

Chapter 2: ModusToolbox™ tools

After completing this chapter, you will understand each of the ModusToolbox™ tools, what they do, and how they fit together to give you a complete development flow.

Table of contents

2.1	Tools and the make system	3
2.1.1	Make.....	3
2.2	Application creation	4
2.2.1	Project Creator (PC) tool GUI	4
2.2.2	Project Creator tool CLI.....	7
2.2.3	Using Git (manual creation)	8
2.3	Importing/exporting applications for IDEs	10
2.3.1	Import projects into the Eclipse IDE	10
2.3.2	Export for Microsoft Visual Studio Code (VS Code)	11
2.3.3	Export for IAR Embedded Workbench	17
2.3.4	Export for Keil µVision.....	19
2.4	Eclipse IDE for ModusToolbox™ tour	20
2.4.1	Quick Panel.....	21
2.4.2	Help menu	22
2.4.3	Integrated debugger	23
2.4.4	Documentation	23
2.4.5	Eclipse IDE tips & tricks	23
2.5	Application directory organization	24
2.5.1	Makefile build settings	25
2.6	Using the command line (CLI)	27
2.7	Library management	30
2.7.1	Library classification	30
2.7.2	Library Manager tool.....	32
2.7.3	Manual library management	35
2.7.4	Re-Downloading Libraries	37
2.8	Configurators	38
2.8.1	BSP configurators	38
2.8.2	Library configurators	43
2.9	Board Support Packages (BSPs)	44
2.9.1	BSP directory structure.....	44
2.9.2	COMPONENT_BSP_DESIGN_MODUS Contents	45
2.9.3	BSP documentation	46
2.9.4	Modifying the BSP Configuration (e.g. design.modus) for an Application	47
2.9.5	Creating your own BSP	48
2.9.6	Updating the MCU device in a BSP	49
2.9.7	Updating the connectivity device in a BSP	50
2.10	Manifests.....	50
2.11	Network considerations.....	51

2.11.1	Network proxy settings	51
2.11.2	Offloading manifest files	52
2.11.3	Offline content	52
2.12	Compiler optimization	53
2.13	Multi-CPU applications	53
2.14	Exercises	55
Exercise 1: Create, build, program and debug a Hello World application in Eclipse		55
Exercise 2: Create, build, program and debug a Hello World application in VSCode		59
Exercise 3: Build and program using the Command Line Interface.....		62
Exercise 4: Use the Library Manager		63
Exercise 5: Create a Custom BSP and Use the Device Configurator		64

Document conventions

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

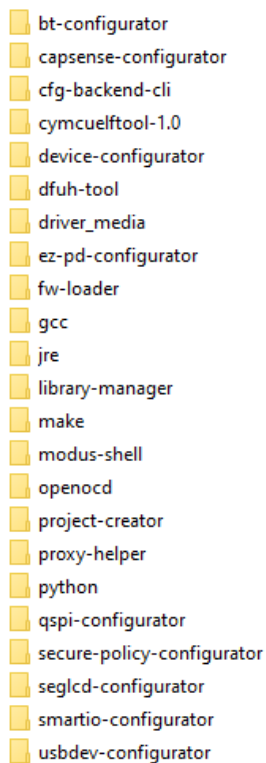
2.1 Tools and the make system

As you learned in the last chapter, the ModusToolbox™ ecosystem consists of a set of tools as well as software assets. In this chapter, we will go through the most commonly used tools and examine each type of asset.

The list of tools may change over time as new tools are added. However, tools that are installed by the ModusToolbox™ tools installer are located by default in the following location:

`~/ModusToolbox/tools_<version>`

The directory will look something like this:



```
bt-configurator
capsense-configurator
cfg-backend-cli
cymcuelftool-1.0
device-configurator
dfuh-tool
driver_media
ez-pd-configurator
fw-loader
gcc
jre
library-manager
make
modus-shell
openocd
project-creator
proxy-helper
python
qspi-configurator
secure-policy-configurator
seglcd-configurator
smartio-configurator
usbdev-configurator
```

Each tool has its own subdirectory containing the tool itself and documentation. In many cases, tools will have both a GUI version and a command line version.

You can launch many tools from inside an IDE – more on that later. In Windows, there are **Start** menu options for each tool or you can enter the tool name in the Windows Search Box. On all systems, you can run tools directly from the tool's installation directory.

2.1.1 Make

One of the directories under tools is *make*. Many operations in the ModusToolbox™ ecosystem are built upon the GNU Make system (<https://www.gnu.org/software/make/>), so it is a fundamental part of the ecosystem. In fact, even when you use an IDE, many operations just call the underlying *make* operation so that you get the same behavior no matter how you run the operation. For example, to build from the command line, you use the command `make build`. When you build the same application from inside an IDE, it just calls the same `make build` command for you. This is important because it means you can move back and forth between IDE operations and command line operations knowing that the results won't change.

2.2 Application creation

The first thing you need to do to start using ModusToolbox™ tools is to create an application. The simplest and easiest way to do this is to use the project creator tool. It requires 2 pieces of information: (1) a starting BSP that specifies the hardware configuration; and (2) a starting template application. Both of these items can be one of the standard ones provided by Infineon, or you can use your own.

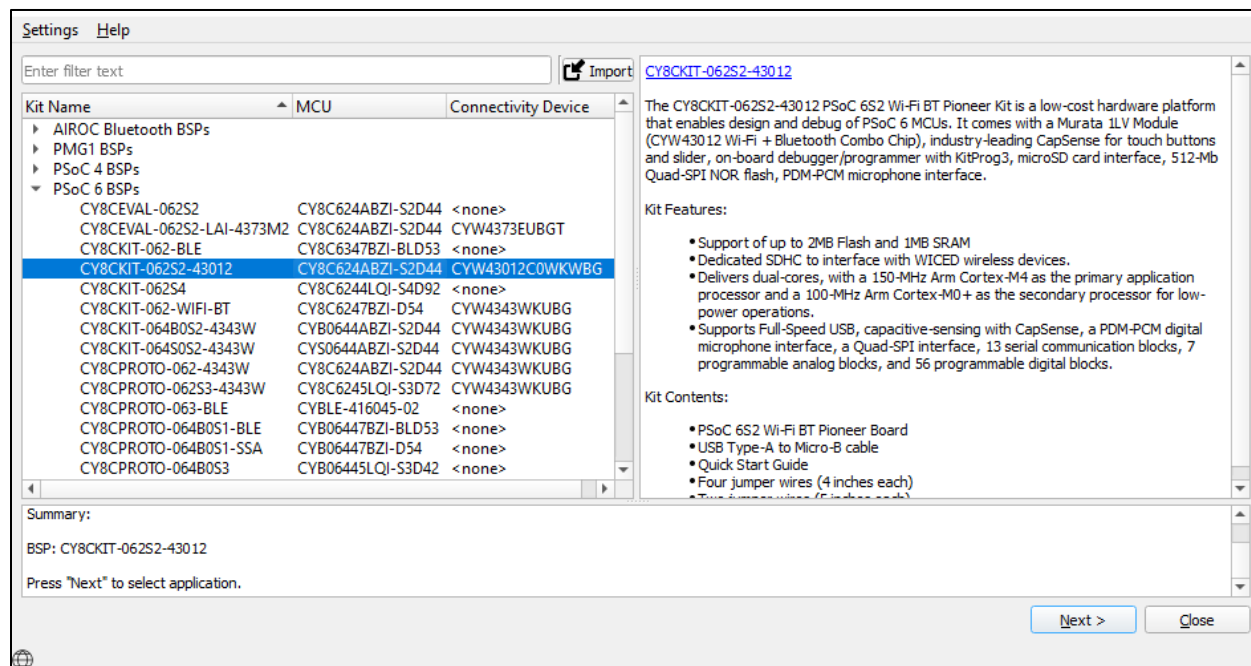
2.2.1 Project Creator (PC) tool GUI

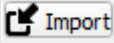
You can launch Project Creator from inside the Eclipse IDE for ModusToolbox™ by using the **New Application** link in the quick panel or the menu item **File > New > ModusToolbox Application**. If you are not using Eclipse, you can run Project Creator independently. In Windows, it is in the **Start** menu or you can enter **project-creator** in the Search Box. On all systems, you can run it from the ModusToolbox™ tools installation directory. The default location is `~/ModusToolbox/tools_<version>/project-creator/project-creator.exe`.

The first thing the project-creator tool does is read the configuration information from our GitHub site so that it knows all the BSPs and code examples that we support. That information is stored in *manifest* files which we will discuss in the [Manifests](#) section.

Note: You can also create projects in offline mode. See the [Offline Content](#) section for details.

Once the manifests have been read, the first window asks you to select the target BSP. They are categorized by device family as you can see here.

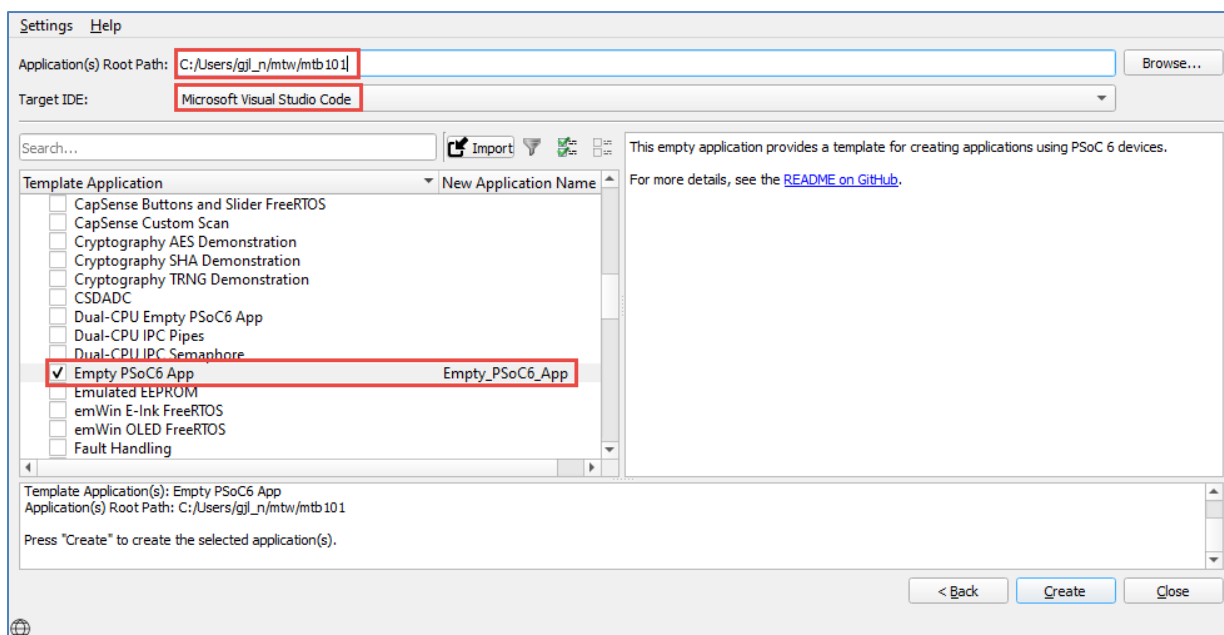


Note that there is an  **Import** button, so you can specify your own custom BSP if you have one. We will show you how to create your own BSP in the [Creating your own BSP](#) section of this chapter.

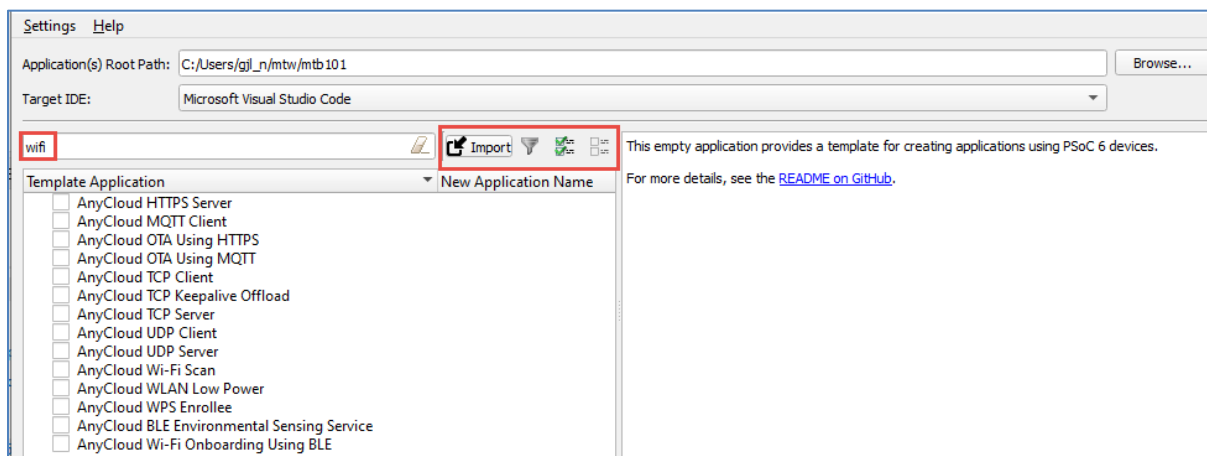
Select a kit and click **Next >**.

Now you will be presented with all the code examples supported by the BSP you chose. Pick an application to start from and give it a name. 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.

The Target IDE selection lets you decide if you want files to be created for use with an IDE. Currently it supports Eclipse IDE for ModusToolbox™, Microsoft Visual Studio Code (VS Code), IAR Embedded Workbench, and ARM Keil™ μVision™. If you run the Project Creator from the Eclipse IDE, this field will be fixed to Eclipse IDE for ModusToolbox™. If you don't select an IDE at this step, you can generate the required files later using the command line. You'll get to see that a bit later.



You can use the "Search..." box to enter a string to match from the application names and their descriptions. For example, if you enter *wifi*, you may see a list like this. (The exact list will change over time as new code examples are created):

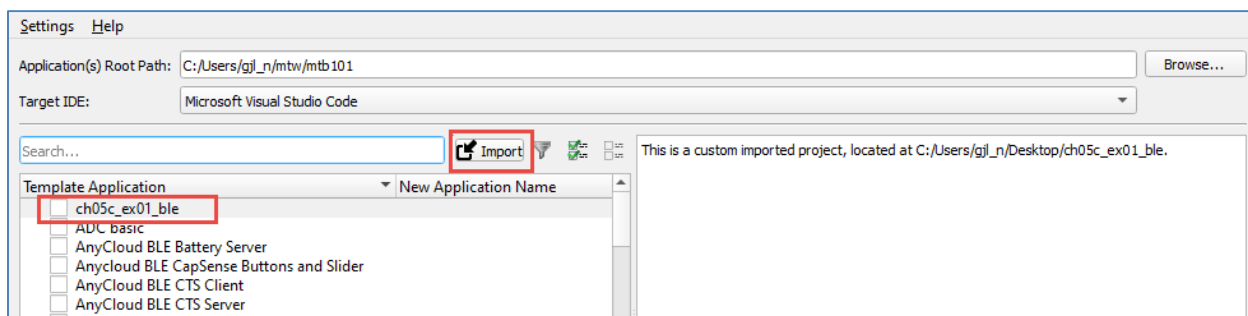


If desired, you can select all the applications in the search result by using the **Select All** button or you can unselect them with the **Unselect All** button.

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 folder, 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 (or another IDE) 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.

Once you have selected the application 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 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**.

2.2.1.1 Creating bundled applications

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*.

Dual-CPU applications use the bundled application concept. For those applications, there is a top-level application name with sub-application directories for each CPU. This will be discussed in more detail in [Multi-CPU](#).

One thing to be careful of when creating bundled applications is the location of any shared libraries relative to the application. This will be discussed in more detail in the [Error! Reference source not found.](#) section.

2.2.2 Project Creator tool CLI

The Project Creator tool described above also has a command line version that you can run from a shell (e.g. modus-shell). It is in the same directory as the Project Creator GUI (*~/ModusToolbox/tools_<version>/project-creator*) but the executable is called *project-creator-cli.exe*. You can run it with no arguments to see the different options available. For example, you can run it to see a listing of all the available BSPs:

```
./project-creator-cli --list-boards
```

You can also see all the available applications for a given BSP:

```
./project-creator-cli --list-apps <BSP_Name>
```

For example, to use the Project Creator in CLI mode to create an empty PSoC™ 6 MCU application for the CY8CKIT-062S2-43012 kit and place it in the user's *mtw/mtb101* directory with a name of *my_app*, the command is:

```
./project-creator-cli  
  --board-id CY8CKIT-062S2-43012  
  --app-id mtb-example-psoc6-empty-app  
  --target-dir "~/mtw/mtb101"  
  --user-app-name "my_app"
```

Note that double-quotes are required around the target directory name if you are using a tilde (~) to specify the user's home directory.

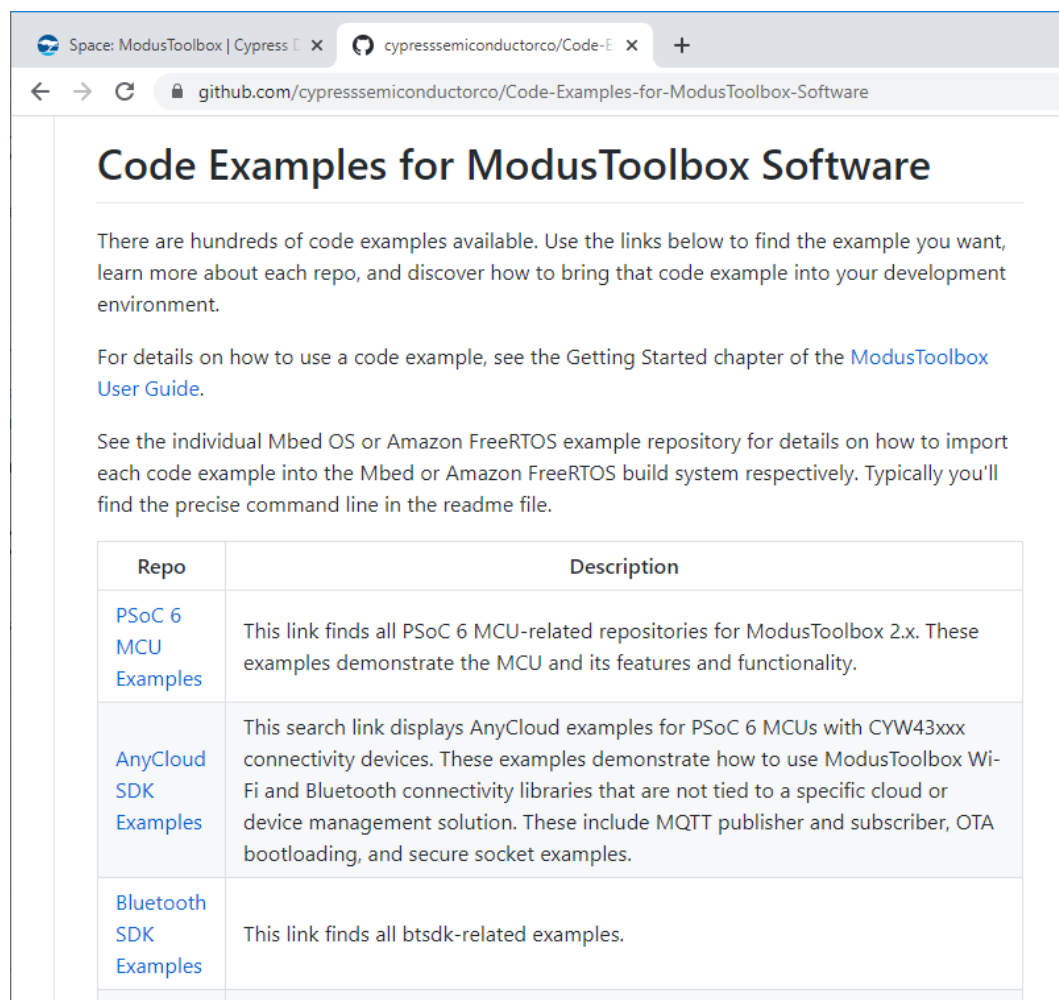
2.2.3 Using Git (manual creation)

Instead of using the Project Creator tool, you can use git commands directly to clone a project.

Note: It is more complicated to set things up manually, so it is typically better to use the Project Creator tool GUI or from the command line, but the manual git method exists if you need it.

You can get to the Cypress GitHub repo using the "Search Online for Code Examples" link in the Eclipse IDE Quick Panel. That will take you the website:

<https://github.com/cypresssemiconductorco/Code-Examples-for-ModusToolbox-Software>



Space: ModusToolbox | Cypress | x cypresssemiconductorco/Code-E x +

github.com/cypresssemiconductorco/Code-Examples-for-ModusToolbox-Software

Code Examples for ModusToolbox Software

There are hundreds of code examples available. Use the links below to find the example you want, learn more about each repo, and discover how to bring that code example into your development environment.

For details on how to use a code example, see the Getting Started chapter of the [ModusToolbox User Guide](#).

See the individual Mbed OS or Amazon FreeRTOS example repository for details on how to import each code example into the Mbed or Amazon FreeRTOS build system respectively. Typically you'll find the precise command line in the readme file.

Repo	Description
PSoC 6 MCU Examples	This link finds all PSoC 6 MCU-related repositories for ModusToolbox 2.x. These examples demonstrate the MCU and its features and functionality.
AnyCloud SDK Examples	This search link displays AnyCloud examples for PSoC 6 MCUs with CYW43xxx connectivity devices. These examples demonstrate how to use ModusToolbox Wi-Fi and Bluetooth connectivity libraries that are not tied to a specific cloud or device management solution. These include MQTT publisher and subscriber, OTA bootloading, and secure socket examples.
Bluetooth SDK Examples	This link finds all btsdk-related examples.

You can use the GitHub website tools to clone projects, or if you know the URL for the repo containing the code example that you want to clone, you can use the `git clone` command from the command line to clone the project. For example:

```
git clone https://github.com/cypresssemiconductorco/mtb-example-psoc6-empty-app
```

Once you have cloned the project from GitHub or the command line, change to the project's directory:

```
cd mtb-example-psoc6-empty-app
```


Run `make` to see the list of available commands:

```
Greg@DESKTOP-31QU767 ~/mtw/mtb101/ch05a_ex01_blink
$ make
Tools Directory: C:/Users/Greg/ModusToolbox/tools_2.2
TARGET.mk: ../mtb_shared/TARGET_CY8CKIT-062S2-43012/latest-v2.X/CY8CKIT-062S2-43012.mk

=====
Getting started
=====
1. Run "make getlibs" to import the dependent libraries.
2. Run "make build" to build the application. Use -j for parallel builds.
3. Run "make program" (or "make qprogram") to program a target board.

For more information, run "make help" to print the help documentation.
Note: The help documentation is available after running Step 1.

Greg@DESKTOP-31QU767 ~/mtw/mtb101/ch05a_ex01_blink
$
```

Then you can run:

```
make getlibs
make build
make program
```

Note: When using the Project Creator tool, all libraries are locked to a specific release instead of pointing to the latest version so they will never change versions without the user requesting it. If you use the manual `git clone` method, direct dependencies that specify `latest-vN.X` in the `.mtb` file will not be locked to a specific release. Therefore, it is recommended that you use the Library Manager to lock dependencies to a specific version after using the `git clone` method to create an application. Alternately, you can update the version in the `.mtb` file in the `deps` directory and re-run `make getlibs` to lock versions. See [Library management](#) for more details.

The Project Creator tool will also convert a BSP from shared to local if specified in the manifest file. That conversion is not done when using `git clone` directly. The BSP location can be changed using the Library Manager or by manually editing the `.mtb` file and re-running `make getlibs`.

For example, the `.mtb` file for a BSP may look like the following when using Project Creator vs. `git clone` for the same application:

```
https://github.com/cypresssemiconductorco/TARGET_CY8CKIT-062S2-43012#release-v2.1.0##LOCAL##/TARGET_CY8CKIT-062S2-43012

https://github.com/cypresssemiconductorco/TARGET_CY8CPROTO-062-4343W#latest-v2.X##ASSET_REPO##/TARGET_CY8CPROTO-062-4343W/latest-v2.X
```

Note: Each code example includes a default BSP. If you use the manual `git clone` method and the BSP you want to use is different than the default, you must: (1) change the `TARGET` in the Makefile; (2) add a `.mtb` file for the desired BSP; and (3) run `make getlibs` again. Alternately, you can use the Library Manager to add the desired BSP and set it as active. See [Library management](#) for more details.

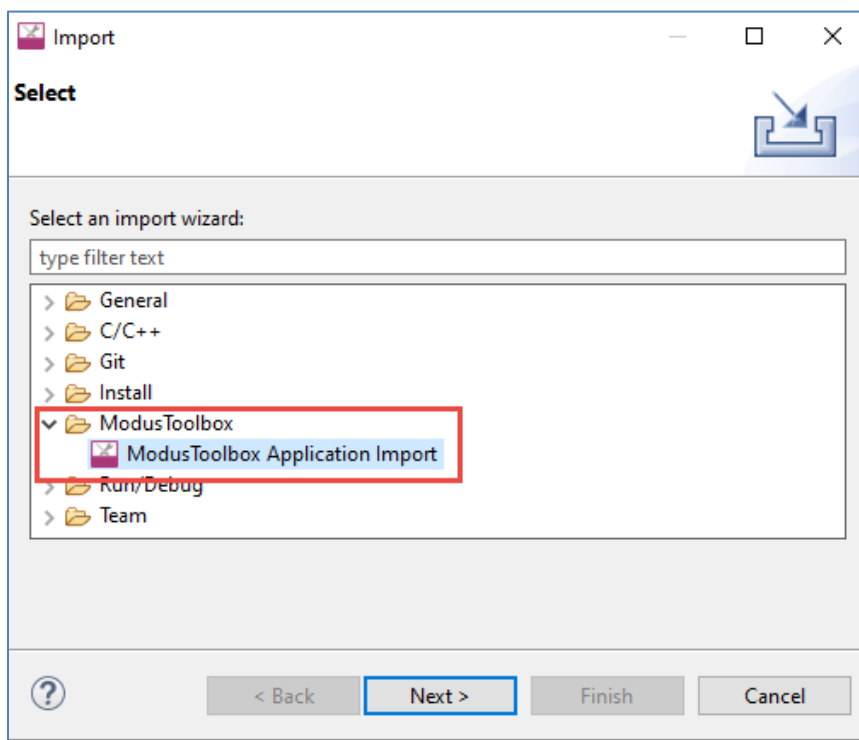
2.3 Importing/exporting applications for IDEs

Once you have an application created with Project Creator, you can use it from the command line, or you can convert it to use in the IDE of your choosing. You can even switch back and forth between the command line and IDE.

If you launched Project Creator from Eclipse, the new application is imported into Eclipse automatically. Likewise, if you ran Project Creator in stand-alone mode and selected a target IDE, the necessary files are created for you. If not, you can still create the necessary files by running the `make <IDE>` command. There is also an import option in the Eclipse IDE for ModusToolbox™ that runs `make eclipse` plus a few other required actions, so it isn't necessary for you to run `make eclipse` manually if that's the IDE you are using.

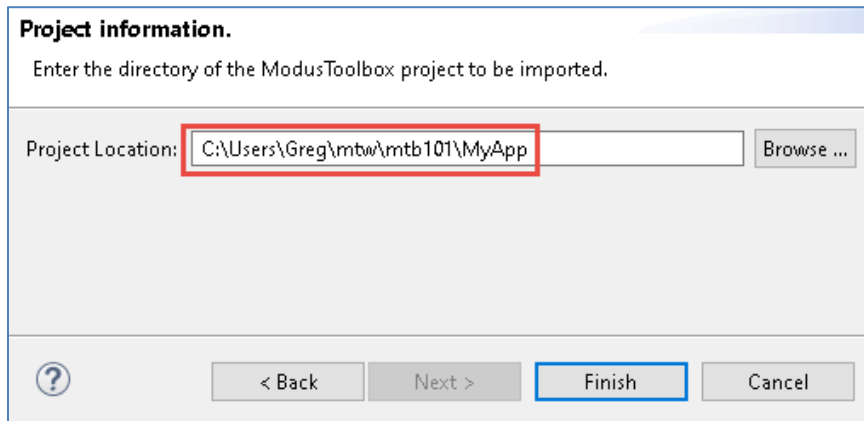
2.3.1 Import projects into the Eclipse IDE

If you have a project that was created from the command line or stand-alone Project Creator tool, you can use the Eclipse Import feature: **File > Import... > ModusToolbox™ > ModusToolbox™ Application Import**.



Note: *This imports the application in-place; it does NOT copy it into your workspace. If you want the resulting application to be in your workspace, make sure you put it in the workspace folder before importing it.*

After clicking **Next >**, browse to the location of the application's directory, select it, and click **Finish**.



This import mechanism runs `make eclipse` plus additional commands to help the application run smoothly in the Eclipse IDE for ModusToolbox™.

2.3.1.1 Launch configs

A few important notes regarding the launch configurations inside Eclipse:

There is a directory inside the application called `.mtbLaunchConfigs`. This is where the launch configurations are located for an application. There are cases where you may end up with old launch configurations in that directory which can confuse Eclipse. In the Quick Panel, you may see no launch configs, the wrong launch configs, or multiple sets of launch configs.

In case you see that behavior, it is safe to blow away the `.mtbLaunchConfigs` directory at any time and then just click on **Generate Launches for <project name>** in the Quick Panel.

Some of the cases where you may end up with stale or multiple sets of launch configs:

- If you rename an application directory before importing.
- If you rename an application inside of Eclipse (that is, **right-click > Rename**).

2.3.2 Export for Microsoft Visual Studio Code (VS Code)

VS Code 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. If you didn't select Microsoft Visual Studio Code in the Project Creator tool when you created the project, you will need to create the required files using the command line interface. To create the files, go to an existing project directory and type the following from the command line:

```
make vscode
```

```
~/mtbx/mtb101/ch05a_ex01_blink
-> Found 0 resource file(s)
Applying filters...
Auto-discovery complete

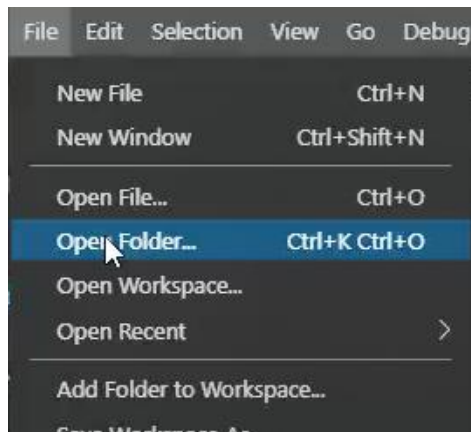
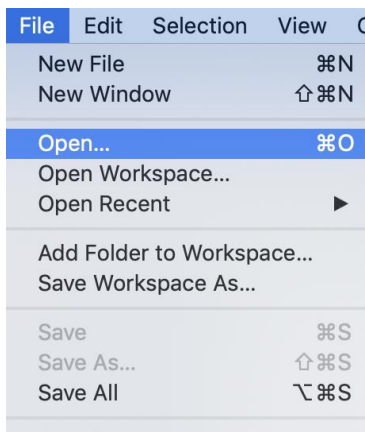
=====
- Generating IDE files -
=====

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 VS Code
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
```

The message at the bottom of the command window tells you the next steps to take. These steps are explained as follows:

1. This step is just to verify that the version of ModusToolbox™ tools is what you expect.
2. Start VS Code.
3. Install the C/C++ and Cortex-Debug extensions if you don't already have them.
4. Open your application in Visual Studio Code using **File > Open** (on macOS) or **File > Open Folder** (on Windows).

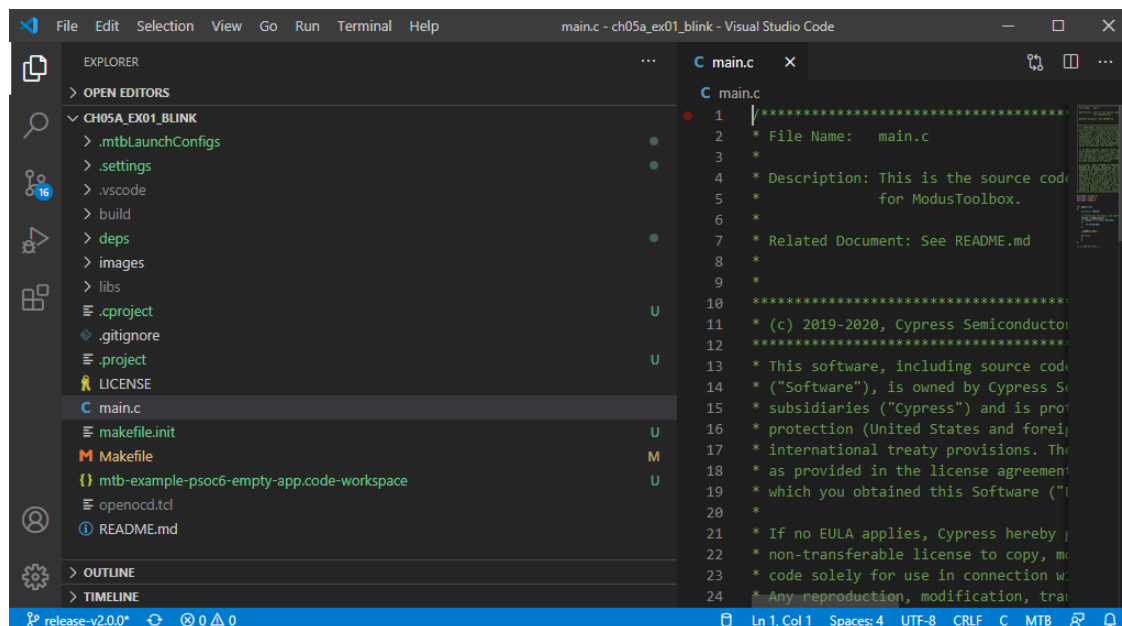


5. Then, navigate to the directory where your project resides and click **Open**.

Note: Alternately, if it is in your path you can just enter `code .` on the command line after running `make vscode` and it will open VS Code and load the application all in one step. (Depending on how you installed VS Code, it may or may not be in your path by default.)

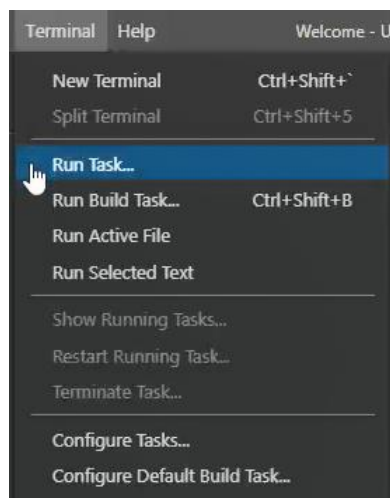
Note: In Windows, you can right-click on the application directory and select the menu option **Open with Code** to start VSCode and open the application.

Now your windows should look something like this. You can open the files by clicking on them in the Explorer window.

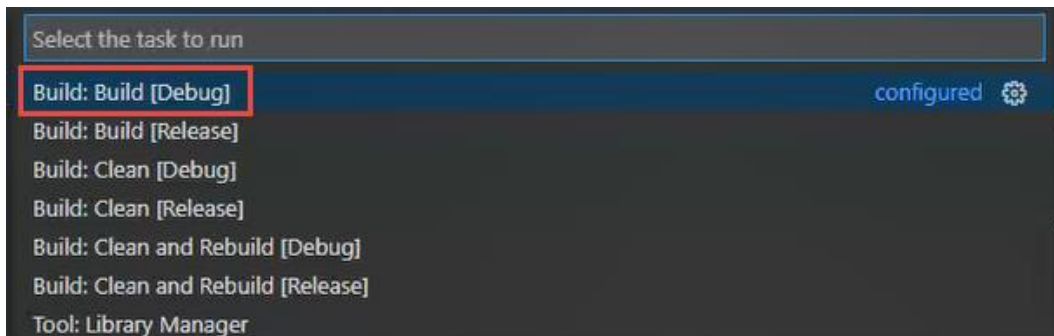


Note: *This method will work, but it does not show you the shared libraries (mtb_shared). If you want to see those libraries, either click the popup window in VSCode to open the workspace or follow the instructions below regarding workspaces.*

- In order to build your project, make sure your application is selected and do **Terminal > Run Task...**

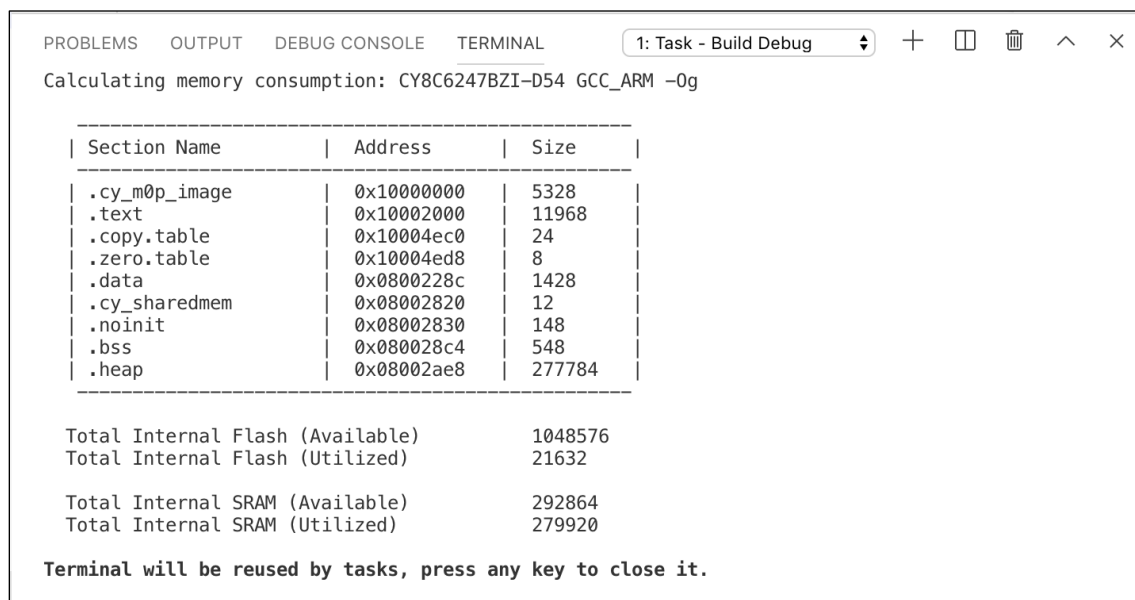


Then, select **Build Debug**. This will essentially run `make build` at the top level of your project.



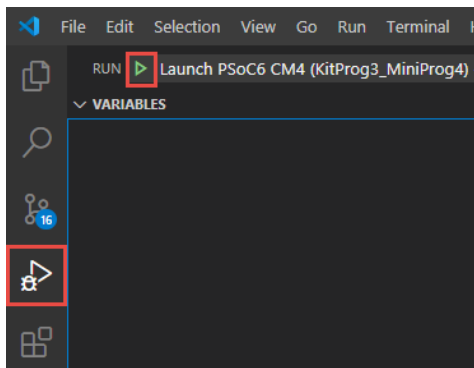
Note: You can also access the Library Manager from the Task list.

You should see the normal compiler output in the console at the bottom of VS Code:

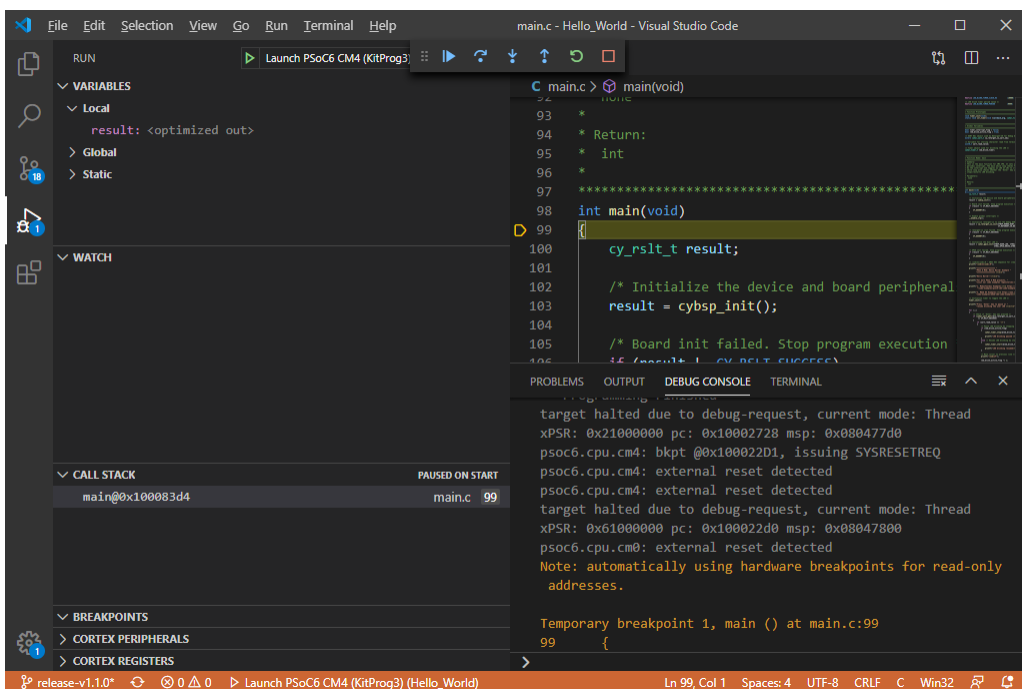


- To build and then program the kit or run the debugger, first make sure you have installed the "Cortex-Debug" VS Code extension from the marketplace. See the software installation exercise in the introduction chapter if you didn't do this when you installed VS Code.

To do programming or debug, click the little "bug" icon on the left side of the screen. Then click the drop-down next to the green **Play** button to select Program (to program), Launch (to program and start the debugger), or Attach (to start the debugger without re-programming first).



If you chose one of the debugging options, after clicking **Play**, your screen should look something like this. Your application has been programmed into the chip (if you chose the Launch option). The reset of the chip has happened, and the project has run until *main*. Notice the yellow highlighted line. It is waiting on you. You can now add breakpoints or get the program running or single step.



You can also run some of the ModusToolbox™ tools such as the Library Manager from inside VS Code. These will show up in the list of tasks as **Tool: <tool name>**.

2.3.2.1 Using workspaces in VS Code

If you want to be able to see the shared libraries or even see all the applications that are in your application root path at once, you can use a VS Code workspace. They are text files with the extension code-workspace.

Note: A workspace in VS Code is not the same as a workspace in Eclipse so don't confuse the two. The default workspace that is created for VS Code using ModusToolbox™ tools consists of the application plus the mtb_shared directory.

You can create and use workspaces in several different ways depending on what you want to see. You can launch the GUI first and then create a workspace or you can use the command line to launch the GUI and create the workspace as a single step.

If you want to see a single application along with the shared directory (e.g. application + mtb_shared) you can use a workspace that is created during `make vscode`. To use it, from the command line, go to the application's directory and run the following to launch the GUI and open the workspace:

```
code <workspace_name>
```

The workspace name ends in the extension `.code-workspace`.

If you already have the GUI running, use the following to open the workspace:

1. **File > Open Workspace...**
2. Navigate to the workspace file and click **Open**.

If you want to see all applications in your application root path, the easiest way to do this is to edit the workspace from one of your apps. The procedure is:

1. Copy/rename the workspace file from any application up one level (parallel with your apps and shared library repo).
2. Edit the workspace file to change the list of folders to match the list of applications and shared folder. You can add as many or as few applications as you want. Notice that the path to mtb_shared was changed. For example (only partial contents are shown):

Original:

```
"folders": [
    {
        "path": "."
    },
    {
        "path": "../mtb_shared"
    }
],
```

Modified:

```
"folders": [
    {
        "path": "ex01_blink"
    },
    {
        "path": "ex03_green"
    },
    {
```



```
        "path": "mtb_shared"  
    },  
],
```

3. Once you have done this, open the workspace as usual (i.e. `code <workspace name>` from the command line or **File > Open Workspace...** from the GUI).

A few notes about workspaces:

- You must run `make vscode` on any application that you include in your workspace so that it contains the required build information.
- If more than one directory is open in a workspace, the Run Task list and the Debug Launch list will show entries for all available task/launches. Be careful to choose the correct entry.
- If you quit and restart VS Code, any workspace that you didn't close will re-open. This may result in multiple instances of VS Code opening at the same time. To resolve this, use **File > Close Workspace**.
- When you close a workspace that you have created, you will be asked if you want to save it. If you chose to save your workspace, you can open it using one of these methods:
 - From the command line: `code <workspace_name>`
 - From the GUI: **File > Open Workspace...**
- If you create new applications after creating a workspace you can add them manually to your workspace file or if you have the workspace open in the GUI you can use **File > Add Folder to Workspace...**

2.3.3 Export for IAR Embedded Workbench

The ModusToolbox™ build system provides a make target for IAR Embedded Workbench. If you didn't generate the files during project creation you can run the following command:

```
make ewarm8 TOOLCHAIN=IAR
```

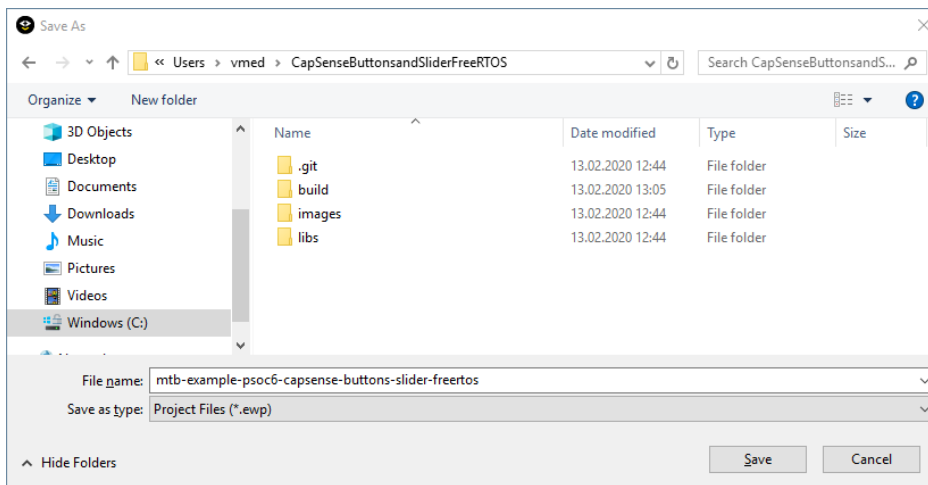
Note: You may want to change the value of TOOLCHAIN in the Makefile for the application rather than having to specify it on the command line.

An IAR connection file appears in the application directory:

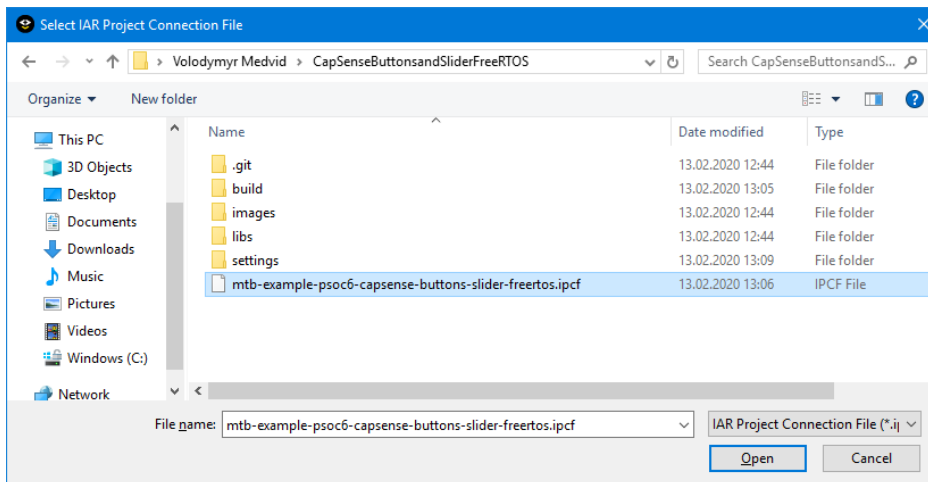
- `<project-name>.ipcf`

1. Start IAR Embedded Workbench.
2. On the main menu, select **Project > Create New Project > Empty project** and click **OK**.

3. Browse to the application directory, enter an arbitrary application name, and click **Save**.



4. After the application is created, select **File > Save Workspace**, enter an arbitrary workspace name and click **Save**.
5. Select **Project > Add Project Connection** and click **OK**.
6. On the Select IAR Project Connection File dialog, select the *.ipcf* file and click **Open**:



7. On the main menu, Select **Project > Make**.

At this point, the application is open in the IAR Embedded Workbench. There are several configuration options in IAR that we won't discuss here. For more details about using IAR, refer to the "Exporting to IDEs" chapter in the [ModusToolbox™ User Guide](#).

2.3.4 Export for Keil µVision

The ModusToolbox™ build system also provides a make target for Keil µVision. If you didn't generate the files during project creation you can run the following command:

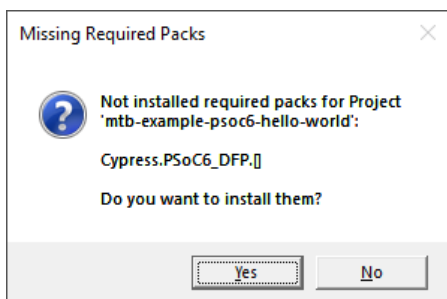
Note: You may want to change the value of TOOLCHAIN in the Makefile for the application rather than having to specify it on the command line.

This generates three files in the application directory:

- `<project-name>.cpdsc`
- `<project-name>.cprj`
- `<project-name>.gpdsc`

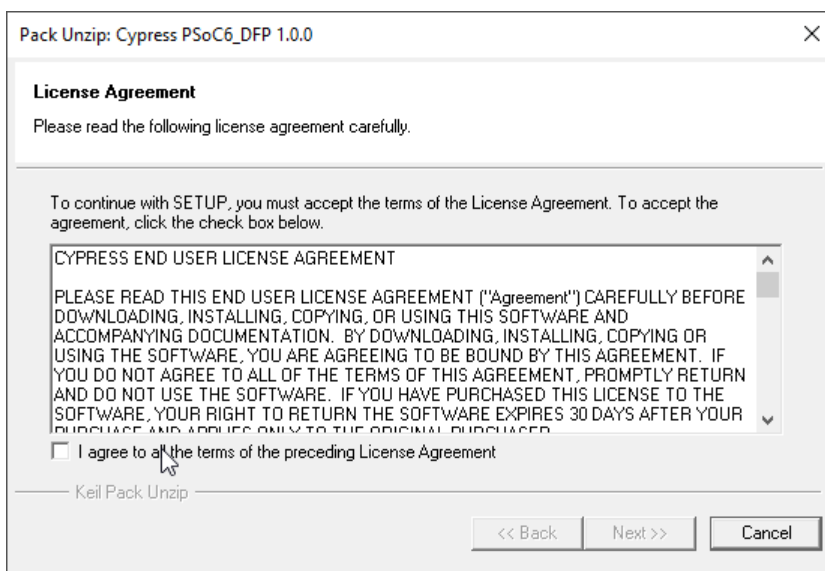
The `cpdsc` file extension should have the association enabled to open it in Keil µVision.

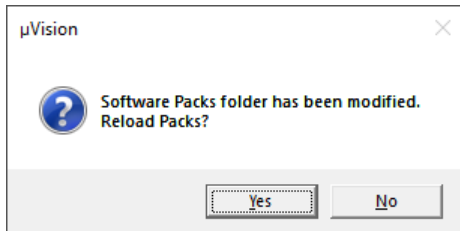
1. Double-click the `cpdsc` file. This launches Keil µVision IDE. The first time you do this, you may see a dialog similar to this:



Click **Yes** to install the device pack. You only need to do this once.

2. Follow the steps in the Pack Installer to properly install the device pack.





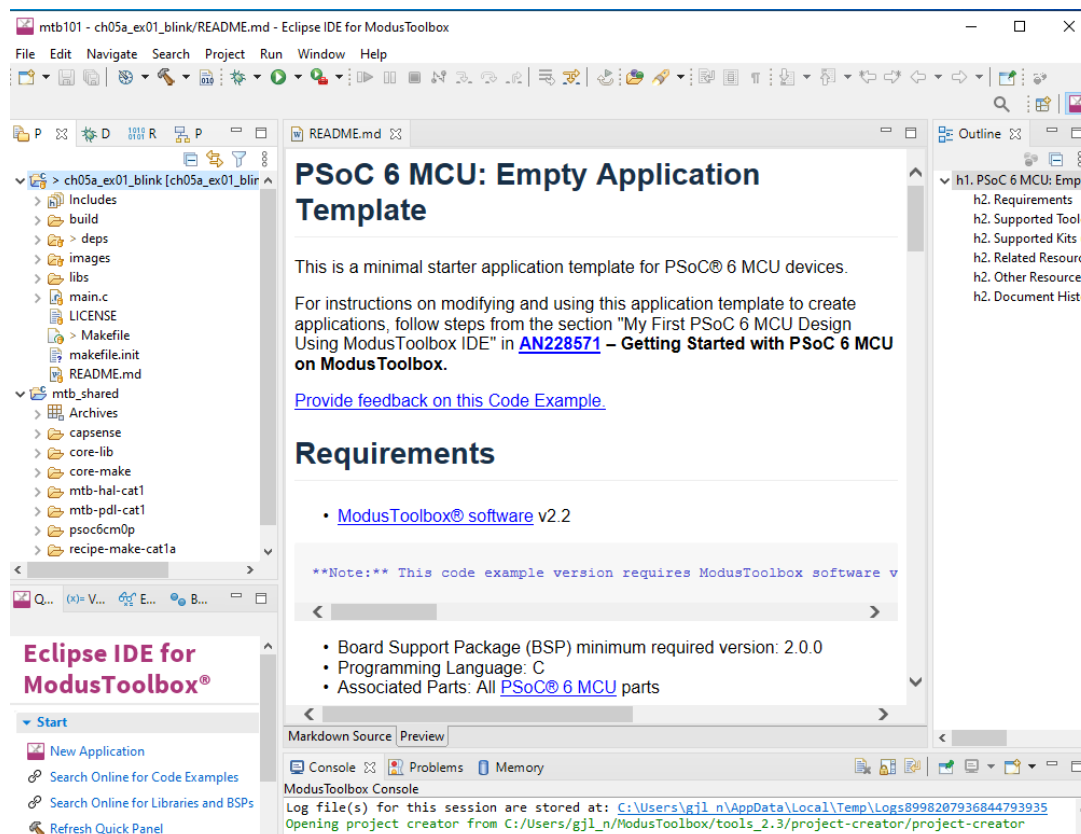
3. When complete, close the Pack Installer and close the Keil µVision IDE.
4. Then double-click the `.cpdsc` file again and the application will be created for you in the IDE.

At this point, the application is open in the Keil µVision IDE. There are several configuration options in µVision that we won't discuss here. For more details about using µVision, refer to the "Exporting to IDEs" chapter in the [ModusToolbox™ User Guide](#).

2.4 Eclipse IDE for ModusToolbox™ tour

The Eclipse IDE for ModusToolbox™ provided in the installation package has some customizations to make it easier to use with the ModusToolbox™ ecosystem. Eclipse still mostly works the way standard Eclipse does, but a short introduction is in order. This isn't a recommendation to use Eclipse instead of another IDE, but since the others are not customized for the ModusToolbox™ ecosystem they have their own documentation.

The Eclipse IDE for ModusToolbox™ uses several plugins, including the Eclipse C/C++ Development Tools (CDT) plugin. The IDE contains Eclipse standard menus and toolbars, plus various views such as the Project Explorer, Code Editor, and Console.



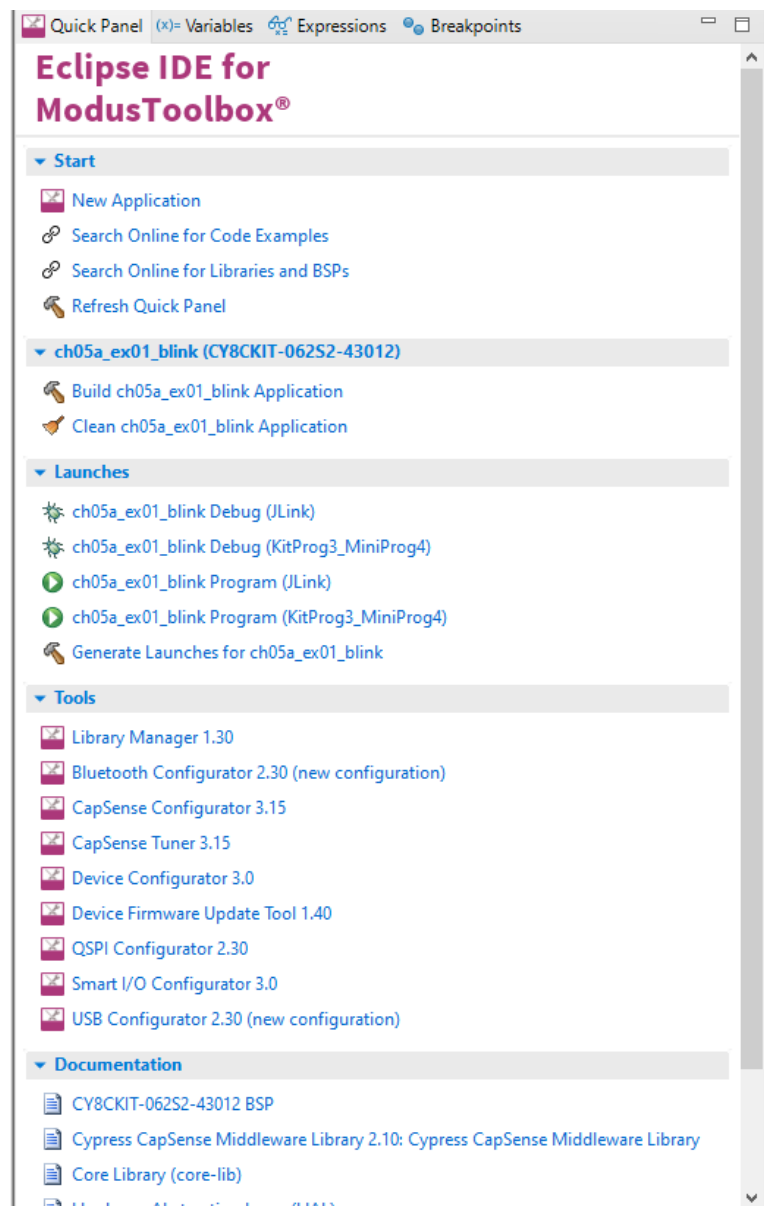
For more information about the IDE, refer to the "Eclipse IDE for ModusToolbox™ User Guide":

<http://www.cypress.com/MTBEclipseIDEUserGuide>

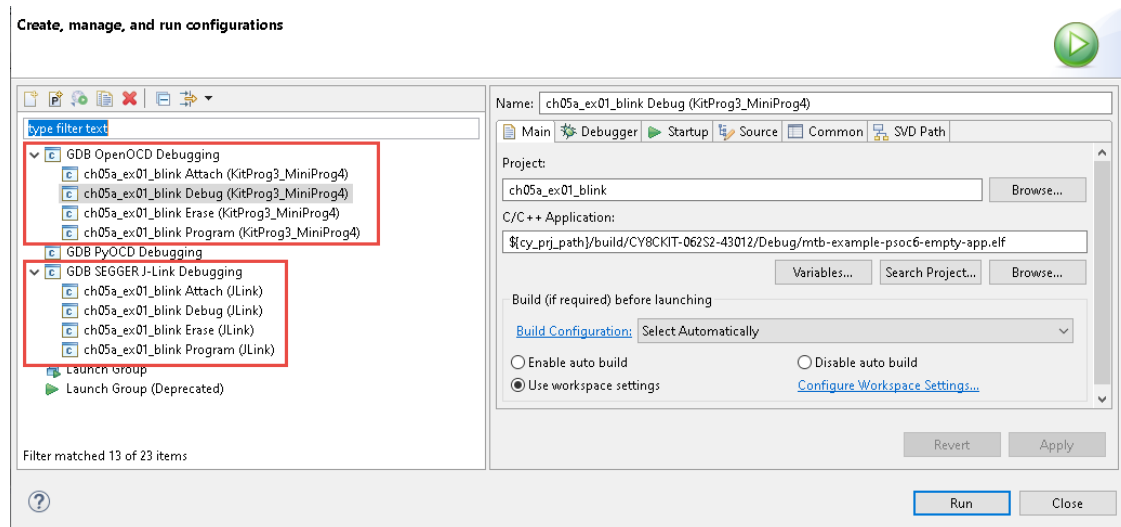
2.4.1 Quick Panel

We have extended the Eclipse functionality with a "ModusToolbox" Perspective. Among other things (such as adding a bunch of useful Debug windows), this perspective includes a panel with links to commonly used functions including:

- Create a New Application
- Clean & Build
- Program/Debug Launches
- Tools (Configurators)
- Documentation

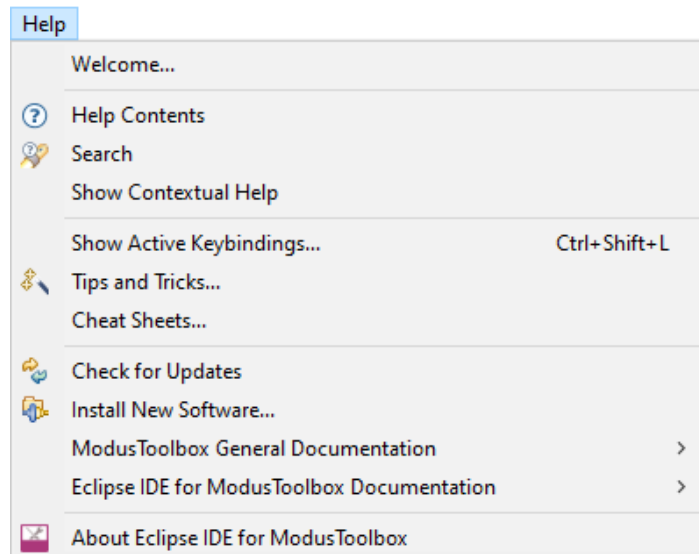


In addition to the launch configurations in the Quick Panel, there are usually others that you can find under **Run > Run Configurations > GDB OpenOCD Debugging** and **GDB SEGGER J-Link Debugging**. These typically include configurations to erase the chip or attach the debugger to a running target. You can also see exactly what the other launch configurations do from that menu.



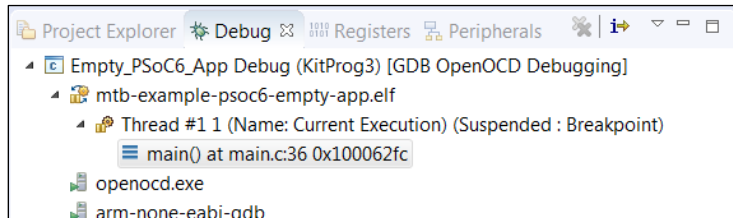
2.4.2 Help menu

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



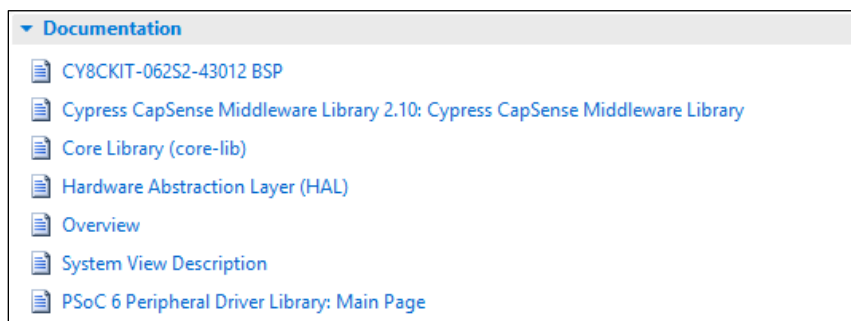
2.4.3 Integrated debugger

The Eclipse IDE provides an integrated debugger using either the KitProg3 or MiniProg4 for PSoC™ 6 MCU or a Segger J-Link debug probe for PSoC™ 6 MCUs and other devices. The exact choices available will depend on the device family.



2.4.4 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.

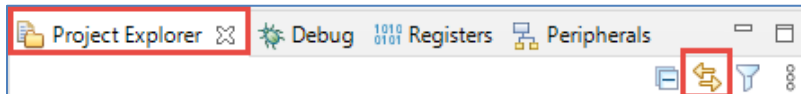


2.4.5 Eclipse IDE tips & tricks

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:
 - Show/hide line numbers
 - Set/clear breakpoints
- 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 sped up by turning off the "Link with Editor" feature. This prevents the active project from being changed every time you switch

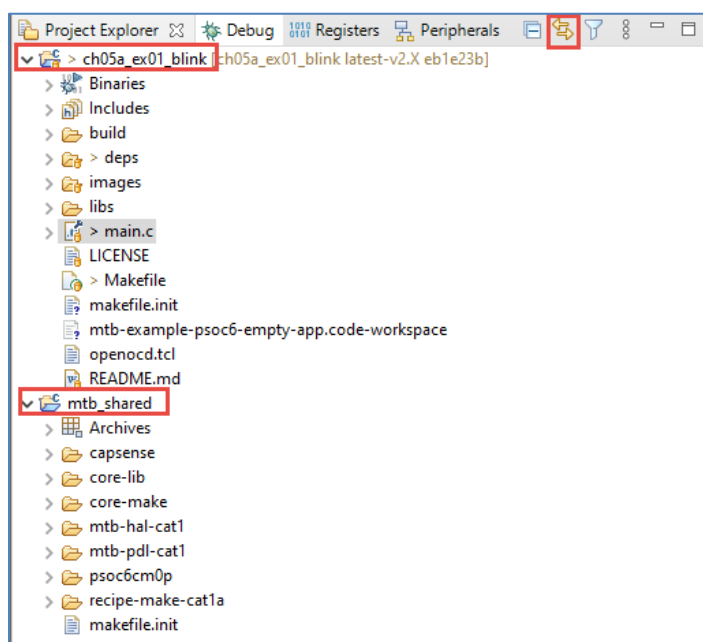
between files. It can be turned off using the icon with two arrows in the Eclipse Project Explorer window:



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

2.5 Application directory organization

A ModusToolbox™ application will contain one or more projects. For simple applications that only control one CPU core, there is usually just one project. In that case, the top-level application directory and project directory may be the same. There is usually another directory containing libraries that may be shared between applications. As an example, a typical PSoC™ 6 MCU application may look like this:



The exact files and directories may be different for other applications, device families or solutions but the basic structure will be the same. The directories and files in the example application's project directory include:

Directory/File	Description
<i>Binaries</i>	Virtual directory that points to the elf file from the build. This folder may not appear until a build is done.
<i>Includes</i>	Virtual directory that shows the include paths used to find header files.
<i>Build</i>	This directory contains build files, such as the .elf and .hex files. It may not appear until a build is done.
<i>deps</i>	<p>This directory contains <i>mtb</i> files that specify where the <code>make getlibs</code> command finds the libraries directly included by the project. Note that libraries included via <i>mtb</i> files may have their own library dependencies listed in the manifest files. We'll talk more about dependencies, <i>mtb</i> files and manifests later on.</p> <p>As you will see, the source code for libraries either go in the shared repository (for shared libraries) or in the <i>libs</i> directory inside the application (for libraries that are not shared).</p>

Directory/File	Description
<i>images</i>	This directory contains artwork images used by the documentation.
<i>libs</i>	<p>The <i>libs</i> directory contains source code for local libraries (i.e. those that are not shared) and <i>mtb</i> 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 <i>libs</i> directory. One notable exception is the BSP which is typically placed local to the app. The <i>libs</i> directory also includes a file called <i>mtb.mk</i> which specifies which shared libraries should be included in the application and where they can be found. This will be discussed in detail in the Library management section.</p> <p><i>Note:</i> As you will see, the <i>libs</i> directory only contains items that can be recreated by running <code>make getlibs</code>. Therefore, the <i>libs</i> directory should normally not be checked into a source control system.</p>
<i>main.c</i>	This is the primary application file that contains the application's code. Depending on the application, this file may have a different name or there may be multiple source code files. In more complex applications, the source code files might be in a subdirectory such as <i>source</i> .
<i>LICENSE</i>	This is the ModusToolbox™ software license agreement file.
<i>Makefile</i>	The <i>Makefile</i> is used in the application creation process. It defines everything that the ModusToolbox™ software needs to know to create/build/program the application. This file is interchangeable between Eclipse IDE and the Command Line Interface so once you create an application, you can go back and forth between the IDE and CLI at will. See Makefile build settings for details about editing the <i>Makefile</i> .
<i>makefile.init</i>	This file contains variables used by the make process. This is a legacy file not needed anymore.
<i>README.md</i>	Almost every code example includes a <i>README.md</i> (mark down) file that provides a high-level description of the application.

2.5.1 Makefile build settings

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=**)

```
# Target board/hardware (BSP).
# To change the target, it is recommended to use the Library manager
# ('make modlibs' from command line), which will also update Eclipse IDE launch
# configurations. If TARGET is manually edited, ensure TARGET_<BSP>.mtb with a
# valid URL exists in the application, run 'make getlibs' to fetch BSP contents
# and update or regenerate launch configurations for your IDE.
TARGET=CY8CKIT-062S2-43012
```

Build configuration (**CONFIG=**)

```
# Default build configuration. Options include:
#
# Debug -- build with minimal optimizations, focus on debugging.
# Release -- build with full optimizations
# Custom -- build with custom configuration, set the optimization flag in CFLAGS
#
# If CONFIG is manually edited, ensure to update or regenerate launch configurations
# for your IDE.
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=**)

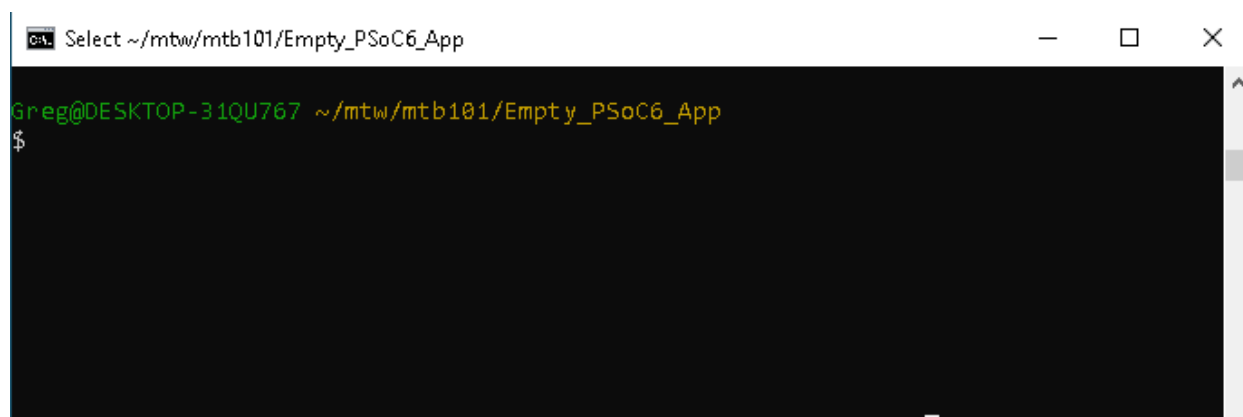
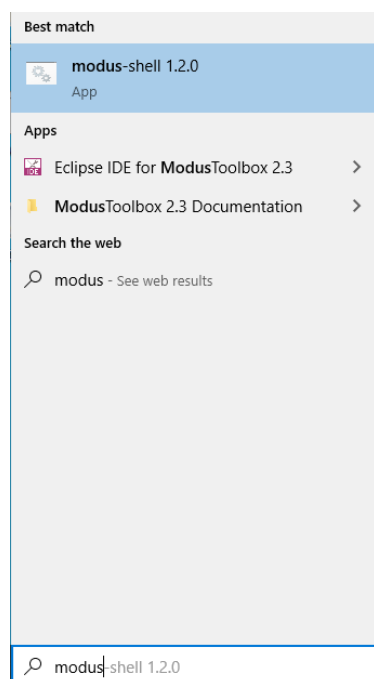
```
#####  
# Advanced Configuration  
#####  
  
# Enable optional code that is ordinarily disabled by default.  
#  
# Available components depend on the specific targeted hardware and firmware  
# in use. In general, if you have  
#  
#     COMPONENTS=foo bar  
#  
# ... then code in directories named COMPONENT_foo and COMPONENT_bar will be  
# added to the build  
#  
COMPONENTS=  
  
# Like COMPONENTS, but disable optional code that was enabled by default.  
DISABLE_COMPONENTS=
```

2.6 Using the command line (CLI)

Instead of (or in addition to) using an IDE, ModusToolbox™ tools provide a command line interface that interacts directly with the make system.

2.6.1.1 Windows

To use the CLI, 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 the ModusToolbox™ ecosystem includes "modus-shell" which is based on Cygwin. It is in the ModusToolbox™ tools installation under *ModusToolbox/tools_<version>/modus-shell*. It is listed in the **Start** menu under **ModusToolbox <version>** or you can just enter `modus-shell` in the Windows search box.



2.6.1.2 macOS / Linux

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

2.6.1.3 Make targets (commands)

In order to have the command line commands work, 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.

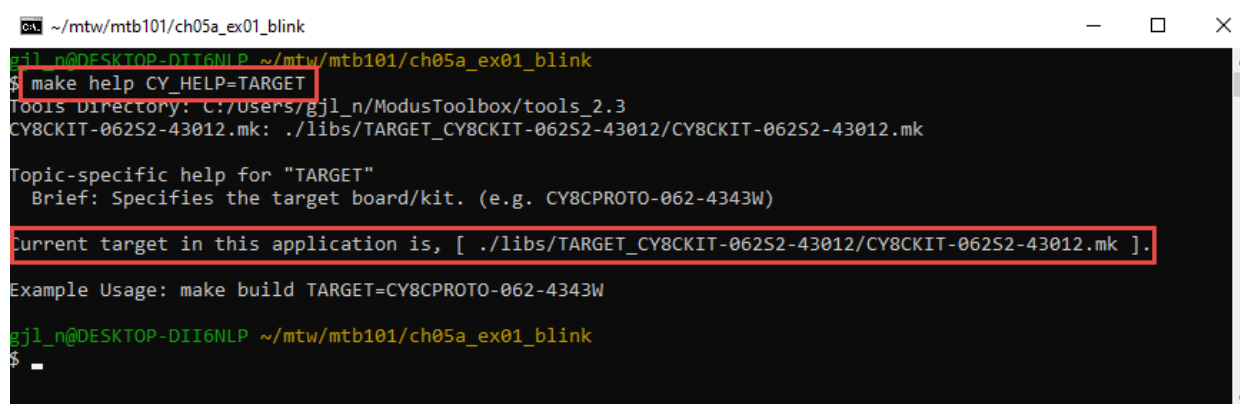
Make command	Description
<code>make help</code>	This command will print out a list of all the make targets. To get help on a specific target type <code>make help CY_HELP=getlibs</code> (or whichever target you want help with).
<code>make getlibs</code>	Process all the <i>mtb</i> files (or <i>lib</i> files) and bring all the libraries into your project. If you manually Git clone an application, this is typically the next step.
<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 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
```



```

~/mtw/mtb101/ch05a_ex01_blink
gjl_n@DESKTOP-DII6NLP ~/mtw/mtb101/ch05a_ex01_blink
$ make help CY_HELP=TARGET
Tools Directory: C:/Users/gjl_n/ModusToolbox/tools_2.3
CY8CKIT-062S2-43012.mk: ./libs/TARGET_CY8CKIT-062S2-43012/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, [ ./libs/TARGET_CY8CKIT-062S2-43012/CY8CKIT-062S2-43012.mk ].

Example Usage: make build TARGET=CY8CPROTO-062-4343W

gjl_n@DESKTOP-DII6NLP ~/mtw/mtb101/ch05a_ex01_blink
$

```

2.6.1.4 Accessing tools from the command line

There are make targets to launch some of the tools from an application's folder. Use `make help` and look for "Tools make targets" for the list. A couple useful examples:

- `make config` launches the Device Configurator and opens the application's configuration file.
- `make modlibs` launches the Library Manager and opens the application's library settings.

You can launch other ModusToolbox™ tools using `make open` and specifying a tool name or a file to open. Use `make help CY_HELP=open` for details. For example:

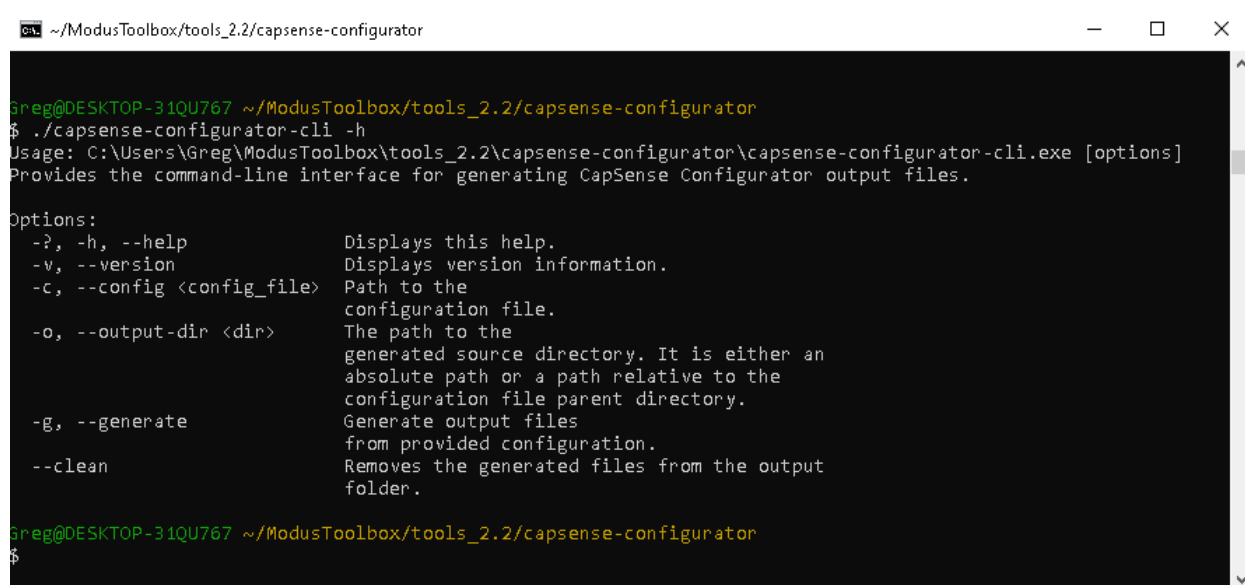
- `make open CY_OPEN_TYPE=capsense-tuner` launches the CAPSENSE™ Tuner and opens the application's *design.cycapsense* file.

Note: If you run `make open CY_OPEN_TYPE=junk` it displays a complete list of valid tool types.

As described earlier, all tools can also be run directly from the tools folder. On Windows you can use the **Start** menu entries or the search box.

For many tools, there is a command line (i.e. non-GUI) version that can be useful for scripting/automation. The `-h` option shows help for the CLI tools. For example:

```
./capsense-configurator-cli -h
```



```
~\ModusToolbox\tools_2.2\capsense-configurator

Greg@DESKTOP-31QU767 ~/ModusToolbox/tools_2.2/capsense-configurator
$ ./capsense-configurator-cli -h
Usage: C:\Users\Greg\ModusToolbox\tools_2.2\capsense-configurator\capsense-configurator-cli.exe [options]
Provides the command-line interface for generating CapSense Configurator output files.

Options:
-?, -h, --help           Displays this help.
-v, --version            Displays version information.
-c, --config <config_file> Path to the
                           configuration file.
-o, --output-dir <dir>   The path to the
                           generated source directory. It is either an
                           absolute path or a path relative to the
                           configuration file parent directory.
-g, --generate           Generate output files
                           from provided configuration.
--clean                 Removes the generated files from the output
                           folder.

Greg@DESKTOP-31QU767 ~/ModusToolbox/tools_2.2/capsense-configurator
$
```

2.7 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.

2.7.1 Library classification

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

2.7.1.1 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.

2.7.1.2 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.

2.7.1.3 Fixed vs. dynamic versions

You can specify fixed version of a library to use in your application (e.g. 1.1.0), or you can specify a major release version with the intent that you will get the latest compatible version (e.g. Latest v1.X). By default, when you create an application the libraries will be fixed versions so that versions of libraries in a given application will never change unless you specifically request a change. This is called "latest locking" and it is done automatically during project creation based on information in the manifest file.

2.7.1.4 Specifying dependencies

In the ModusToolbox™ 2.2 tools release, the method by which applications reference libraries was modified such that applications typically share libraries by default. The main exception is BSPs which are placed local to the application. 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 libraries are stored in a new *mtb_shared* directory adjacent to your application directories. This method of specifying and locating dependencies is called the MTB flow.

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

Prior to the ModusToolbox™ 2.2 tools release, a different library management method (called the LIB flow) was in use. Since it is only used for old designs, it will not be discussed in this training. You can tell which flow an application uses by looking in its *deps* directory. If there are files with the extension *mtb* then it uses the MTB flow. If there are files with the extension *lib*, then it uses the LIB flow.

*Note: The library management method used for a single application is a binary choice. If an application uses the MTB flow, only *mtb* files will be processed, and *lib* files will be ignored.*

As mentioned above, the source code for all libraries except BSPs will be put in a shared directory by default. (You'll see how to make them local 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 the *libs* directory inside the application.

ModusToolbox™ tools know 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/retarget-io/#latest-  
v1.X##$ASSET_REPO$/retarget-io/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 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 (more on that in the [Manifests](#) section). 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	<code>./deps/</code>	<code>./deps/</code>	<code>./libs/</code>
library source code	<code>../mtb_shared/</code>	<code>./libs/</code>	<code>../mtb_shared/</code>

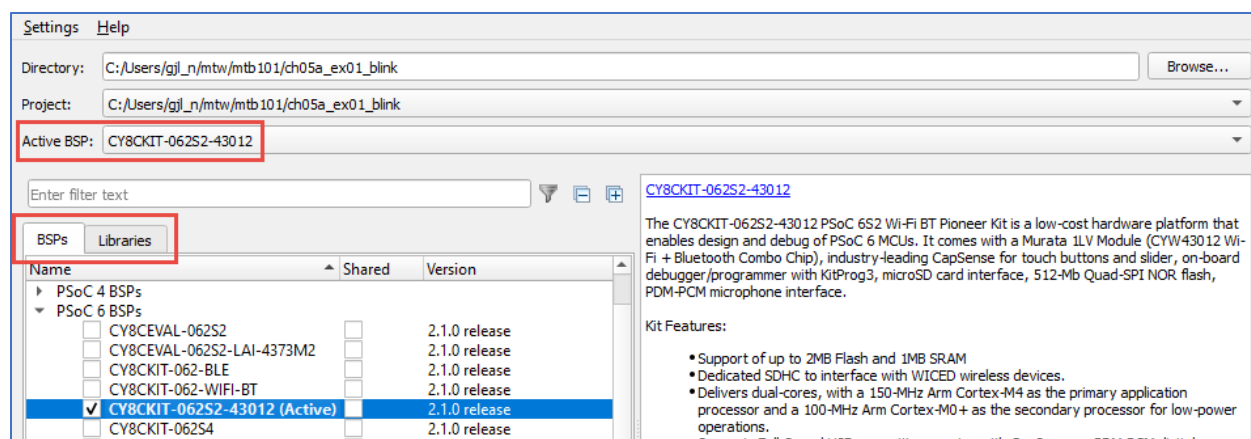
2.7.2 Library Manager tool

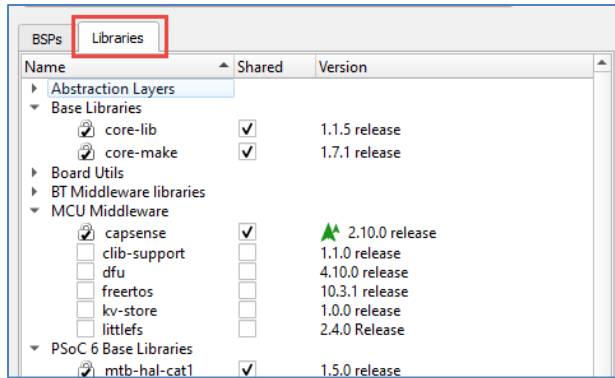
The ModusToolbox™ ecosystem provides a GUI for helping you manage library files. You can run this from the Quick panel link "Library Manager". It is available outside the IDE from *ModusToolbox/tools_<version>/library-manager*, from the Windows **Start** menu, or by entering `library-manager` in the Windows search box. 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, which will be discussed in the [Manifests](#) section. 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.

If, for some reason, you don't want to use the Library Manager GUI for a library that is in a manifest file, you can create an *mtb* file (or a *lib* file for the LIB flow) in the *deps* directory of your application and then run `make getlibs` from the root directory of your application. This will do the exact same thing that the Library Manager does.

The Library Manager GUI looks like. The tabs and windows will be slightly different for the LIB flow.





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*). The main exception is *BSPs* which are usually local by default. If you uncheck the Shared box for a given library, its source code will be copied to the *libs* directory in the application itself.

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. If you have a specific version of a library selected and there's a newer one available, one of the following symbols will appear next to the version number:

▲ 1.1.0 release ▲▲ 2.10.0 release

The single green arrow indicates that there is a new minor version of the library available, while two green arrows indicates that there is a new major release of the library available.

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* folder). 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. You can always do this for yourself from the command line.

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.

2.7.2.1 Eclipse IDE

Manually changing the `TARGET` in the *Makefile* will change the project being built by the IDE, but it will **NOT** update the Eclipse launch configs, so it does not affect which hex file is copied to the kit. Therefore, it is recommended to use the Library Manager's **Active BSP** selection to change the target board if you are using the IDE to program/debug. If you do edit the *Makefile* manually, there is a link in the Quick Panel that says "Generate Launches for <app-name>" that you can use to fix them.

2.7.2.2 VS Code

In the case of VS Code, after updating the Active BSP or after manually editing the `TARGET` in the *Makefile*, you must update the appropriate launch configurations. You can re-run `make vscode` to regenerate the launch configurations, but keep in mind that this may over-write other VS Code settings, so it is safer to update them manually. When it is re-run, the `make vscode` process will create a backup copy of the previous files in `.vscode/backup`.

2.7.3 Manual library management

If a library is not in a manifest file, it will not appear in the Library Manager and will need to be managed manually. This is a two-step process:

1. Acquire the library
2. Acquire the library's dependencies

These steps can be done several different ways depending on the construction of the library. Note that for (custom) BSPs, the Project Creator tool simplifies this process greatly. You can still use the other methods on BSPs if you want to include a custom BSP into an existing application. We will discuss how to create a custom BSP in [Creating your own BSP](#).

Note that by manually managing libraries, all dependencies will be included directly in the application. Indirect dependencies only occur when libraries and dependencies are included in manifest files as described in the [Error! Reference source not found.](#) section.

2.7.3.1 Acquiring a library

There are three basic ways to get a library that isn't in a manifest file using the MTB flow:

Requirements	Method
Library is hosted in a Git repo	<ul style="list-style-type: none">• Create an <i>mtb</i> file for the library in the application's deps directory.• Run <code>make getlibs</code> from the application's root directory.• The library can be local or shared based on the <i>mtb</i> file contents.
Any library Library is local to app	<ul style="list-style-type: none">• Use <code>copy</code>, <code>git clone</code>, or use some other revision control system to pull the library into the application. It can be located anywhere except the <i>libs</i> directory.
Any library Library is in a shared location	<ul style="list-style-type: none">• Use <code>copy</code>, <code>git clone</code>, or use some other revision control system to pull the library into a shared location.• Edit the <code>SEARCH</code> variable in the application's <i>Makefile</i> to point to the library.
Custom BSP	<ul style="list-style-type: none">• Use <code>copy</code>, <code>git clone</code>, or use some other revision control system to get the BSP somewhere on disk if it isn't already. The location is not important. It will be copied into the application folder during project creation.• Use the Import function to select the BSP in Project Creator. The custom BSP will be copied into the application.

Once you have a library, the next step is to get its dependencies.

2.7.3.2 Acquiring dependencies

There are several ways to acquire dependencies:

Requirements	Method
Library contains <i>mtbx</i> files for its dependencies Dependencies can be local or shared	<ul style="list-style-type: none"> Run <code>make import_deps</code> followed by <code>make getlibs</code> from the application's root directory. (See <code>import_deps</code> details below)
Library does not contain <i>mtbx</i> files but does contain <i>lib</i> files for its dependencies Dependencies will be local	<ul style="list-style-type: none"> Run <code>make lib2mtbx</code>, then <code>make import_deps</code> and finally <code>make getlibs</code> from the application's root directory. (See <code>lib2mtbx</code> and <code>import_deps</code> details below)
Dependencies are in a manifest	<ul style="list-style-type: none"> Use the Library Manager.
Dependencies are hosted in Git repos	<ul style="list-style-type: none"> Create an <i>mtb</i> file for each dependency in the application's <code>deps</code> directory. Run <code>make getlibs</code> from the application's root directory. The dependencies can be local or shared based on the <i>mtb</i> file contents.
Any Dependency Dependency is local to app	<ul style="list-style-type: none"> Use <code>copy</code>, <code>git clone</code>, or use some other revision control system to pull the library into the application. It can be located anywhere except the <i>libs</i> directory.
Any dependency Dependency is in a shared location	<ul style="list-style-type: none"> Use <code>copy</code>, <code>git clone</code>, or use some other revision control system to pull the library into a shared location. Edit the <code>SEARCH</code> variable in the application's <i>Makefile</i> to point to the library.
Custom BSP	<ul style="list-style-type: none"> Dependencies are pulled automatically by Project Creator.

The `make import_deps` target is intended specifically to acquire dependencies in an application for libraries that don't have their dependencies specified manifest files. To use it, you need an *mtbx* file in the library for each dependency. The format of *mtbx* files is identical to *mtb* files. When you run `make import_deps`, it will copy the *mtbx* files from the library into the application's `deps` folder while also changing the extension to *mtb*. This results in the dependencies being direct to the application which are then pulled in when `make getlibs` is run. The libraries can either be local or shared depending on the contents of the *mtbx* files.

Usage:

```
make import_deps IMPORT_PATH=<path_to_library>
```

The `lib2mtbx` target is mainly intended for libraries (or BSPs) that were created for ModusToolbox™ 2.0 or 2.1 applications that do not contain *mtbx* files but do contain *lib* files for their dependencies. When you run `make lib2mtbx`, it will create *mtbx* files in the library based on the *lib* files in the library. The *mtbx* files will be created to place the dependencies local to the app unless the optional argument "shared" is specified. You can save the *mtbx* files to the library's source control so that you don't need to do this step the next time the library is included into an application.

Usage:

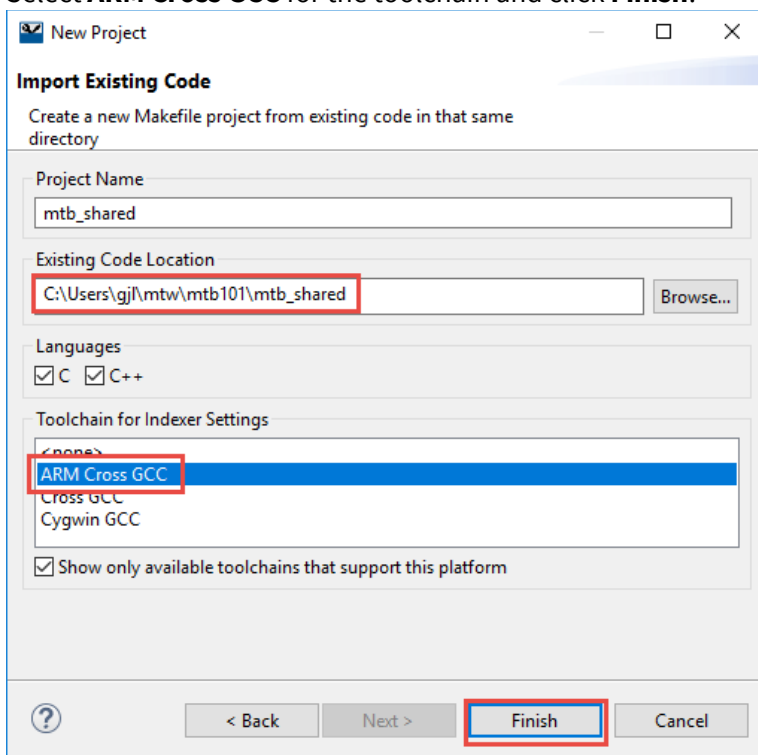
```
make lib2mtbx CONVERSION_PATH=<path_to_library> [CONVERSION_TYPE="shared"]
```

2.7.4 Re-Downloading Libraries

The local library directory (*libs*) and shared library repo (*mtb_shared*) can be deleted at any time since they can be re-created using either `make getlibs` from the command line or using **Update** in the Library Manager. Typically, those directories should NOT be checked into a revision control system since they should only contain libraries that are already controlled and versioned.

If you delete the *mtb_shared* directory from disk, you can easily re-acquire the libraries by using the Library Manager update function or by using `make getlibs` on an application, but the *mtb_shared* directory will not have Eclipse project information, so it may not be possible to open it from inside the Eclipse IDE and it may not work properly for IntelliSense. To fix, follow these steps from inside Eclipse after regenerating *mtb_shared* using `make getlibs` or the Library Manager **Update** function.

1. From the Project Explorer window, right-click on *mtb_shared* and select **Delete**. Do NOT select the check box **Delete project contents on disk** (if you do, you will have to regenerate it again).
2. Select **File > Import > C/C++ > Existing Code as Makefile Project** and click **Next >**.
3. **Browse** to the *mtb_shared* directory and click **Select Folder**.
4. The **Project Name** will be filled in automatically.
5. Select **ARM Cross GCC** for the toolchain and click **Finish**.



6. To get IntelliSense to work again for an application you must re-build it first.

2.8 Configurators

ModusToolbox™ software provides graphical applications called configurators that make it easier to configure hardware blocks and libraries. 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 included as part of the ModusToolbox™ 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 hardware block and libraries

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

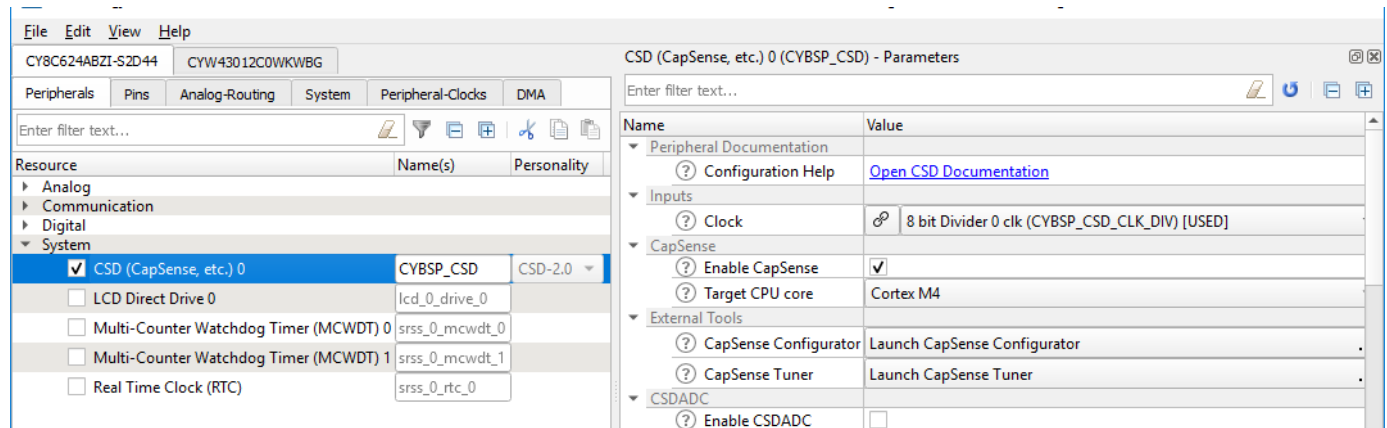
2.8.1 BSP configurators

BSP Configurators are closely related to the hardware resources on the board such as CAPSENSE™ sensors, external Flash memory, etc. As described earlier, these files are part of the BSP so 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 a standard BSP. Instead, you can create a custom configuration for a single application (see [Modifying the BSP Configuration \(e.g. design.modus\) for an Application](#)) or you can create a custom BSP (see [Creating your own BSP](#)).

Note: All BSP configurators rely on information from the device configurator's files. Therefore, you can only edit/save changes in one BSP configurator at a time. If you open a BSP configurator other than the device configurator and then try to launch the device configurator, you will get an error. If you launch another BSP configurator from inside the device configurator, the device configurator will be greyed out until the other configurator is closed.

2.8.1.1 Device configurator

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

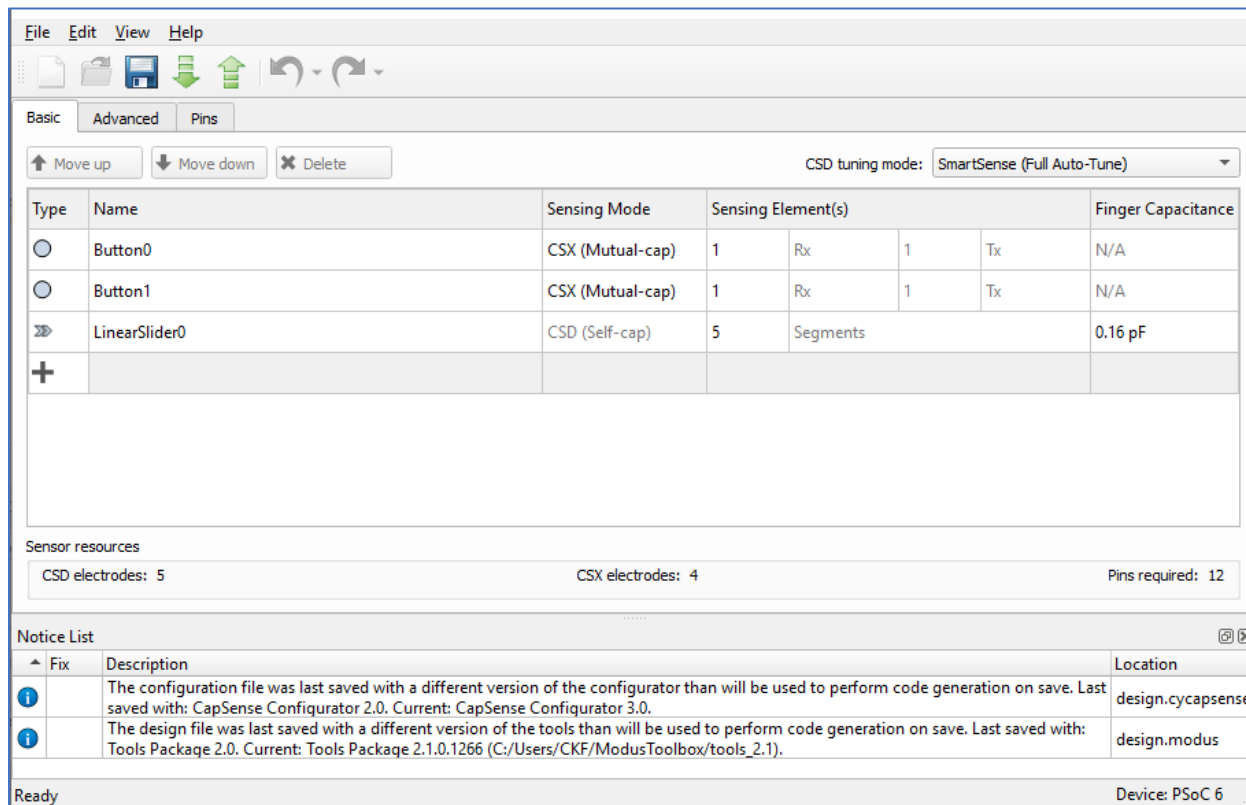


Hint: The + and - buttons can be used to expand/contract all categories and the filter button can be used to show only items that are selected. This is particularly useful on the Pins tab since the list of pins is sometimes quite long.

All other BSP configurators can be launched either stand-alone or from inside the device configurator. To launch from inside the device configurator, enable the appropriate block and click the button in the Parameters pane. For example, for the CAPSENSE™ configurator, enable **Peripherals > CSD** and then click the **Launch CapSense Configurator** button as you can see above.

2.8.1.2 CAPSENSE™ configurator and tuner

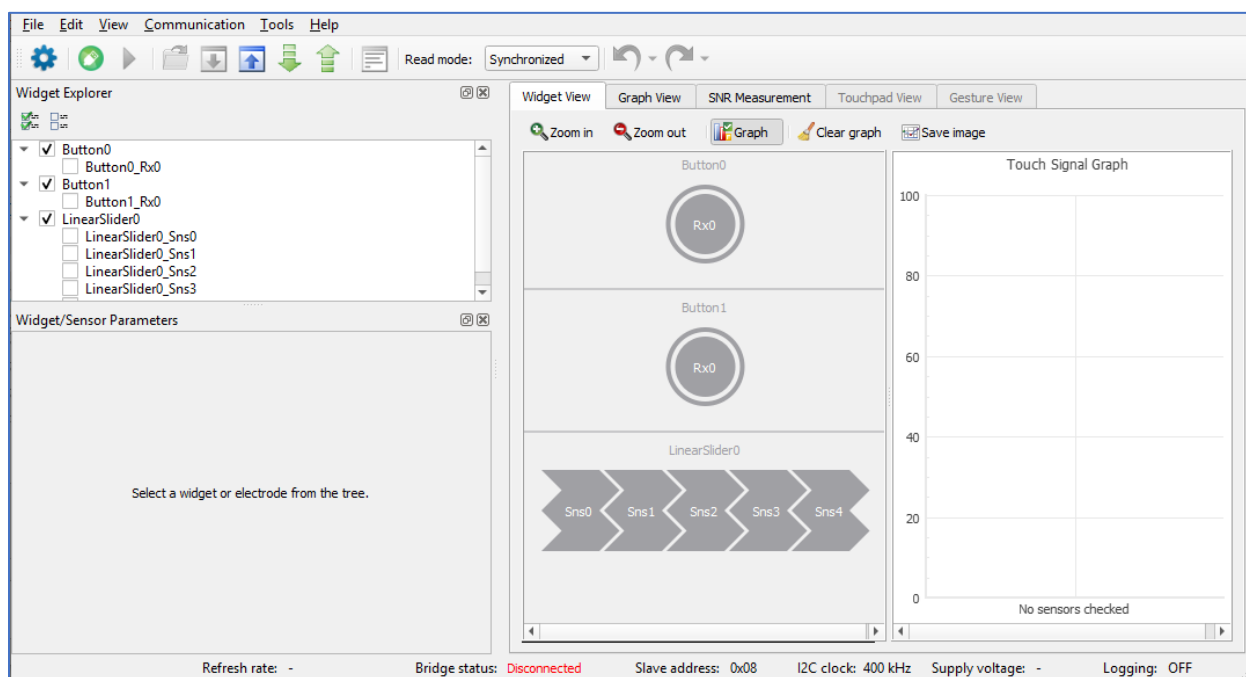
The CAPSENSE™ block includes a configurator and tuner. The configurator allows the user to setup sensor configuration, pin mapping, scan parameters, etc. The tuner allows the user to view sensor data real time from the hardware to understand and improve the performance.



The CAPSENSE configurator interface shows a table of sensor configurations. The table has columns for Type, Name, Sensing Mode, Sensing Element(s), and Finger Capacitance. The configurations are as follows:

Type	Name	Sensing Mode	Sensing Element(s)	Finger Capacitance
<input type="radio"/>	Button0	CSX (Mutual-cap)	1 Rx 1 Tx	N/A
<input type="radio"/>	Button1	CSX (Mutual-cap)	1 Rx 1 Tx	N/A
<input checked="" type="radio"/>	LinearSlider0	CSD (Self-cap)	5 Segments	0.16 pF

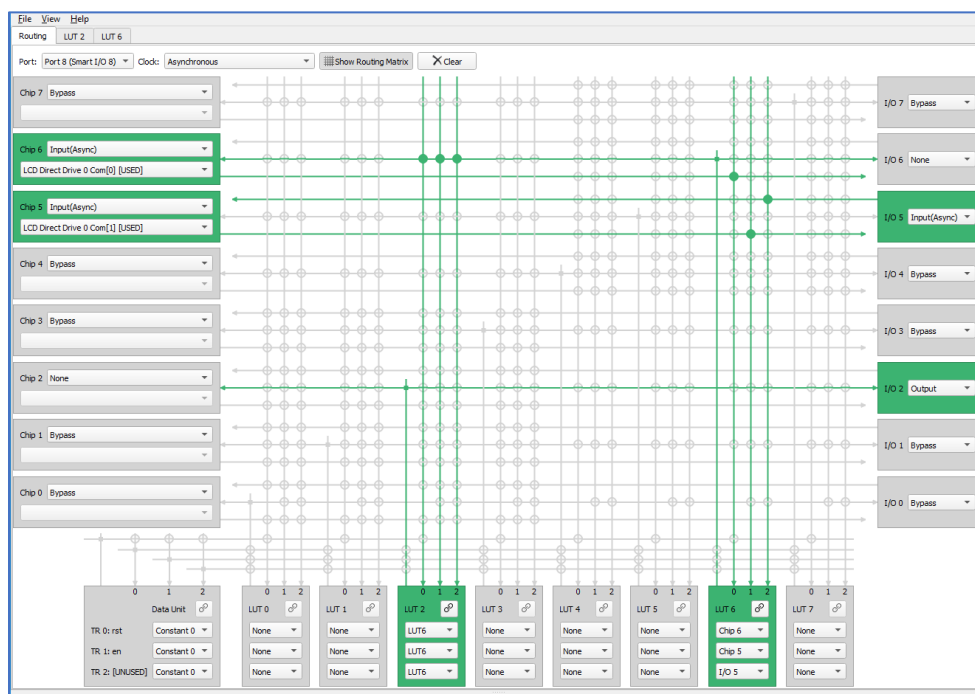
Below the table, the sensor resources are listed: CSD electrodes: 5, CSX electrodes: 4, and Pins required: 12. The CSD tuning mode is set to SmartSense (Full Auto-Tune). The Notice List at the bottom shows two messages about configuration file versions. The status bar at the bottom indicates the device is PSoC 6.



The CAPSENSE tuner interface shows a graphical representation of the sensor configuration. The Widget Explorer on the left lists the widgets: Button0, Button1, and LinearSlider0, along with their respective electrodes. The main area displays a Touch Signal Graph with a Y-axis ranging from 0 to 100. The graph shows no data points, indicating that no sensors have been checked. The status bar at the bottom shows the bridge status as Disconnected, slave address as 0x08, I2C clock as 400 kHz, supply voltage as -, and logging as OFF.

2.8.1.3 Smart I/O configurator

Configure the Smart I/O block, which adds programmable logic to an I/O port. Not all MCUs have Smart I/O and for those that do, there are usually only a few ports with Smart I/O capability, so it is important to consider that up front. For example, on the device 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 port is chosen via a drop-down along the top of the window. Also selected at the top of the window is the clocking scheme. It can be asynchronous, or a variety of clock sources can be selected.

Along the left edge you will see a set of 8 chip connection points. These can be inputs to Smart I/O from chip peripheral outputs such as TCPWM signals or they can be outputs from Smart I/O to chip peripheral inputs such as SCB inputs.

The right edge allows connections to the chip's pins for that port - in our case, P8[7:0] and P9[7:0].

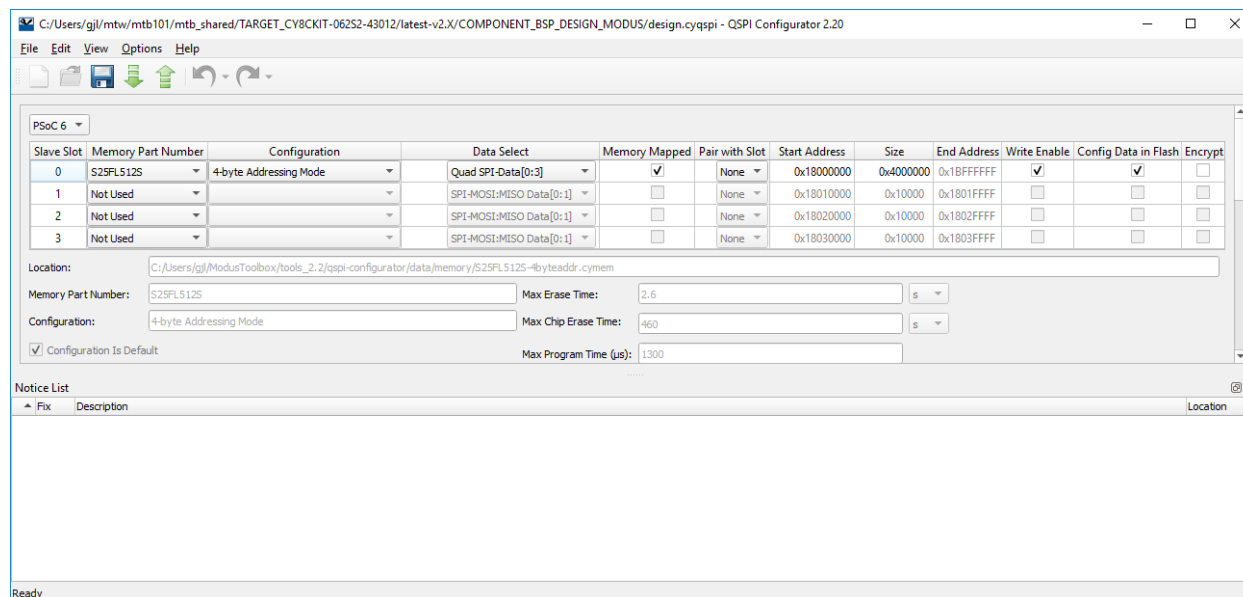
The bottom edge contains LUTs which can be used to combine and route signals to/from the chip's peripherals and to/from the chip's pins. The LUT inputs can come from peripherals, I/O pins, or from another LUT. Once you enable a LUT by selecting its inputs, a tab along the top appears for you to setup the truth table for the LUT. Note that if you don't need 3 inputs to a LUT, you can assign more than one input to the same signal.

The final section is the 8-bit Data Unit at the lower left corner. It allows you to setup trigger conditions that perform operations such as increment, decrement, and shift which are specified on the Data Unit tab once it is enabled by selecting one or more inputs.

Additional information is available in the Smart I/O user guide which can be found from the Help menu. You can also search for Smart I/O code examples in the usual places (In the Project Creator starter application list or on the Cypress GitHub site).

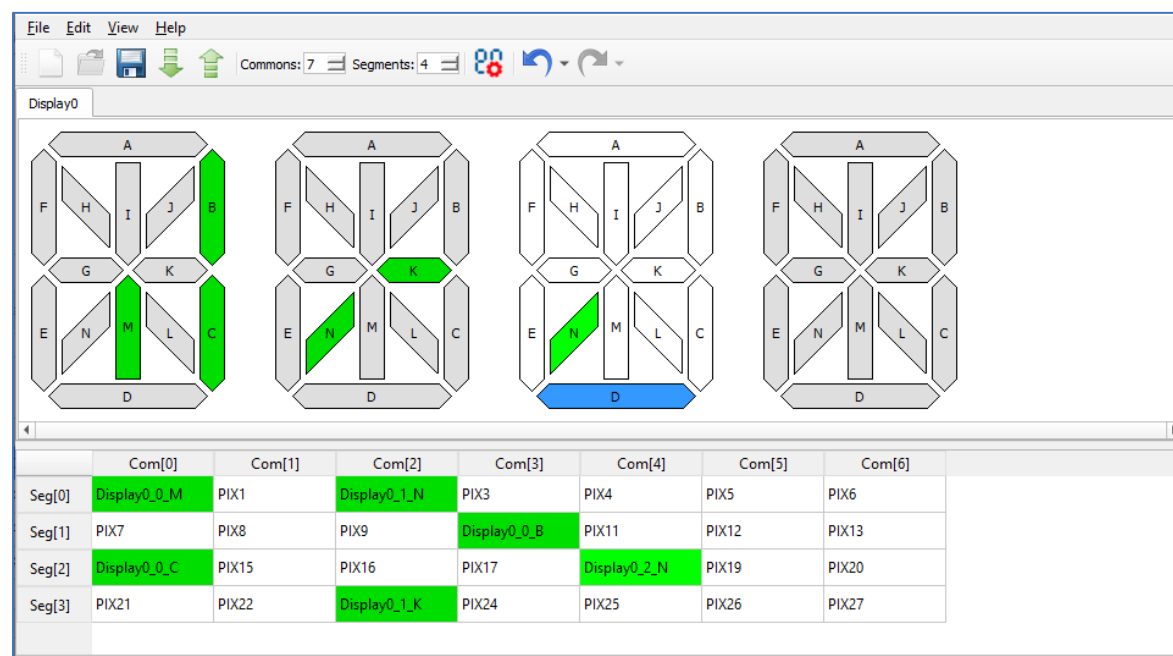
2.8.1.4 QSPI configurator

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



2.8.1.5 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.

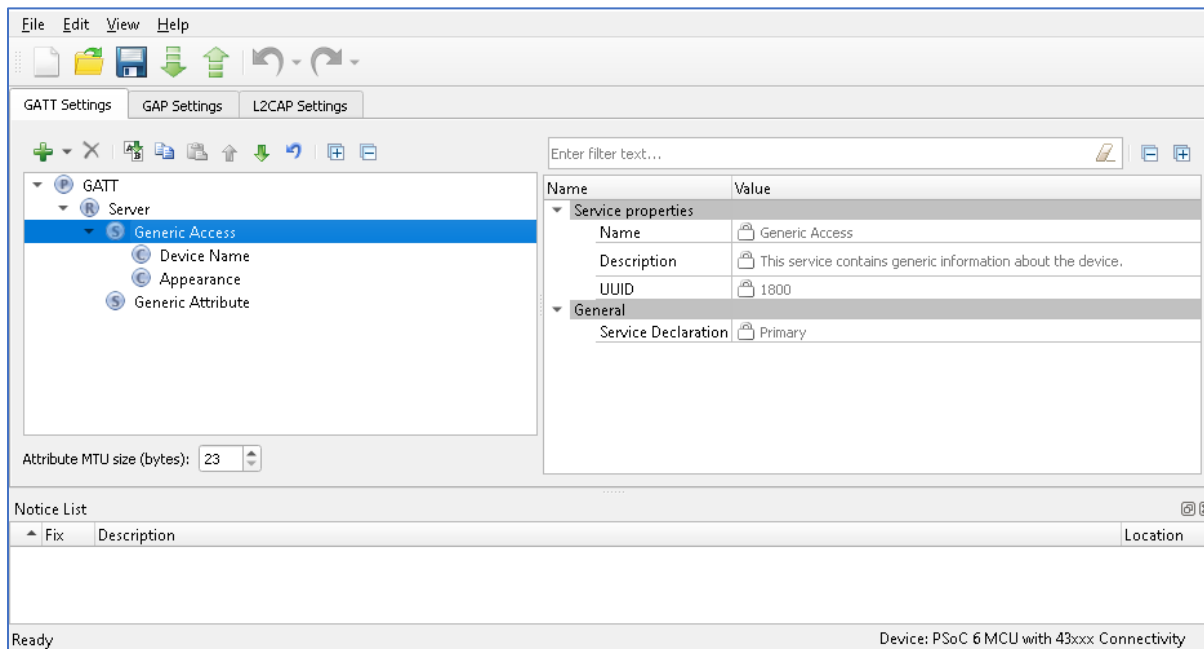


2.8.2 Library configurators

Library Configurators are used to setup and configure some software libraries. The files for these are contained in the project hierarchy rather than inside the BSP. Other configurators may be available depending on the device you are using. Each of the configurators has its own documentation accessible from the **Help** menu inside the tool.

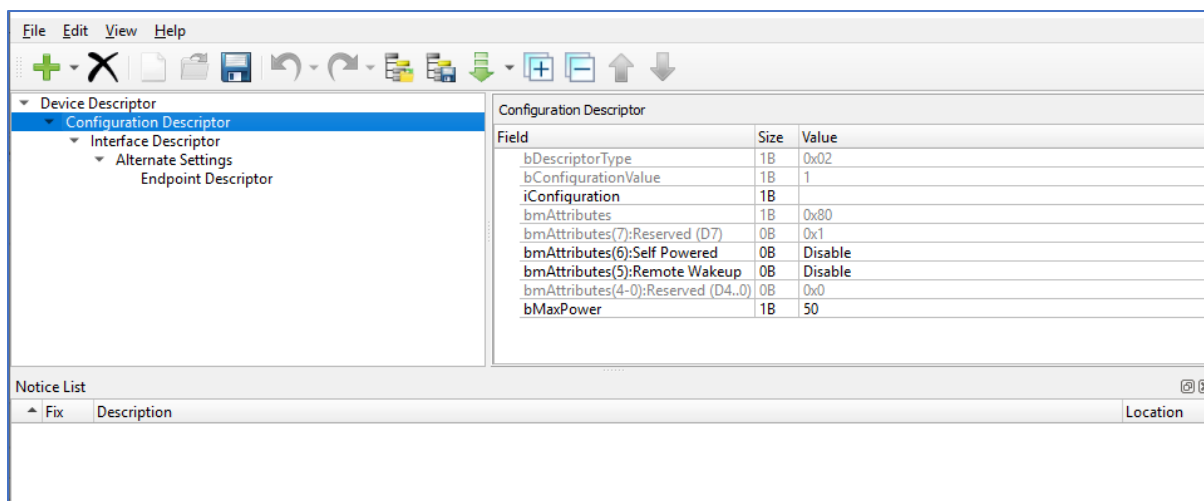
2.8.2.1 Bluetooth® configurator

Configure Bluetooth® settings. This includes options for specifying what services and profiles to use and what features to offer by creating GATT databases in generated code.



2.8.2.2 USB configurator

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

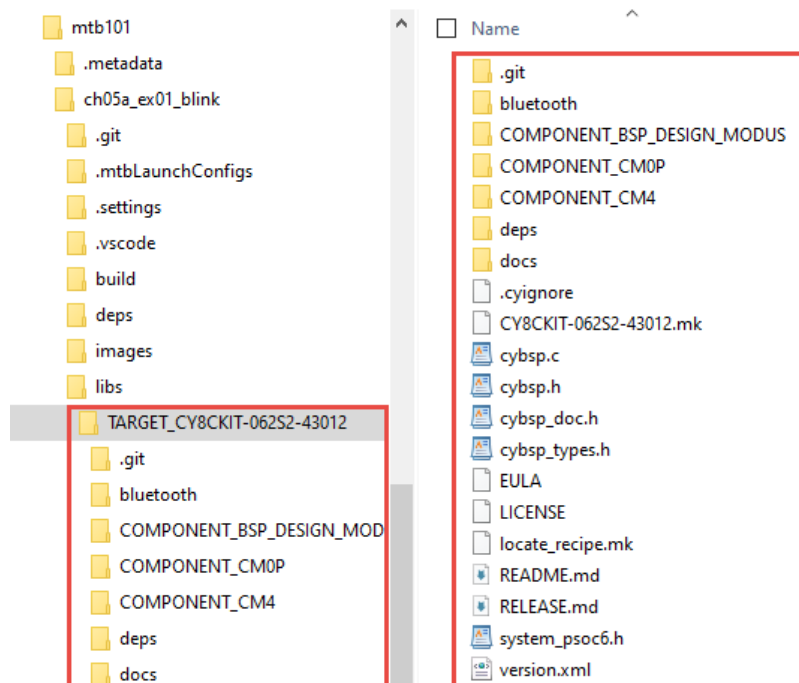


2.9 Board Support Packages (BSPs)

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 *libs* directory (by default) because the BSP is local to the application. This allows you to modify the BSP configuration for an application without affecting all other applications in the same workspace.

2.9.1 BSP directory structure

The following image shows a typical directory structure for a PSoC™ 6 MCU BSP.



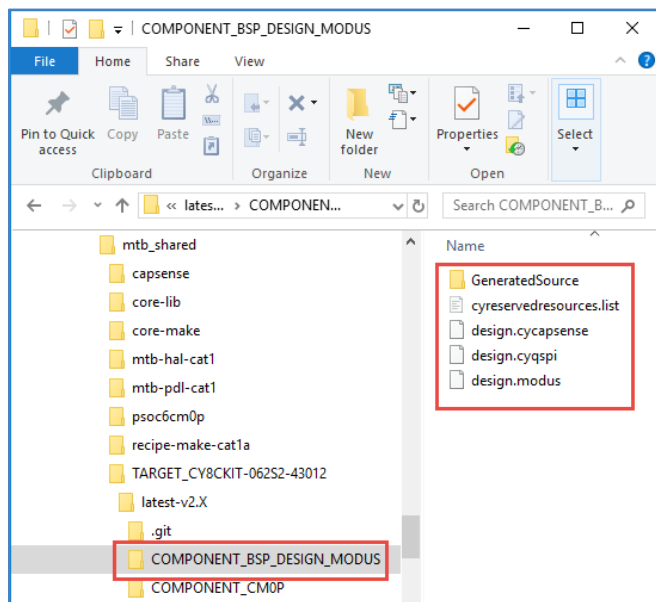
The various directories and files included in a BSP include:

Directory/File	Description
<i>COMPONENT_BSP_DESIGN_MODUS</i>	This directory contains configuration and generated source files. See COMPONENT_BSP_DESIGN_MODUS Contents for more details about this directory.
<i>COMPONENT_CM4</i> and <i>COMPONENT_CM0P</i>	These directories contain startup code and linker scripts for all supported toolchains for each of the two cores - the CM4 and the CM0+.
<i>Docs</i>	The <i>docs</i> directory contains the HTML based documentation for the BSP. See BSP .
<i>Deps</i>	The <i>deps</i> directory contains <i>lib</i> files for libraries that the BSP requires as dependencies. These are only used for the application flow that uses <i>lib</i> files. For the flow that uses <i>mtb</i> files, the dependency information is contained in a manifest file and the <i>lib</i> files are ignored.

Directory/File	Description
<i>cybsp_types.h</i>	The <i>cybsp_types.h</i> 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.
<i>cybsp.h</i> / <i>cybsp.c</i>	These files contain the API interface to the board's resources. You need to include only <i>cybsp.h</i> in your application to use all the features of a BSP. Call the <i>cybsp_init</i> function from your code to initialize the board (that function is defined in <i>cybsp.c</i>).
<targetname>.mk	This file defines the DEVICE and other BSP-specific make variables such as COMPONENTS.
<i>locate_recipe.mk</i>	This file provides the path to the core and recipe make files that are needed by the build system.
<i>system_psoc6.h</i>	For PSoC™ 6 MCU BSPs, this file contains device system header information. The file name will be different for other device families.

2.9.2 COMPONENT_BSP_DESIGN_MODUS Contents

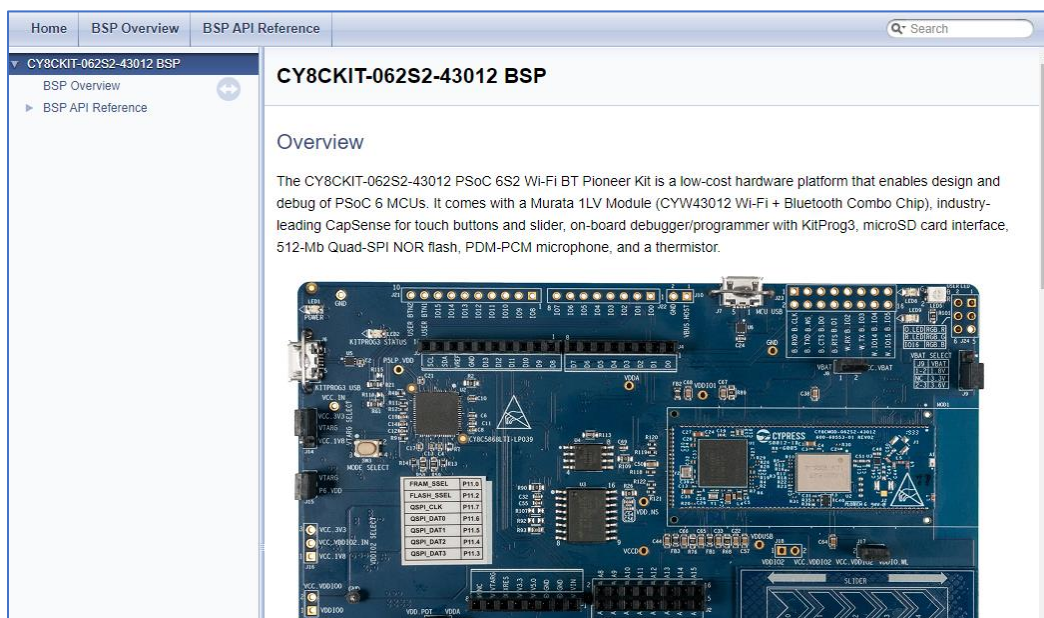
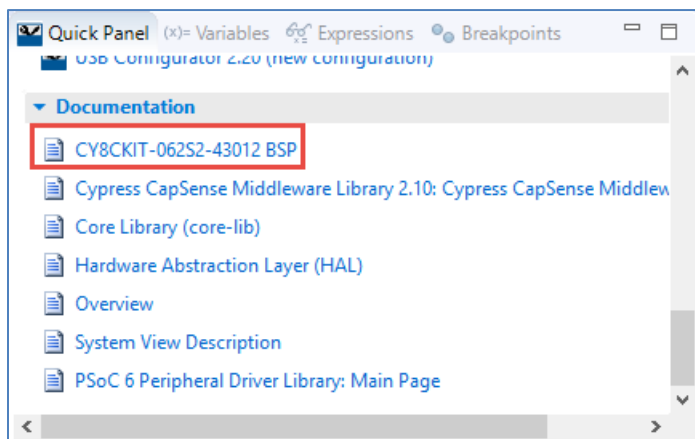
This directory contains the configuration files (such as *design.modus*) for use with various BSP configurator tools, including the Device configurator, QSPI configurator, and CAPSENSE™ configurator. At the start of a build, the build system invokes these tools to generate the source files in the *GeneratedSource* directory. A typical *COMPONENT_BSP_DESIGN_MODUS* directory looks like this:



Note: 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 an Application](#)) or create a custom BSP (see [Creating your own BSP](#)).

2.9.3 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. For example:



2.9.4 Modifying the BSP Configuration (e.g. design.modus) for an Application

If you want to modify the BSP configuration for an application (such as different pin or peripheral settings), you can modify the BSP, but if you are using a standard BSP that means you are dirtying the Git repo. If the BSP happens to be shared among multiple applications, any changes will affect all of the applications. If you don't want to do that, you can use the following process to create a custom set of configuration files for a specific application. This process is used for code examples that need to modify the default BSP configuration so you will see is show up in some example applications.

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

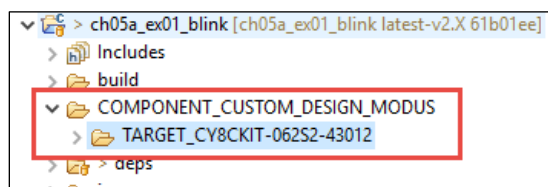
This is a recommended best practice. In an upcoming step, you will modify the *Makefile* to include files from that directory instead of the directory containing the default configuration files in the BSP.

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:

```
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.

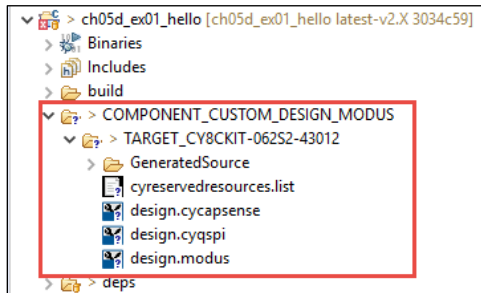
The first line disables the configuration files from the original BSP since they are now in different directory.

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.

2.9.5 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. Optionally you may specify a new device (`DEVICE_GEN`) if it is different from the BSP that you started from. For example:

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

This command creates a new BSP with the provided name at the top 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 (see [Manual library](#)).

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. Import dependencies using a make target. Note that the path to the BSP including `TARGET_` must be included in the command. For example, if you have a custom BSP called `MyBSP` in the application's root directory:

```
make import_deps IMPORT_PATH=TARGET_MyBSP
```

The command above finds the `.mtbx` files from the provided BSP and uses them to create direct dependencies in the application's `deps` directory.

4. Get all of the newly imported libraries:

```
make getlibs
```
5. Update the application's `Makefile` `TARGET` variable to point to your new BSP.

Note: If you are using the Eclipse IDE for ModusToolbox™, click the **Refresh Quick Panel** link. This is necessary so that when you open the Device Configurator it will open the file from the custom BSP. This only needs to be done once after changing the `Makefile`.

6. 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`. If you are using the Eclipse IDE for ModusToolbox™, this step can be done by clicking the **Generate Launch Configs for <app_name>** link in the Quick Panel.
7. 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.
8. 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 [Manual library](#).

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.

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: 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.

2.9.6 Updating the MCU device in a BSP

Based on project requirements, you may need to change the MCU device on an existing custom BSP. That can be done using `make update_bsp`. You must specify the BSP to update and the new MCU device to use. For example:

```
make update_bsp TARGET_GEN=MyBSP DEVICE_GEN=Cy8C624ABZI-S2D44
```

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` for VSCode or click the **Generate Launches...** link in the Quick Panel for Eclipse. This step must be done whenever the target device is changed since the launch configurations may change from device to device.

2.9.7 Updating the connectivity device in a BSP

If you want to change the connectivity (Wi-Fi/Bluetooth®) companion device in a BSP, you must make the changes manually. The required changes are:

1. Open the *design.modus* file in a text editor, find the line that contains `Device mpn=<connectivity device>` and update to the new connectivity device's name. This is usually near the end of the file.
2. Open the *<bsp>.mk* file in a text editor and:
 - a. Find the line that contains `ADDITIONAL_DEVICES:=<connectivity device>` and update to the new connectivity device's name.
 - b. Find the `COMPONENTS` line that specifies the connectivity device family and update as necessary. For example, you may need to change `COMPONENTS+=43013` to `COMPONENTS+=4343W`.
 - c. You may need to make changes to other `COMPONENTS` settings to get the proper Wi-Fi and Bluetooth® firmware from the `wifi-host-driver` and `bluetooth-freertos` libraries, respectively. The exact changes will depend on the construction of the BSP.

2.10 Manifests

Manifests are XML files that tell the Project Creator and Library Manager how to discover the list of available boards, example projects, libraries and library dependencies. 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 tool as well as the list of BSP packages that are presented in the Library Manager tool. There is also a separate BSP dependencies manifest that lists the dependent libraries associated with each BSP.
- 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. There is also a separate middleware dependencies manifest that lists the dependent libraries associated with each middleware library.

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 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 our provided files on GitHub. The following is a subset of the full set of manifest repos that exist. Each repo may also contain multiple manifests.

- Super Manifest: <https://github.com/infineon/mtb-super-manifest>
- Code Example Manifest: <https://github.com/infineon/mtb-ce-manifest>
- BSP Manifest: <https://github.com/infineon/mtb-bsp-manifest>
- Middleware Manifest: <https://github.com/infineon/mtb-mw-manifest>
- Wi-Fi Middleware Manifest: <https://github.com/infineon/mtb-wifi-mw-manifest>

2.11 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.

2.11.1 Network proxy settings

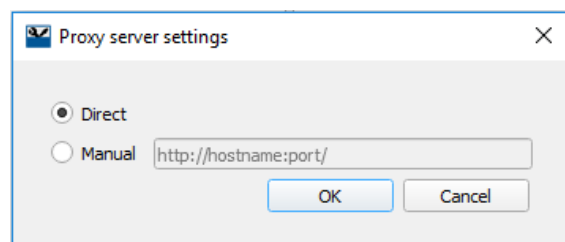
If you have environment variables for `http_proxy` and `https_proxy` (for example, to get Mbed to work), those same variables will work for ModusToolbox™ tools. However, there is a simpler way to enable/disable the proxy settings in the Project Creator and Library Manager tools that doesn't require you to create/delete the environment variables whenever you want to switch between a network that requires a proxy and a second network that does not require a proxy.

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/`.



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.

Note that "Direct" will also work if you have the *http_proxy* and *https_proxy* environment variables set so if you prefer to use the environment variables, just use "Direct" and create/delete the variables depending on whether your site needs the proxy or not.

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

2.11.2 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:

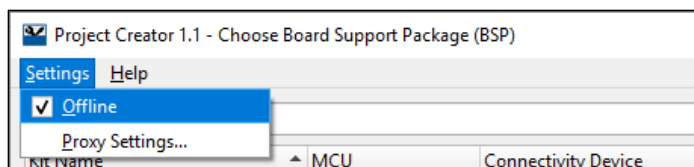
<https://community.cypress.com/t5/Knowledge-Base-Articles/Offloading-the-Manifest-Files-of-ModusToolbox-KBA230953/ta-p/252973>

2.11.3 Offline content

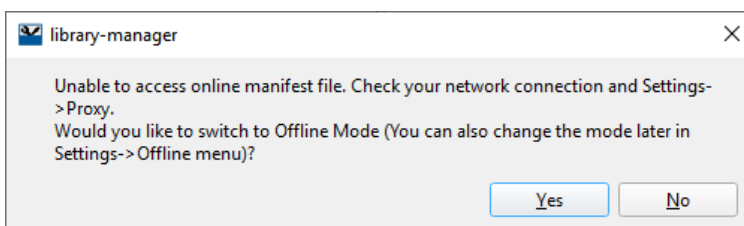
In the case where network access is just not possible, Infineon provides a zipped-up bundle with all of our GitHub repos to allow you to work offline, such as on an airplane or if for some reason you don't have access to GitHub. To set this up, you must have access to cypress.com in order to download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



2.12 Compiler optimization

The default compiler setting is "Debug". If you want to change this to "Release" to increase the optimization that is done, you must do 2 things:

1. Change the value of the variable `CONFIG` in the *Makefile* to "Release".
2. Regenerate or update any IDE launch configurations. This is necessary because the build output files go in a directory that has the `CONFIG` setting in the path. Therefore, if you don't update the configurations, the program and debug step will use the old build output files.

Note that the definition of Debug and Release are compiler dependent. For GCC_ARM, you can find the settings in *recipe-make-cat1a/<version>/make/toolchains/GCC_ARM.mk*:

```
ifeq ($(CONFIG), Debug)
CY_TOOLCHAIN_DEBUG_FLAG=-DDEBUG
CY_TOOLCHAIN_OPTIMIZATION=-Og
else ifeq ($(CONFIG), Release)
CY_TOOLCHAIN_DEBUG_FLAG=-DNDEBUG
CY_TOOLCHAIN_OPTIMIZATION=-Os
else
CY_TOOLCHAIN_DEBUG_FLAG=
CY_TOOLCHAIN_OPTIMIZATION=
endif
```

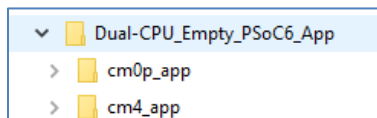
You can also set the value of `CONFIG` to a custom name (e.g. "Custom") and then use the `CFLAGS` variable in the *Makefile* to set the optimization parameters as you wish.

2.13 Multi-CPU applications

Some Infineon devices contain multiple CPU cores. For example, many PSoC™ 6 devices contain 2 CPUs - a Cortex M4 and a Cortex M0+. Most PSoC™ 6 MCU code examples just use the M4, but the M0+ can also be used when necessary to offload the M4 or to achieve lower power.

Note: Since both CPUs use many of the same resources (AHB, Flash, Ram, etc.), it is critically important to consider how an application running on one CPU may affect the other.

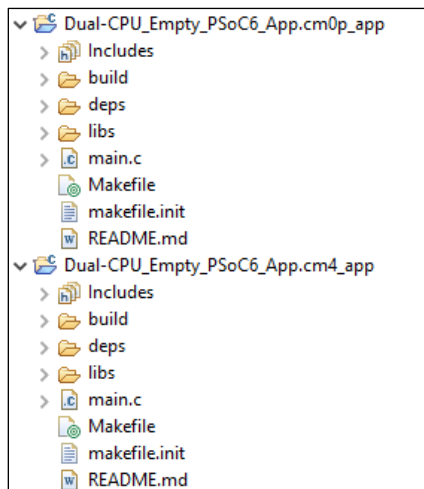
Dual-CPU designs are accomplished using two separate projects that are bundled into a single application. When you create an application based on a dual-CPU design, on disk you will have a top-level application directory that contains the projects for each individual CPU. For example, the Dual-CPU Empty PSoC™ 6 application will look like this:



Each application specifies its `CY_GETLIBS_SHARED_PATH` as `../..` (two levels up) instead of `../` (one level up). This accounts for the extra level of directory hierarchy so that the shared location is the same as for any other application in the same workspace.

Note: The above is true for Dual-CPU applications that use the MTB flow. Dual-CPU applications using the older LIB flow include shared libraries using a different method.)

Inside the Eclipse IDE, each project will appear separately, but the hierarchy will be reflected in the project names:



The *Makefiles* are set up so that if you build the CM4 project, it will build the CM0+ project as a dependency and will include it in the resulting build artifacts. Therefore, building/programming the CM4 project is all that is necessary to build and program both projects into the device.

See [AN215656](#) (PSoC™ 6 Dual-CPU System Design) for much more information on Dual-CPU designs including inter-processor communication (IPC), interrupts and debugging.

Several Dual-CPU code examples are available in the usual places (In the Project Creator starter application list or on the Cypress GitHub site) to demonstrate using IPC for sharing data, semaphores for synchronization, etc.

2.14 Exercises

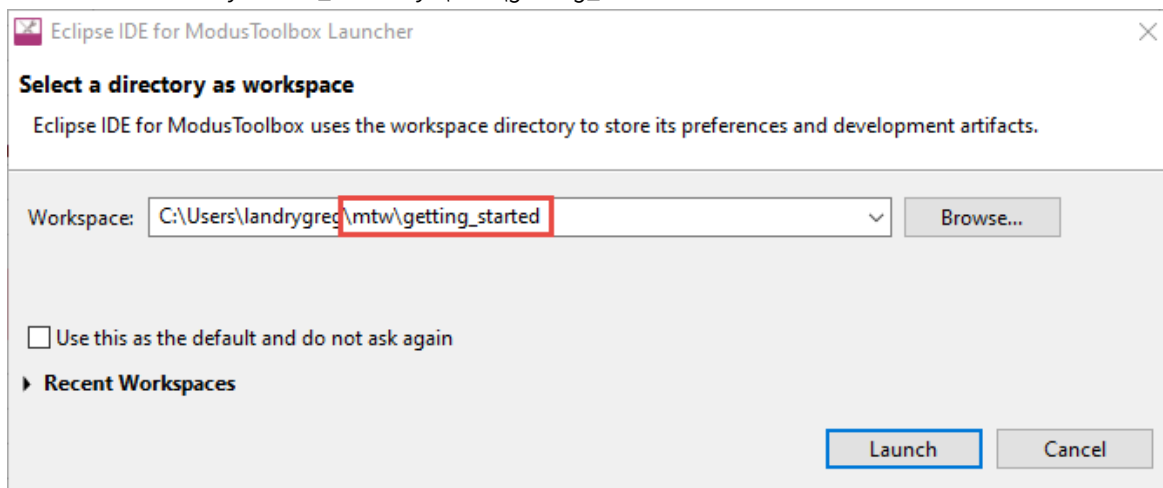
In the following exercises, the CY8CKIT-062S2-43012 kit is used but feel free to use any available BSP to explore the family or solution of your choice. The exact steps and starting application that you use may be different than what's shown here, but the concepts will be the same.

Exercise 1: Create, build, program and debug a Hello World application in Eclipse

For the first exercise, we will use the Eclipse IDE to create, build, program, and debug a Hello World application.

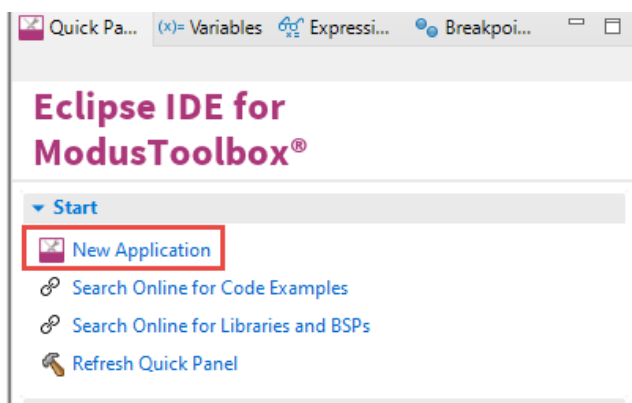
1. Start the Eclipse IDE for ModusToolbox™.

When it asks for a workspace directory, you can use the default of `<home_directory>\mtw`, you can add another level of hierarchy under `<home_directory>\mtw`, or you can use any other path that you prefer. I will use the directory `<home_directory>\mtw\getting_started`.



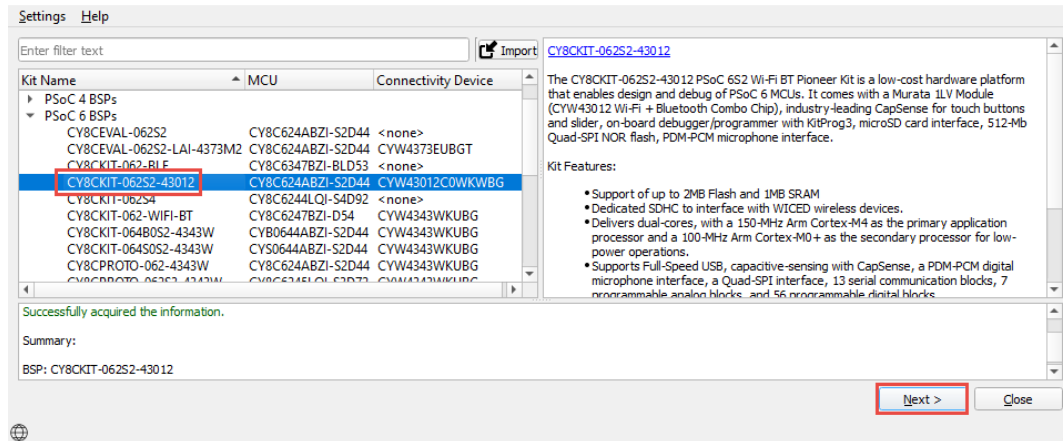
You can change workspaces at any time and you can have as many workspaces as you want.

2. On the Quick Panel, click the **New Application** link or select either **File > New > ModusToolbox Application**. Either option launches the Project Creator tool.

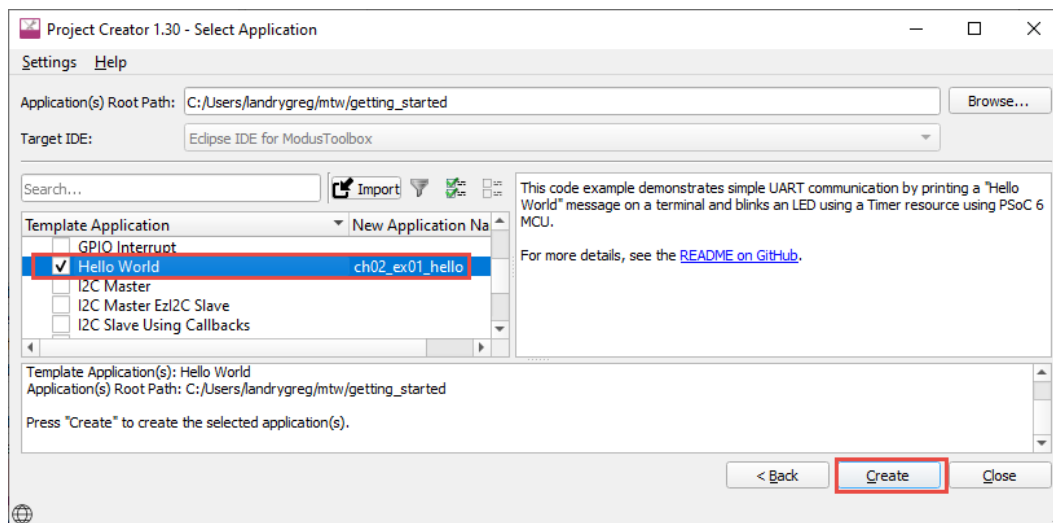




- Choose the correct development kit and click **Next >**. I will use the CY8CKIT-062S2-43012 but you can use a different kit if you prefer.



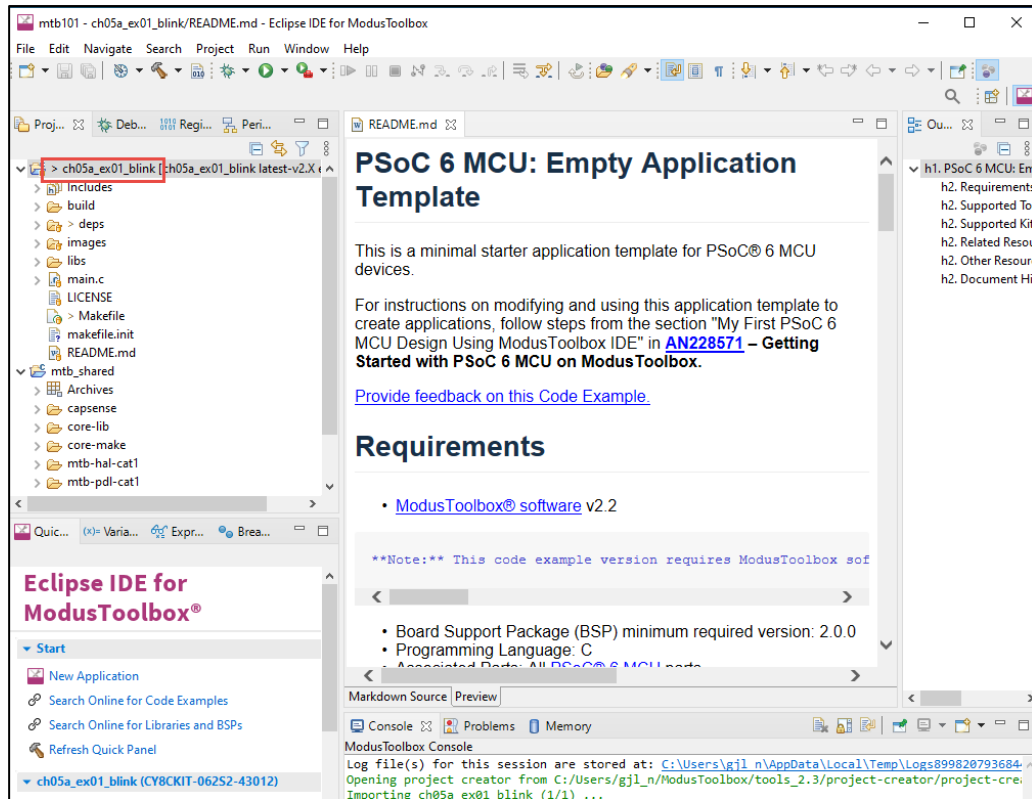
- Check the box next to *Hello World* and give your project a name (in this case I'll use **ch02_ex01_hello**). Then, click **Create**.



You can check the box for more than one application at a time if you want to create multiple applications in a single operation.

There are application searching and filtering capabilities which we will talk about in the [Project Creator](#) section.

Once the application has been created, the new project creation window will close (unless there are errors) and the new application is imported into the Eclipse IDE.



5. The *README.md* file will be open in the main Eclipse window. Read it to see what the example does.

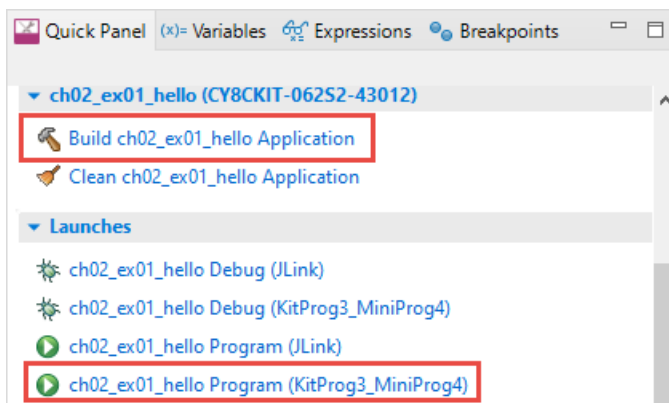
*Note: Eclipse does not render tables properly in Markdown files and may number lists differently. If you have a system default Markdown viewer/editor you can use that instead. Just right-click on *README.md* in the Project Explorer window, and select **Open With > System Editor**.*



6. Double click on *main.c* in the Project Explorer to see what the code looks like.



7. Click the build link in the quick panel to build the project or click the appropriate program link to build and then program.



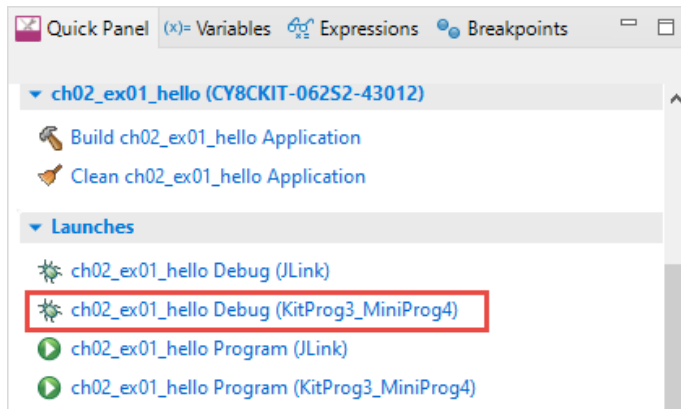
The kit I'm using has a KitProg3 on it so that's the one I will choose.

You should now see a blinking LED. You can also open a serial terminal window to see messages. Refer to the *README.md* file for the code example for instructions.

Note: If programming fails, the KitProg 3 might be in the wrong mode. Press and release the MODE SELECT button on the kit until the KitProg3 Status LED remains on.



- Click the appropriate debug link in the quick panel to build the project, program, and then launch the debugger.



- Start/pause execution, single step through the code, and add a breakpoint. See the Eclipse IDE for ModusToolbox™ User Guide for additional information on debugging.

Exercise 2: Create, build, program and debug a Hello World application in VSCode

For the next exercise, we will do the same thing as the prior exercise but using VS Code instead of Eclipse.

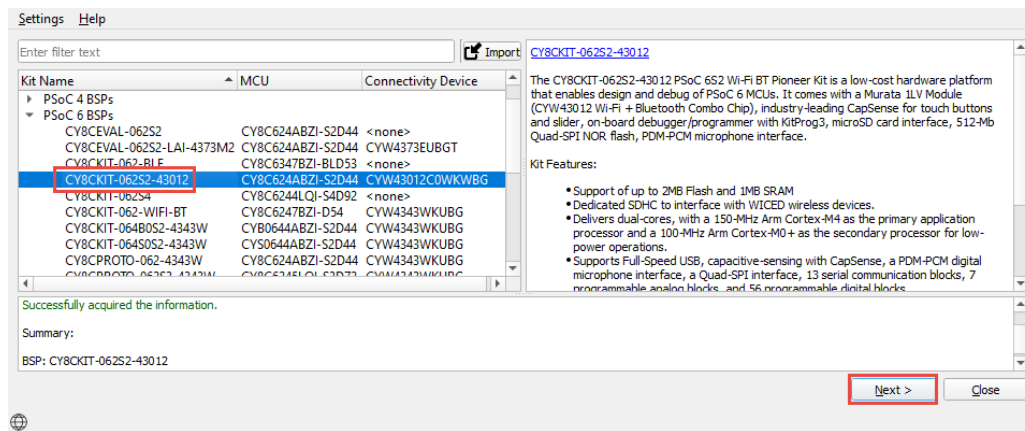


1. Run the Project Creator tool.

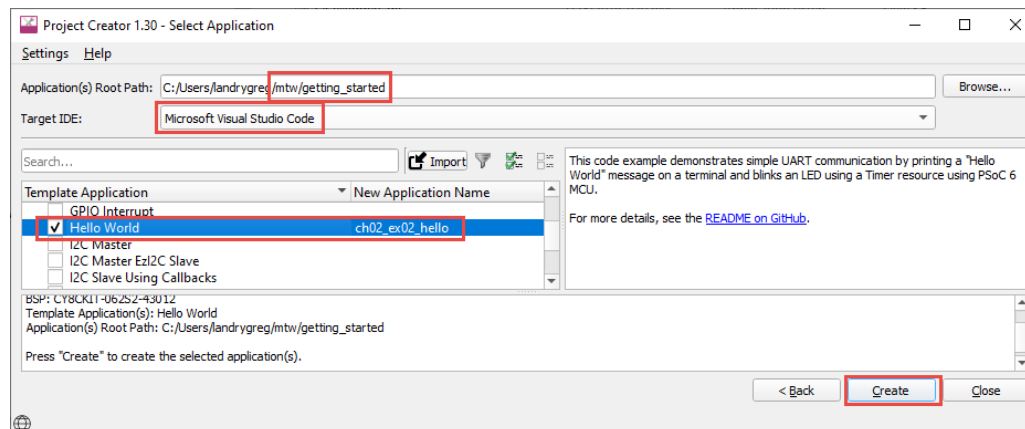
In Windows, you can open the search box and enter `project-creator`. On other operating systems, you can use a terminal to go to the directory `<install_path>/ModusToolbox/tools_<version>/project_creator` and run `project_creator.exe`.



2. Once Project Creator launches, select a kit and click Next just like in the previous exercise.



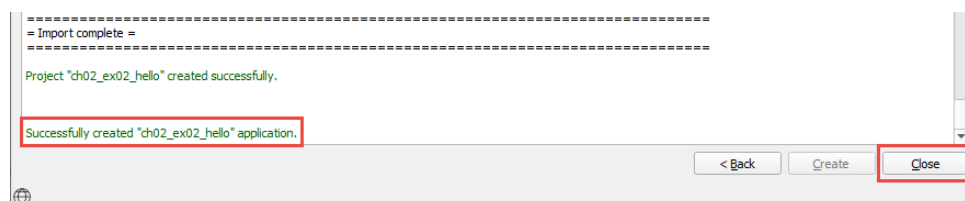
3. On the Application page, select the *Hello World* application and change the name to **ch02_ex02_hello**. Change the **Target IDE** to Microsoft Visual Studio Code. I'll leave the Application root path set as it is so that it will go in the same directory as the previous exercise but you can change that if you wish.



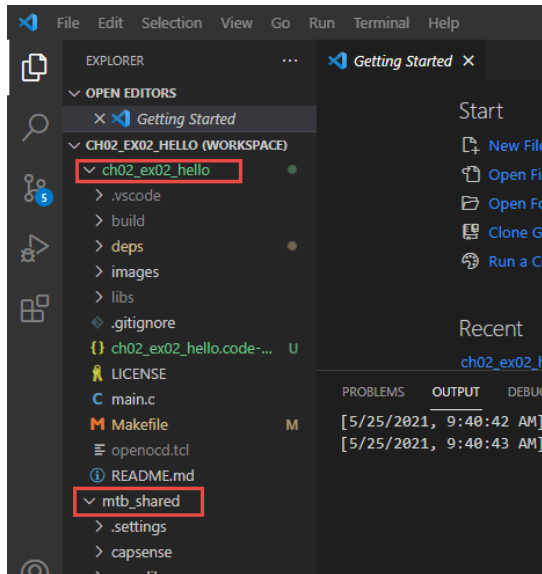
Note: If you forget to select the **Target IDE** when you create the application, remember that you can run `make vscode` from the command line once application creation is done.



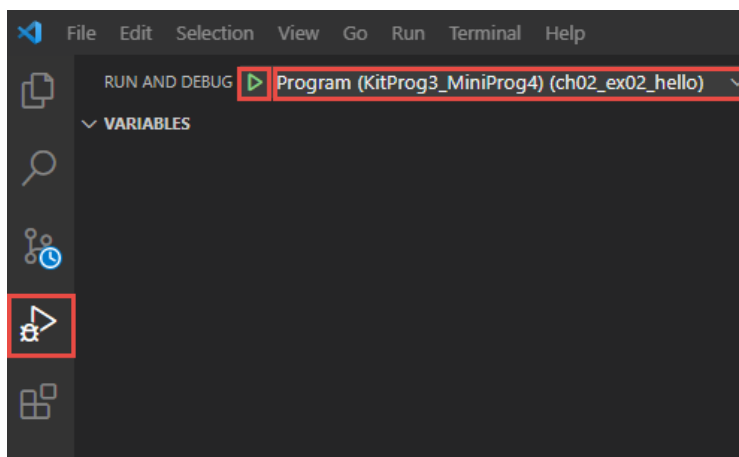
4. Once the application has been created, click the **Close** button.



- ☐ 5. Open the application in VS Code and open the workspace. If you open it correctly, you will see the application directory and the `mtb_shared` directory. See [Export for Microsoft Visual Studio Code \(VS Code\)](#) if you need a reminder.



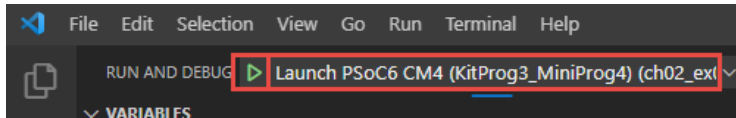
- ☐ 10. Right-click on the `README.md` file and select **Open Preview** to see the formatted Markdown file.
- ☐ 11. Open `main.c` and change `LED_BLINK_TIMER_PERIOD` to a smaller value (e.g. 999) so that the LED will blink faster this time.
- ☐ 12. Click on the Debug icon on the left side, select the appropriate Program item from the drop-down, and then click the green triangle to build and then program the board. You will see build and program messages in the terminal window at the bottom of the screen.



You should now see the faster blinking LED. You can also open a serial terminal window to see messages. Refer to the `README.md` file for the code example for instructions.

Note: If programming fails, the KitProg 3 might be in the wrong mode. Press and release the MODE SELECT button on the kit until the KitProg3 Status LED remains on.

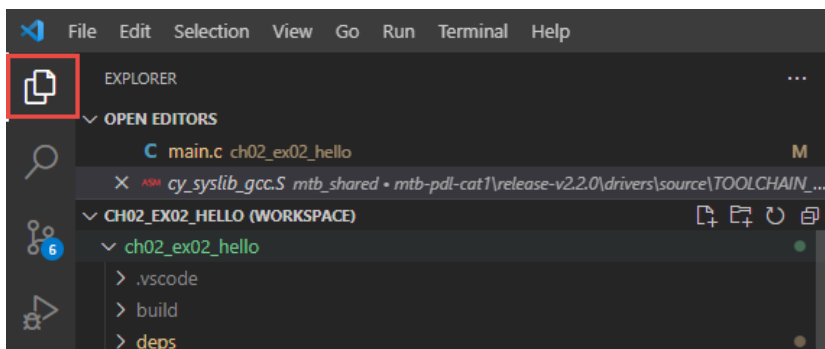
- ☐ 13. To debug, select the appropriate Launch item from the drop-down, and then click the green triangle to launch the operation.



- ☐ 14. Start/pause execution, single step through the code, and add a breakpoint. See the VS Code user documentation for additional information on debugging. Click the stop button when done.



- ☐ 15. To go back to the project explorer window view, click the icon on the left side.



Exercise 3: Build and program using the Command Line Interface

For this exercise, you will build and program an existing application using the CLI on one of your existing applications.

- ☐ 1. Start modus-shell (Windows) or open a command terminal (other platforms).
- ☐ 2. Change to the directory for the application from exercise 1.
- ☐ 3. Enter the command `make help` to see a list of possible commands.
- ☐ 4. Enter the command `make program` to build the application and program the board based on the settings in the application's *Makefile*.

Exercise 4: Use the Library Manager

In this exercise, you will use the Library Manager to add a new library to the Hello World application. I'll use the RGB LED library but if you are using a device other than a PSoC™ 6 MCU, that library may not be available. If so, review the other available libraries and try one that will allow you to add new functionality the application.

- ☐ 1. Create a new application from the Hello World application and call it **ch02_ex04_rgb**.

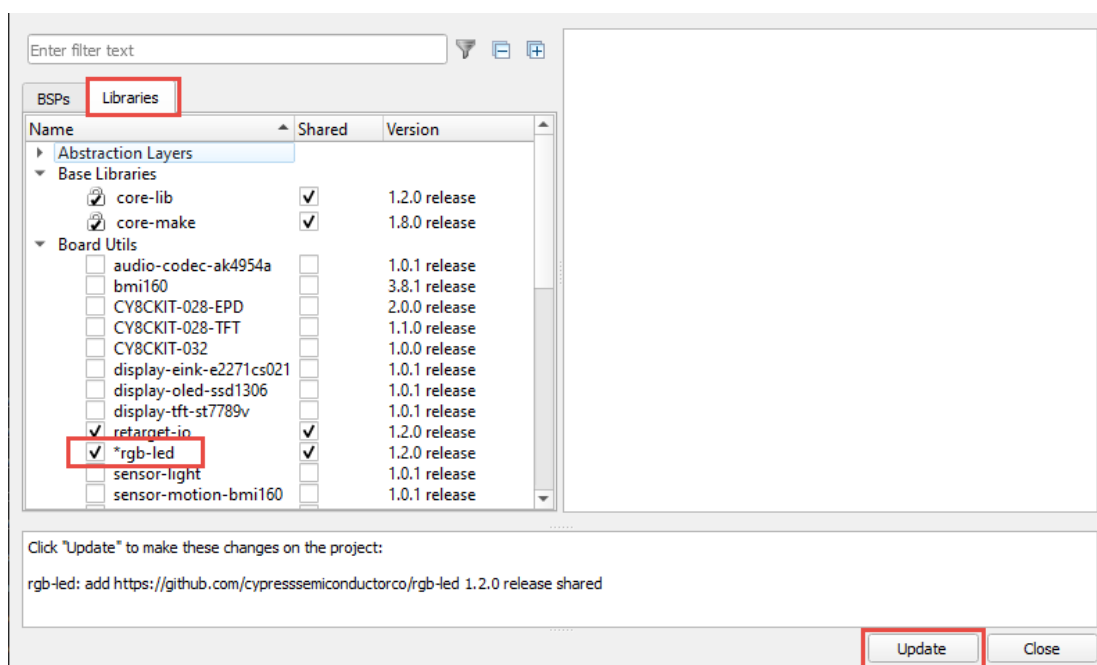
- ☐ 2. Run the Library Manager from within the application:

Eclipse: Click the link in the Quick Panel

VS Code: Use the menu item **Terminal > Run Task... > Tool: Library Manager**

CLI: Enter the command `make modlibs`

- ☐ 3. Once the Library Manager window fully loads, select the **Libraries** tab. Check the box for the *rgb-led* library and click **Update**.



- ☐ 4. Once the update is done, click **Close** and then quit the Library Manager.

- ☐ 5. Look at the documentation for the RGB LED library and add code to *main.c* to cycle through various colors when the timer expires (e.g. red, green, blue).

- ☐ 6. Build and program. Observe the RGB LED's behavior.

Exercise 5: Create a Custom BSP and Use the Device Configurator

In this exercise, you will use the Device Configurator. I will demonstrate changing a pin name to swap LED pin assignments but you can make any change you want.

- ☐ 1. Create a new application from the Hello World application and call it **ch02_ex05_custom**.

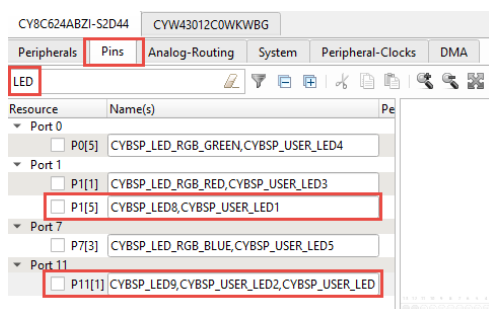
We are going to change the device configuration in the BSP so let's create a custom BSP first so that we are not editing the starting BSP git repo.
- ☐ 2. From the command line, go to the application's directory and run the command `make bsp TARGET_GEN=MyBSP`.
- ☐ 3. Run the command `make import_deps IMPORT_PATH=TARGET_MyBSP`.
- ☐ 4. Run the command `make getlibs` to download the libraries from GitHub.
- ☐ 5. Edit the *Makefile* and change the `TARGET` variable to `MyBSP`.
- ☐ 6. If you are using VS Code, run the command `make vscode`.
- ☐ 7. If you are using Eclipse, click the link **Generate Launches for ch02_ex05_custom** from the Quick Panel.

The steps above are necessary so that program and debug will work properly with the new BSP.
- ☐ 8. If you are using Eclipse, click the link **Refresh Quick Panel**.

This step is necessary so that the Device Configurator will open the file from the custom BSP.
- ☐ 9. Run the Device Configurator from within the application:

Eclipse: Click the link in the Quick Panel
VSCode: If there is no Run Task menu entry for the Device Configurator, use the CLI method instead.
CLI: Enter the command `make config`

- ☐ 10. Once the Device Configurator opens, select the **Pins** tab and enter LED in the filter text box. Move the definition for `CYBSP_USER_LED` from `P1[5]` to `P11[1]`.



Note: Don't forget the comma and do NOT put in any spaces since they will be converted to underscores.
The code in *main.c* uses the name `CYBSP_USER_LED` so this change will cause a different LED to blink.

- ☐ 11. Save the change and exit the Device Configurator.
- ☐ 12. Build and program. Observe the blinking LED is now the other user LED (red instead of orange).

Trademarks

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

Published by
Infineon Technologies AG
81726 Munich, Germany

© 2021 Infineon Technologies AG.
All Rights Reserved.

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffheitsgarantie").

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

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

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

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS

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

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