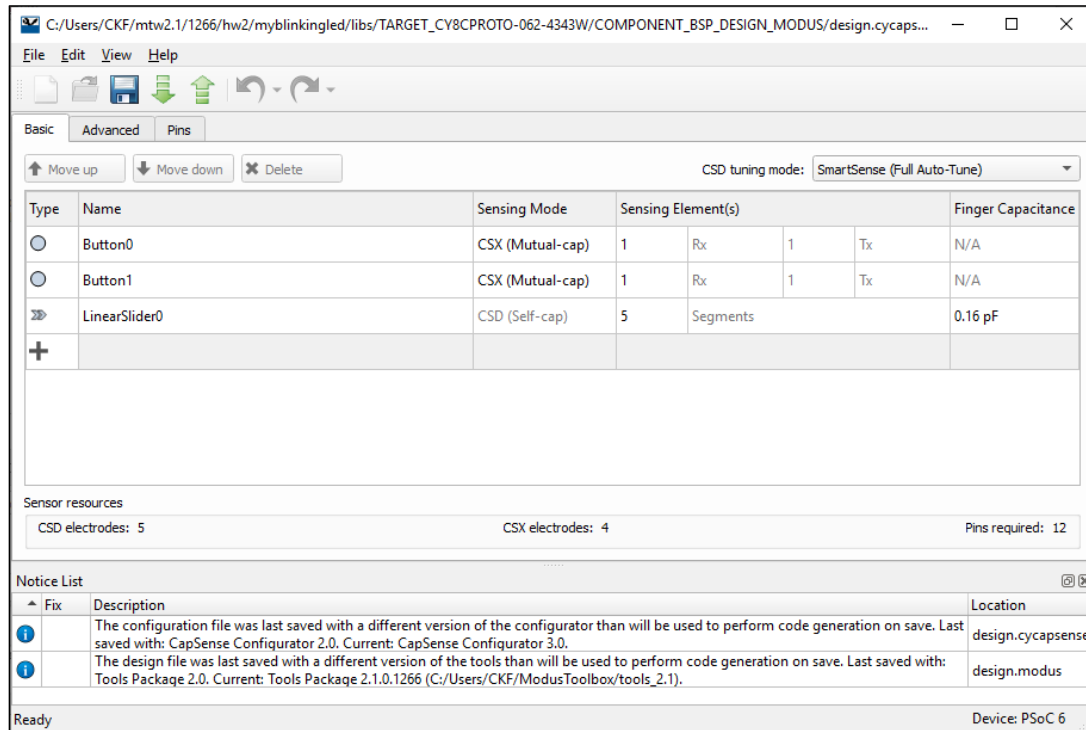
Set up the system (platform) functions such as pins, interrupts, clocks, and DMA, as well as the basic peripherals, including UART, Timer, etc.



**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.

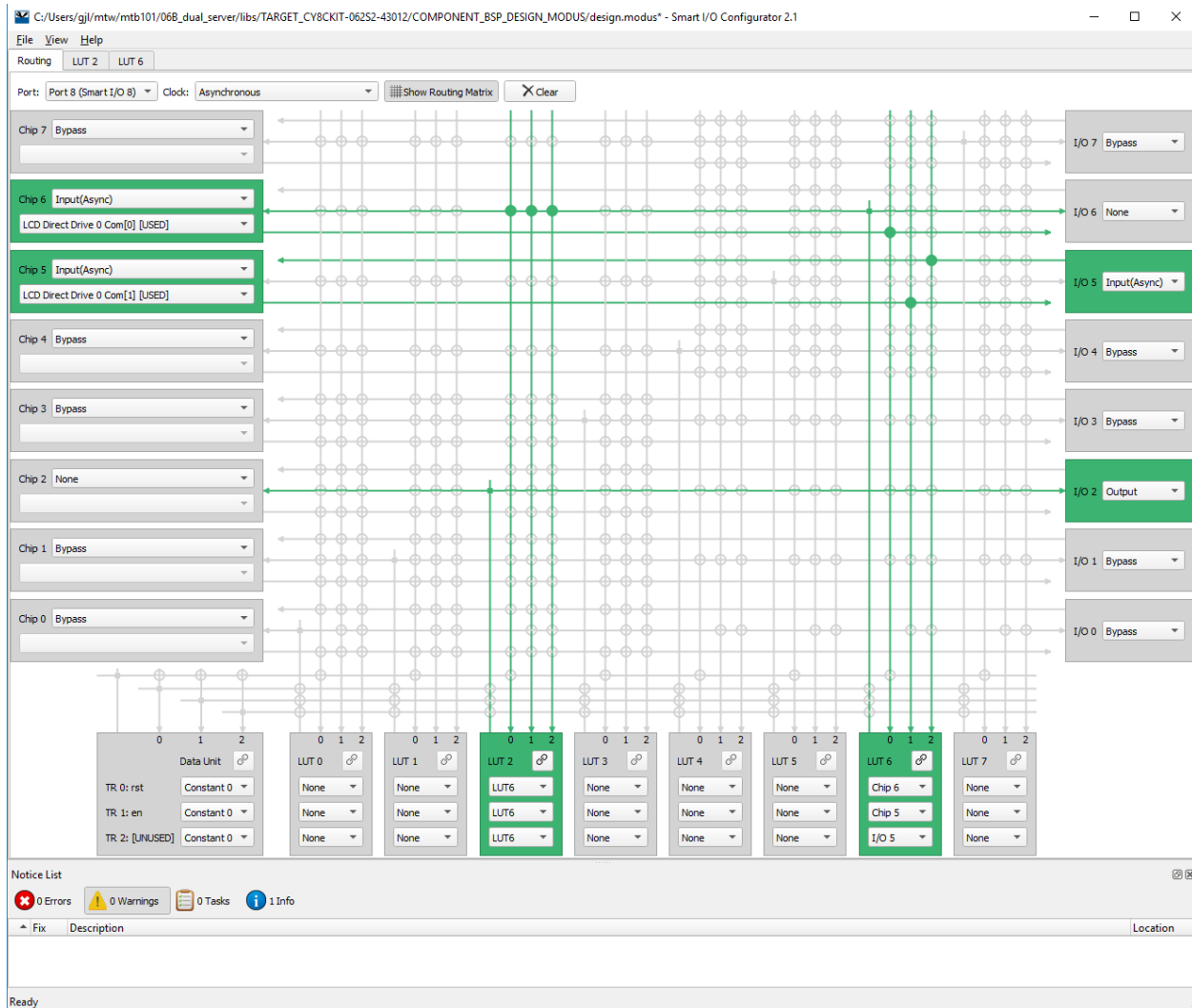
## CapSense Configurator/Tuner

Configure CapSense hardware and generate the required firmware. This includes tasks such as mapping pins to sensors and how the sensors are scanned.



## Smart I/O™ Configurator

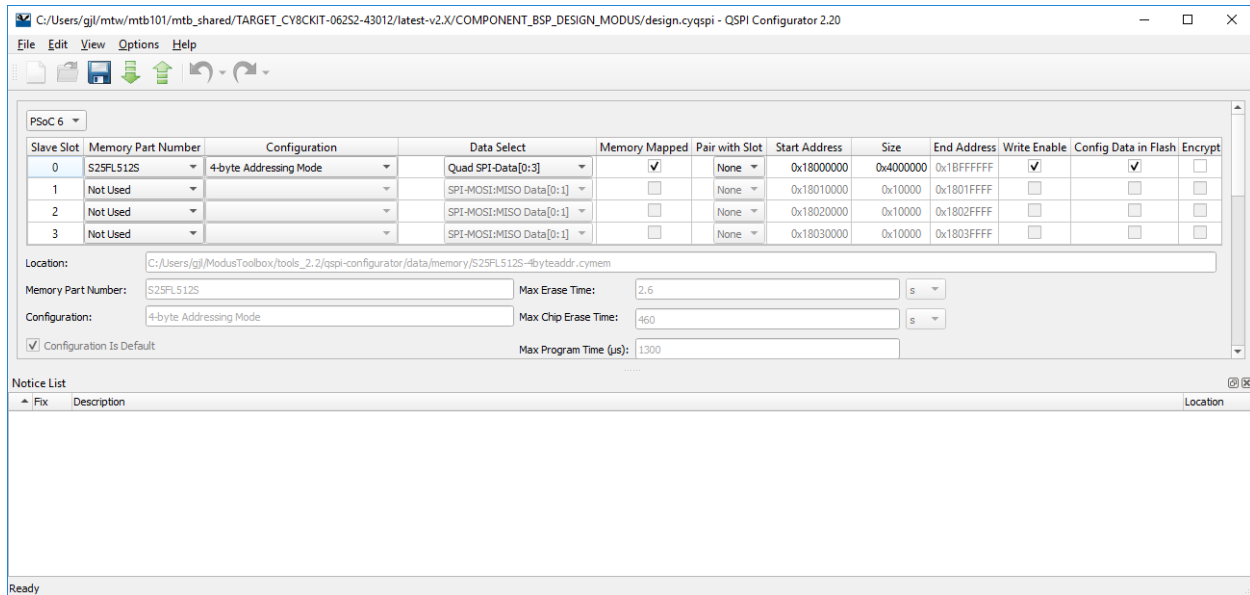
Configure the Smart I/O block, which adds programmable logic to an I/O port. There are usually only a few ports on a PSoC with Smart I/O capability, so it is important to consider that up front. On the PSoC from the CY8CKIT-062S2-43012, ports 8 and 9 have Smart I/O. Each port operates independently - that is, signals from port 8 cannot interact with signals from port 9.



The Smart I/O Configurator 2.1 interface displays a routing matrix for Port 8 (Smart I/O 8). The interface includes a menu bar (File, View, Help) and a toolbar with buttons for "Routing", "LUT 2", "LUT 6", "Show Routing Matrix", and "Clear". The main area shows a grid of routing connections between various components. On the left, a list of chips is shown: Chip 7 (Bypass), Chip 6 (Input(Async) with LCD Direct Drive 0 Com[0] [UNUSED]), Chip 5 (Input(Async) with LCD Direct Drive 0 Com[1] [UNUSED]), Chip 4 (Bypass), Chip 3 (Bypass), Chip 2 (None), Chip 1 (Bypass), and Chip 0 (Bypass). On the right, a list of I/O pins is shown: I/O 7 (Bypass), I/O 6 (None), I/O 5 (Input(Async)), I/O 4 (Bypass), I/O 3 (Bypass), I/O 2 (Output), I/O 1 (Bypass), and I/O 0 (Bypass). At the bottom, a row of LUTs (Look-Up Tables) is displayed, each with three inputs (0, 1, 2) and a dropdown menu. LUT 2, LUT 6, and LUT 7 are highlighted in green. The LUT 2 dropdown is set to "LUT6", LUT 6 is set to "Chip 6", and LUT 7 is set to "Chip 5". The bottom status bar shows "Ready" and a "Notice List" with 0 Errors, 0 Warnings, 0 Tasks, and 1 Info.

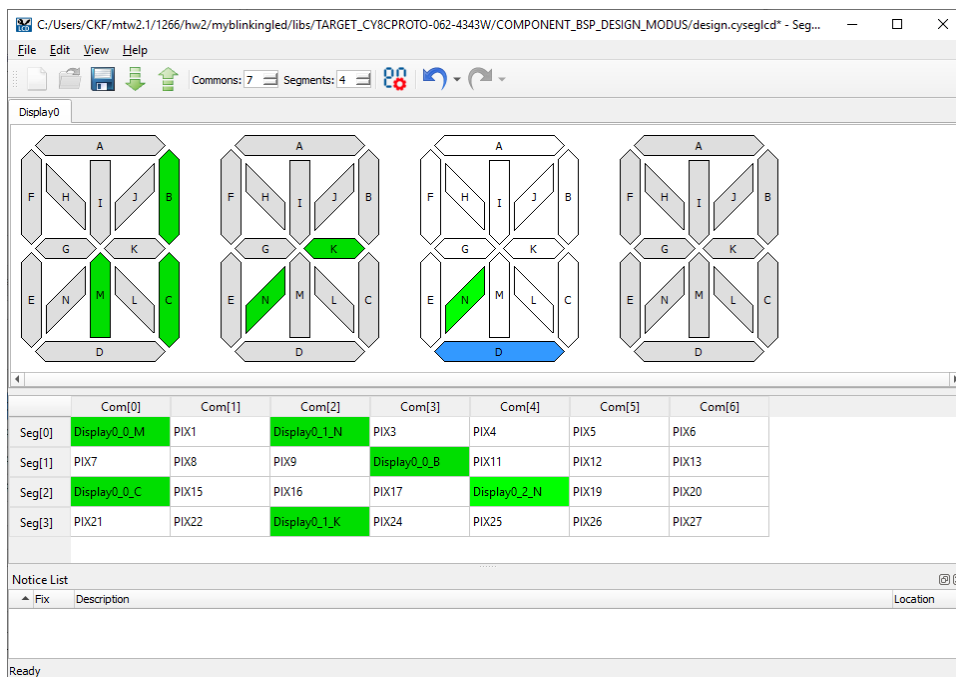
## QSPI Configurator

Configure external memory and generate the required firmware. This includes defining and configuring what external memories are being communicated with.



## SegLCD Configurator

Configure LCD displays. This configuration defines a matrix Seg LCD connection and allows you to setup the connections and easily write to the display.

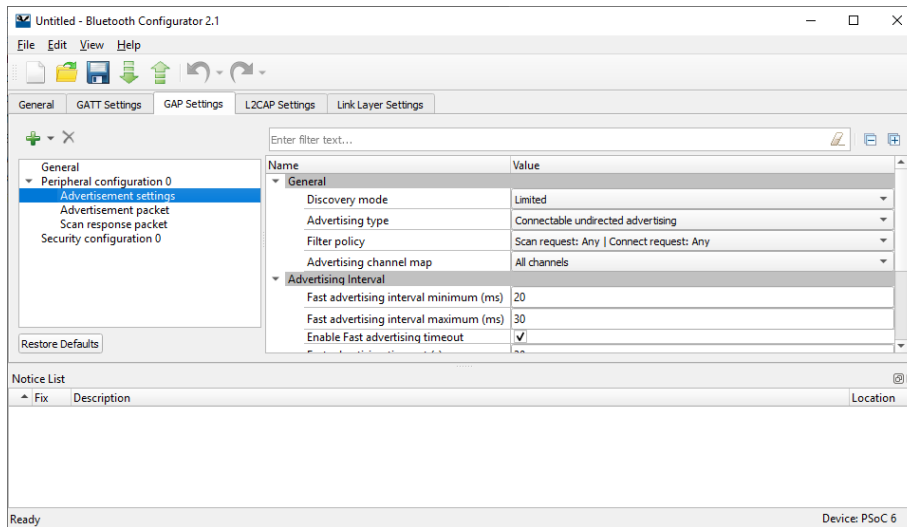


## 1.8.5 Application Configurators

Application Configurators are used to setup and configure project specific settings. The files for these are contained in the project hierarchy rather than inside the BSP.

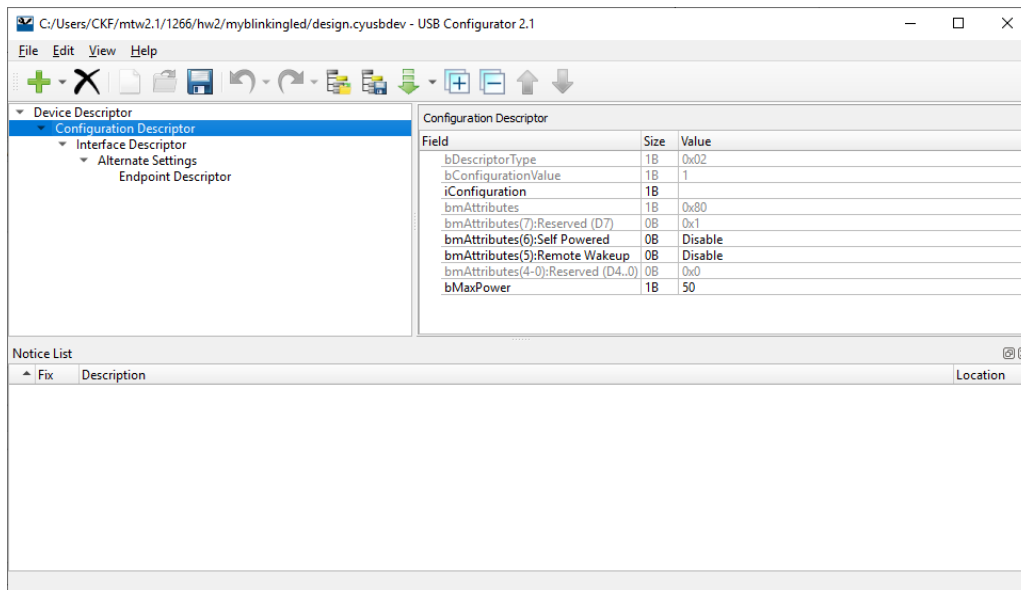
### Bluetooth Configurator

Configure Bluetooth settings. This includes options for specifying what services and profiles to use and what features to offer by creating SDP and/or GATT databases in generated code. This configurator supports both PSoC MCU and WICED Bluetooth applications.



### USB Configurator

Configure USB settings and generate the required firmware. This includes options for defining the 'Device' Descriptor and Settings.



### 1.8.6 Command Line

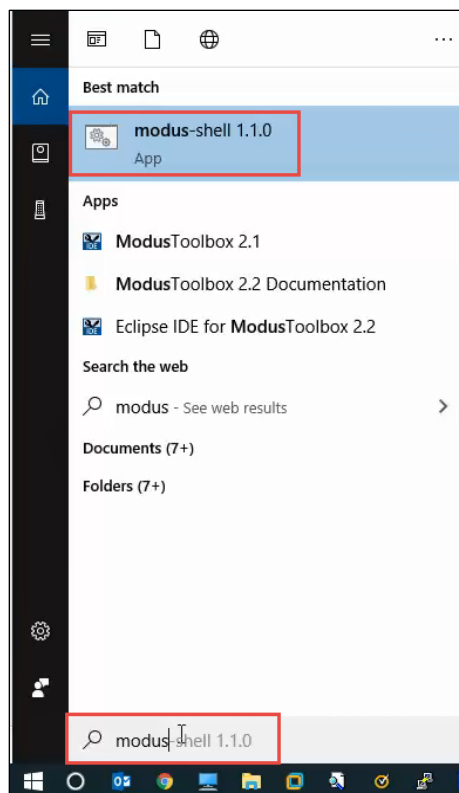
In addition to the Eclipse IDE for ModusToolbox, there is a command line interface that can be used for all operations including downloading applications, running configurators, building, programming, and even debugging.

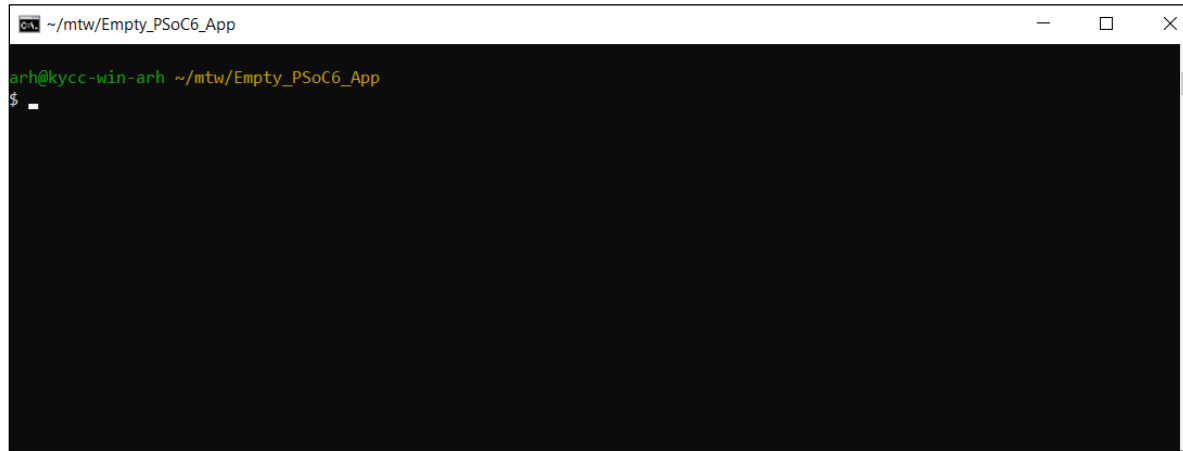
The make/build system works the same in all environments so once you have an application setup in an IDE, you can go back and forth between the command line and IDE at will. You can run all the tools using the command line.

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



```
modus-shell 1.1.0
gjl key_ch05a_ex01_blink $ make help CY_HELP=TARGET
Tools Directory: /cygdrive/c/Users/gjl/ModusToolbox/tools_2.2
CY8CKIT-062S2-43012.mk: ../mtb_shared/TARGET_CY8CKIT-062S2-43012/latest-v2.X/CY8CKIT-062S2-43012.mk

Topic-specific help for "TARGET"
  Brief: Specifies the target board/kit. (e.g. CY8CPROTO-062-4343W)

Current target in this application is, [ ../mtb_shared/TARGET_CY8CKIT-062S2-43012/latest-v2.X/CY8CKIT-062S2-43012.mk ].

Example Usage: make build TARGET=CY8CPROTO-062-4343W
gjl key_ch05a_ex01_blink $
```

## 1.9 Visual Studio 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 Visual Studio 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:

```
modus-shell 1.1.0
Build rules construction complete

=====
= Generating IDE files =
=====

=====
= Building application =
=====
Generating compilation database file...
-> ./build/compile_commands.json
Compilation database file generation complete

Generated Visual Studio Code files: c_cpp_properties.json launch.json openocd.tcl settings.json tasks.json
J-Link users, please see the comments at the top of the launch.json
file about setting the location of the gdb-server.

Instructions:
1. Review the modustoolbox.toolsPath property in .vscode/settings.json
2. Open VSCode
3. Install "C/C++" and "Cortex-Debug" extensions
4. File->Open Folder (Welcome page->Start->Open folder)
5. Select the app root directory and open
6. Builds: Terminal->Run Task
7. Debugging: "Bug icon" on the left-hand pane

landrygreg ((latest-v2.X) *) ch02_ex01_tft $
```

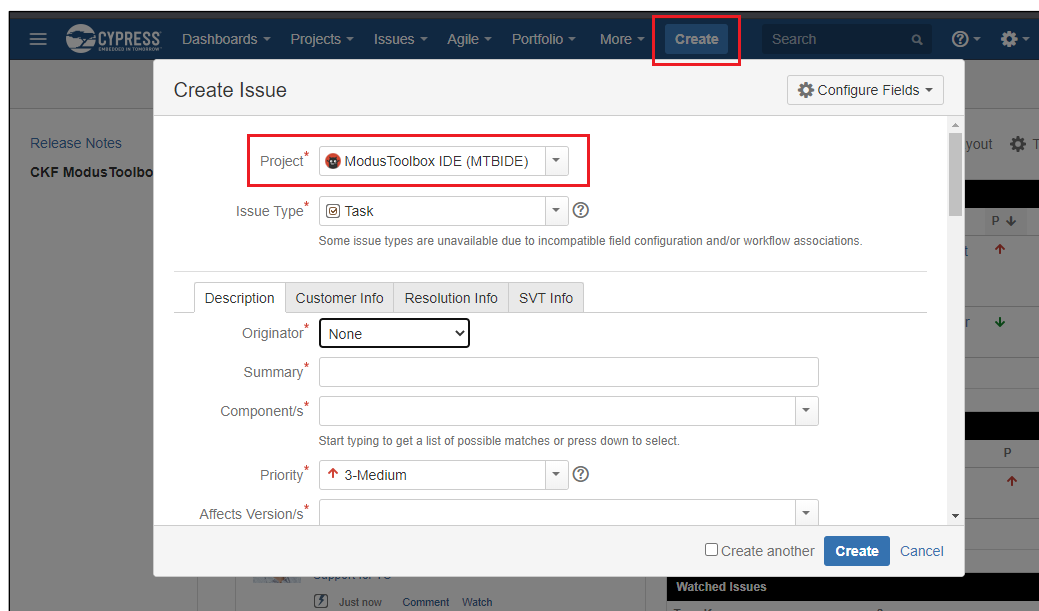
The message at the bottom of the command window tells you the next steps to take. However, for steps 2, 4, and 5, it is better to instead open the workspace file so that you will see the application as well as the shared library files from inside VSCode. From the command line just enter `<application-name>.code-workspace`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

## 1.10 Reporting Issues

If you are a Cypress employee and you find an issue in ModusToolbox (bug, missing or confusing documentation, enhancement request), please use a “JIRA” to report it. Non-Cypress employees should report issues via the forum. JIRA can be accessed at:

[jira.cypress.com](https://jira.cypress.com)

Click on **Create** to start submitting a JIRA. Use the project type of **MTBIDE** and fill in as many details as you can to report the issue. If you are reporting an issue with a kit, use “KITS:” as a prefix to the summary.



The screenshot shows the JIRA 'Create Issue' form. The 'Project' dropdown is set to 'ModusToolbox IDE (MTBIDE)' and is highlighted with a red box. The 'Issue Type' is set to 'Task'. The 'Priority' is set to '3-Medium'. The 'Create' button is highlighted with a blue box.

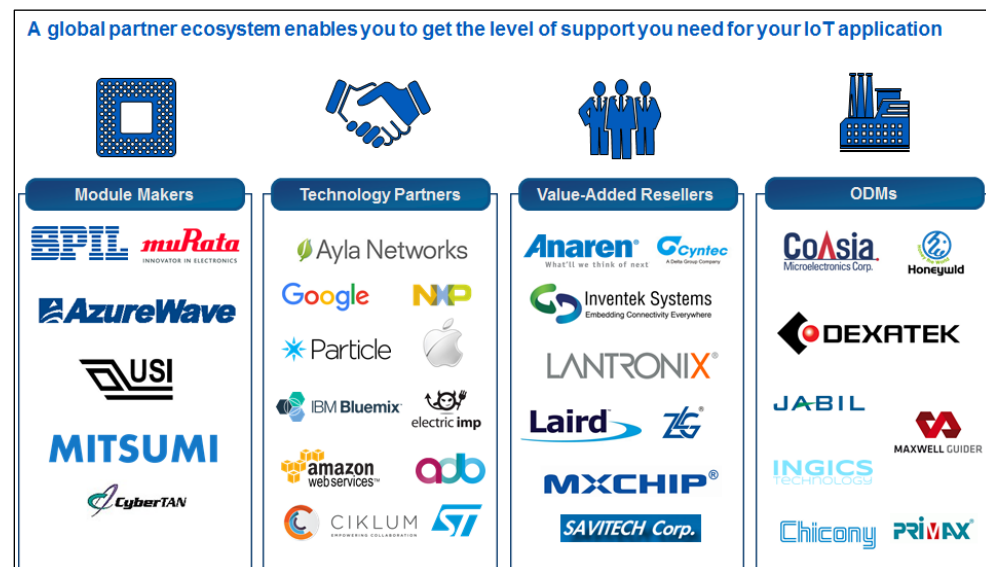
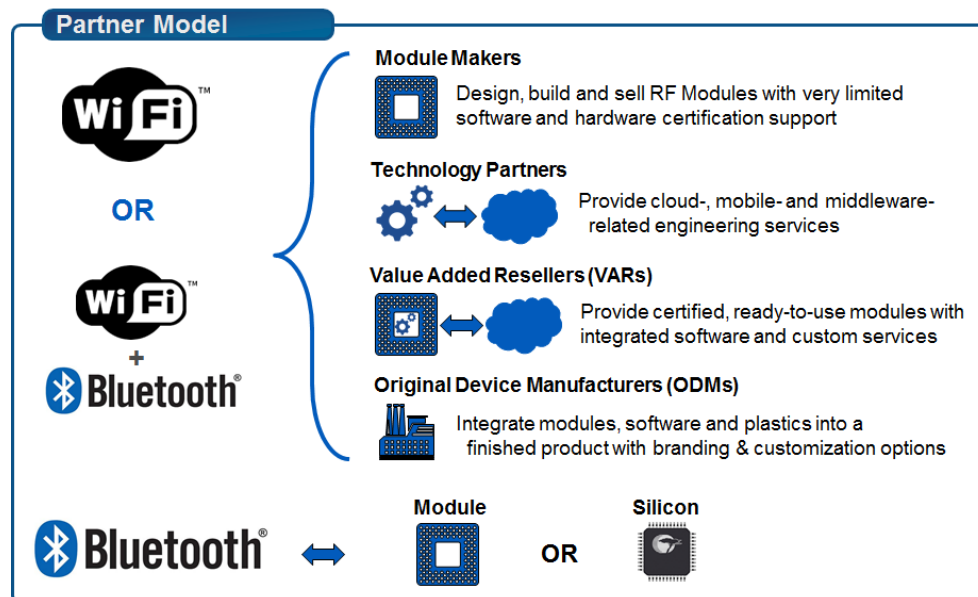
## 1.11 Tour of Wi-Fi

IEEE Standard	Mbits/s	Freq GHz	# Chan	Chan Width MHz	MIMO	Comment
802.11	2	2.4	14	22	-	
<a href="#">802.11b</a>	11	2.4	14	22	-	Same as 802.11 with new coding scheme
<a href="#">802.11a</a>	54	5	22	20	-	New coding scheme OFDM + 5 GHz
<a href="#">802.11g</a>	54	2.4	14	22	-	New coding scheme OFDM
<a href="#">802.11n</a>	600	2.4 5	14 22	20/40	4	MIMO=Multiple Antennas 4 streams of 150 Mbits/s
<a href="#">802.11ac</a>	3600	2.4 5	22 10 5 1	20 40 80 160	8	433 Mbits/s per stream Beam forming directional
802.11ax	10,000	2.4 5		20 40 80 160	4x4	

## 1.12 Tour of Chips

Device	Key Features	Notes
CYW43362	<ul style="list-style-type: none"> <li>Single band 2.4GHz</li> <li>1x1 11n</li> <li>Modules paired w/ STM32F205 and STM32F411</li> </ul>	Recommend new designs with 43364
CYW4390	<ul style="list-style-type: none"> <li>Single band 2.4GHz</li> <li>1x1 11n</li> </ul>	Recommend new designs with BCM43903/7 Black Box Only
CYW43340	<ul style="list-style-type: none"> <li>Dual band combo 2.4GHz and 5GHz, 1x1 11n</li> <li>BT4.0/BLE</li> </ul>	Currently only production dual band combo in single chip for WICED RTOS SDK
CYW43364	<ul style="list-style-type: none"> <li>Single band 2.4GHz, 1x1 11n</li> <li>Next Gen BCM43362</li> </ul>	Lower power and cost compared to BCM43362
CYW4343W	<ul style="list-style-type: none"> <li>Single band combo 2.4GHz</li> <li>BT4.1/BLE</li> </ul>	Lower cost and power compared to BCM43340
CYW43903	<ul style="list-style-type: none"> <li>Single band 2.4GHz , 1x1 11n</li> <li>SOC w/ ARM CR4 160Mhz</li> <li>1MB on chip RAM</li> <li>Secure OTP and HW crypto engine</li> </ul>	Lower cost solution for White Box High end Black Box features
CYW43907	<ul style="list-style-type: none"> <li>Dual band 2.4 and 5GHz, 1x1 11n</li> <li>SOC w/ dual ARM CR4 320Mhz</li> <li>2MB on chip RAM</li> <li>Secure OTP and HW crypto engine</li> </ul>	Ideal solution for White Box Multiple low power modes
CYW43012	<ul style="list-style-type: none"> <li>Dual band 2.4 and 5GHz, 1x1 11ac "friendly"</li> <li>Single 20 MHz channel compatible with 11ac networks</li> </ul>	Ideal for IOT since it gets power benefits of 11ac without complexity/overhead of 40, 80, and 160 MHz channels or MIMO

## 1.13 Tour of Partners



You can find an IoT Selector Guide including partner modules available in the Community at:

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

## 1.14 Tour of Development Kits

A list of some development kits that include Wi-Fi functionality are shown below. Some of these kits are supported by WICED Studio instead of ModusToolbox. Separate training exists for WICED Studio.

### 1.14.1 Cypress CY8CKIT-062S2-43012

<https://www.cypress.com/documentation/development-kitsboards/psoc-62s2-wi-fi-bt-pioneer-kit-cy8ckit-062s2-43012>

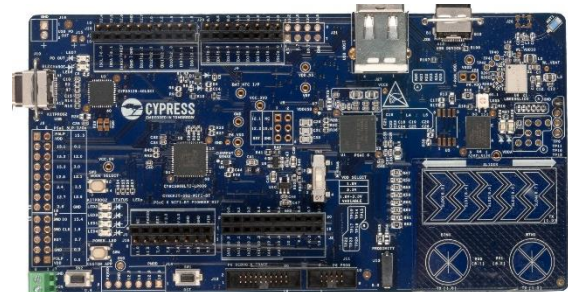
- Wi-Fi + BT Combo kit (CYW43012)
- PSoC 6 with 150 MHz M4 and 100 MHz M0+
- 2 MB Onboard Flash, 1 MB SRAM
- 4 Mbit 108 MHz F-RAM
- USB host and device support
- CapSense Buttons, and Slider
- 2 User Mechanical Buttons, 2 LEDs, 1 RGB LED
- Mbed OS Support



### 1.14.2 Cypress CY8CKIT-062-WiFi-BT

<https://www.cypress.com/documentation/development-kitsboards/psoc-6-wifi-bt-pioneer-kit-cy8ckit-062-wifi-bt>

- Wi-Fi + BLE combo kit (CYW4343W)
- PSoC 6 with 150 MHz M4 and 100 MHz M0+
- 1 MB Onboard Flash, 512 MB External Flash, 288 kB SRAM
- CCG3 USB Type-C Power Delivery
- USB host and device support
- CapSense Buttons, Slider, and Proximity
- 1 User Mechanical Button, 2 LEDs, 1 RGB LED
- Includes a CY8CKIT-028-TFT Display Shield



### 1.14.3 Cypress CYW943907AEVAL1F

<https://www.cypress.com/documentation/development-kitsboards/cyw943907aeval1f-evaluation-kit>

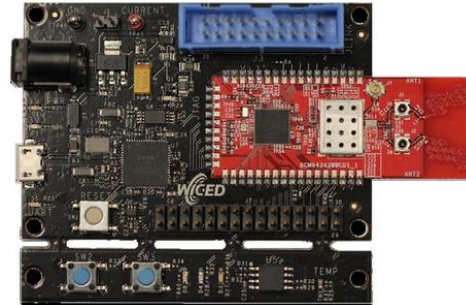
- Dual band 2.4 and 5 GHz Wi-Fi, 1x1 11n
- Ethernet
- SOC w/ Arm CR4 320 Mhz
- 2 MB on chip RAM
- Secure OTP and HW crypto engine
- USB JTAG Programmer/Debugger



#### 1.14.4 Cypress CYW94343WWCD1\_EVB Evaluation and Development Kit

<http://www.cypress.com/documentation/development-kitsboards/bcm94343wwcd1evb-evaluation-and-development-kit>

- Wi-Fi + BLE combo kit (CYW4343W)
- 512 kB Flash, 128 kB SRAM, 8 Mb SPI Flash
- 2 User Buttons, 2 User LEDs
- Thermistor
- USB JTAG Programmer/Debugger



#### 1.14.5 Future Nebula IoT Development Kit

<https://www.futureelectronics.com/p/development-tools--development-tool-hardware/neb1dx-02-future-electronics-dev-tools-5094171>

- Murata 1DX module with Wi-Fi + BLE combo kit (CYW4343W)
- 2 MB Flash, 256 kB SRAM, 16 Mb Serial Flash
- 2 User Buttons, 2 User LEDs
- Arduino, PMOD, and mikroBus Interfaces



#### 1.14.6 Arrow Quadro IoT Wi-Fi Kit

[https://cms.edn.com/ContentEETimes/Documents/EDN/WP\\_Arrow\\_Quadro\\_IOT\\_WiFi\\_Kit.pdf](https://cms.edn.com/ContentEETimes/Documents/EDN/WP_Arrow_Quadro_IOT_WiFi_Kit.pdf)

- Dual band 2.4 and 5 GHz Wi-Fi
- Ethernet
- 2 MB on chip RAM
- Designed for mounting on a custom PCB

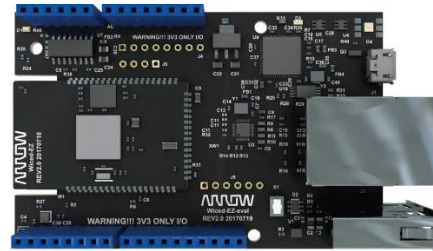




### 1.14.7 Arrow Quicksilver IoT Kit

<https://www.arrow.com/en/products/quicksilver/arrow-development-tools> <https://cms.edn/>

- CWY943907 Wi-Fi
- Ethernet
- Cypress 64 Mb QSPI NOR Flash
- Temperature and Humidity Sensors
- Arduino Compatible Headers



### 1.14.8 Avnet BCM4343W IoT Starter Kit

<http://cloudconnectkits.org/product/avnet-bcm4343w-iot-starter-kit>

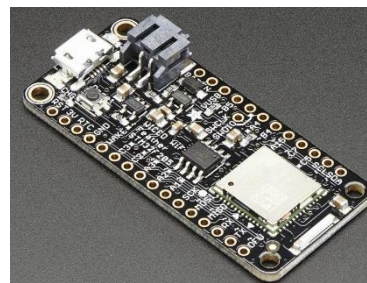
- Wi-Fi + BLE combo kit (CYW4343W)
- 512 kB Flash, 128 kB SRAM, 8 Mb SPI Flash
- 1 User Button, 2 User LEDs
- Ambient Light Sensor
- Arduino Compatible Headers
- USB JTAG Programmer/Debugger



### 1.14.9 Adafruit Feather

<https://www.adafruit.com/products/3056>

- Wi-Fi kit (CYW43362)
- 128 kB Flash, 16 kB SRAM, 16 Mb SPI Flash
- Programmable using Arduino IDE
- USB Bootloader



### 1.14.10 Electric Imp

<https://www.electricimp.com/platform/>

- Wi-Fi kit (IMP003- CYW43362, IMP005 – CYW43907)
- Programmable using imp IDE

<No Image>

#### 1.14.11 Particle Photon

[https://store.particle.io/products/photon?\\_pos=1&\\_sid=60412a1f8&\\_ss=r](https://store.particle.io/products/photon?_pos=1&_sid=60412a1f8&_ss=r)  
<https://www.particle.io/products/hardware/photon-wifi-dev-kit>

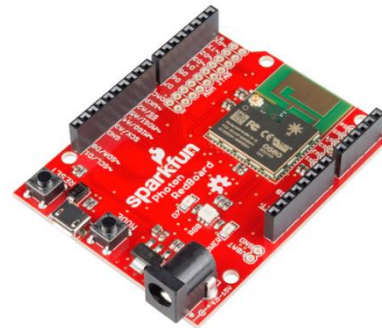
- Wi-Fi kit (CYW43362)
- 1 MB Flash, 128 kB SRAM



#### 1.14.12 SparkFun with Particle Photon Module

<https://www.sparkfun.com/products/13321>

- Wi-Fi kit (CYW43362)
- 1 MB Flash, 128 kB SRAM
- Arduino Compatible Headers

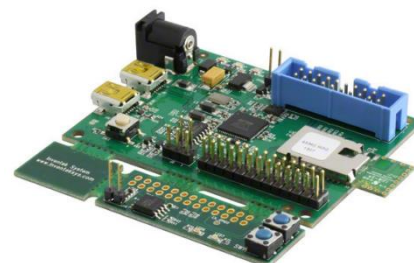


#### 1.14.13 Inventek

<http://www.inventeksys.com/>

##### ISM43362-M3G-EVB

- Wi-Fi Kit (CYW43362)
- 2 User Buttons, 2 User LEDs
- Thermistor
- USB JTAG Programmer/Debugger



##### ISM43340-M4G-EVB

- Wi-Fi & Bluetooth Combo Kit (CYW43340)
- 2 User Buttons, 2 User LEDs
- Thermistor
- USB JTAG Programmer/Debugger





#### ISMART Arduino Shield

<No Image>

- Wi-Fi, Bluetooth, NFC Combo (CYW43362)
- Arduino stackable shield

#### ISM43340-L77-EVB

- Wi-Fi & Bluetooth Combo Kit (CYW43340)
- Wi-Fi over SDIO
- Bluetooth over UART
- Micro-SD Connector



## 1.15 Exercise(s)

### 1.15.1 Exercise 1: Create a forum account

☐  
☐

1. Go to <https://community.cypress.com/welcome>
2. Click **Sign 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 **Software** menu on the left side of the page.
4. Click on the **ModusToolbox** link or the **ModusToolbox > ModusToolbox AnyCloud SDK** link.
5. Browse the existing forum articles or search for a particular topic that interests you.

### 1.15.2 Exercise 2: Install ModusToolbox

☐  
☐

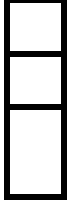
1. Go to <https://www.cypress.com/products/modustoolbox-software-environment>
2. Click **Download ModusToolbox** for the operating system you are running.

**Note** If you do not have an account, you will need to create one first.

☐

3. For additional help refer to the [ModusToolbox Installation Guide](#).

### 1.15.3 Exercise 3: Open the documentation



1. Open the help menu in the Eclipse IDE and try visiting the different documentation links.
2. Navigate through the Quick Panel Documentation links to see what is available in each.
3. Navigate through the *mtb\_shared* (or *libs* directory if using the LIB flow) directory in the Project Explorer and open some of the various library documentation.