

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	Software, application and BSP versions	3
2.2	Tools and the make system	3
2.2.1	Make.....	4
2.2.2	Using the command line interface (CLI)	5
2.3	Application creation	8
2.3.1	Project Creator tool GUI.....	8
2.3.2	Project Creator tool CLI.....	11
2.3.3	Using Git (manual creation)	12
2.3.4	Template processing.....	13
2.4	Board Support Packages (BSPs)	14
2.4.1	BSP directory structure.....	14
2.4.2	BSP dependency version locking	16
2.4.3	BSP names.....	16
2.4.4	BSP documentation	16
2.5	Application directory organization	17
2.5.1	Single-core application.....	17
2.5.2	Multi-core applications.....	21
2.6	Eclipse IDE for ModusToolbox™ tour	24
2.6.1	Quick Panel.....	25
2.6.2	Help menu	27
2.6.3	Integrated debugger	27
2.6.4	Console	27
2.6.5	Terminal	28
2.6.6	Documentation	28
2.6.7	Eclipse IDE tips & tricks.....	28
2.7	Library management	30
2.7.1	Library classification	30
2.7.2	Including libraries and dependencies	31
2.7.3	Library Manager tool.....	33
2.7.4	Re-Downloading Libraries	37
2.8	Configurators	38
2.8.1	BSP configurators	38
2.8.2	Library configurators	44
2.9	Working with custom BSPs	46
2.9.1	BSP Assistant.....	46
2.9.2	Using a custom BSP in a new application	53
2.11	Importing/exporting applications for IDEs	54
2.11.1	Import projects into the Eclipse IDE.....	54
2.11.2	Export for Microsoft Visual Studio Code (VS Code)	56

2.11.3	Export for IAR Embedded Workbench	62
2.11.4	Export for Keil µVision	64
2.12	Version control and sharing applications	66
2.12.1	Files to include/exclude	66
2.12.2	Using version control software	66
2.12.3	Manual file copy	67
2.12.4	Using Eclipse archive files	67
2.13	Manifests.....	68
2.14	Network considerations	70
2.14.1	Network proxy settings	70
2.14.2	Offloading manifest files	71
2.14.3	Offline content	71
2.15	Compiler optimization	71
2.16	Serial Terminal Emulator	73
2.16.1	Determine the Port.....	73
2.16.2	PuTTY (Windows and Linux)	73
2.16.3	Screen (MacOS)	75
2.16.4	Eclipse IDE for ModusToolbox™ built-in terminal.....	75
2.17	Version 2.x BSPs/applications versus 3.x BSPs/applications.....	76
2.18	Exercises	77
	Exercise 1: Create, build, program and debug a Hello World application in Eclipse	77
	Exercise 2: Create, build, program and debug a Hello World application in VS Code	81
	Exercise 3: Build and program using the Command Line Interface.....	84
	Exercise 4: Use the Device Configurator to change a BSP setting	85
	Exercise 5: Use the BSP Assistant	86

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 Software, application and BSP versions

This training focuses on ModusToolbox™ 3.x tools. Applications and BSPs created with the ModusToolbox™ 2.x ecosystem use a slightly different format. They will fully function using the 3.x tools but you will not be able to take advantage of all of the 3.x features. For additional details about 2.x applications and BSPs, see section [2.17](#).






























2.2 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:

-  bsp-assistant
-  bt-configurator
-  capsense-configurator
-  cymcuelftool-1.0
-  device-configurator
-  dfuh-tool
-  driver_media
-  ez-pd-configurator
-  fw-loader
-  gcc
-  jre
-  kp-firmware
-  library-manager
-  lin-configurator
-  make
-  modus-shell
-  mtbgetlibs
-  mtbideexport
-  mtbquery
-  mtbsearch
-  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 or from a command line interface.

2.2.1 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 the Eclipse IDE for ModusToolbox™ or the Visual Studio Code 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 Eclipse or VS Code, 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.

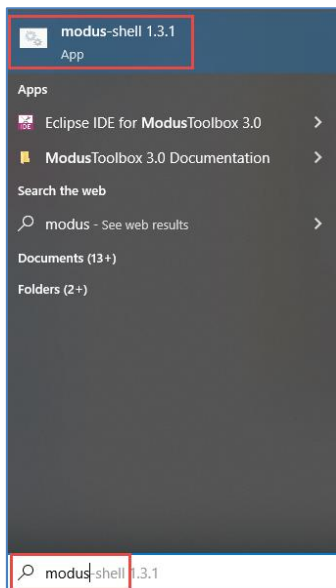
The IAR Embedded Workbench and Keil µVision IDEs have their own internal build system. So, if you are using one of those IDEs, you probably will want to stick with the IDE's build system. However, other make-based commands that you will see later such as `make library-manager` and `make device-configurator` can still be run from the command line on those applications.

2.2.2 Using the command line interface (CLI)

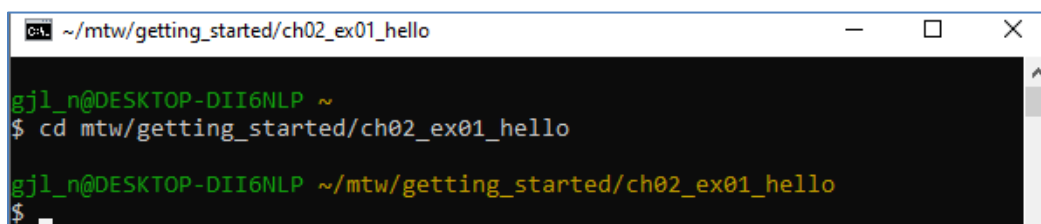
Instead of (or in addition to) using an IDE, each tool in ModusToolbox™ provides a command line interface that interacts directly with the make system.

2.2.2.1 Windows

To use the CLI, you need a shell that has the correct tools (such as git, make, etc.). The ModusToolbox™ ecosystem includes "modus-shell" which is based on Cygwin and ensures all settings and variables are correct for working with ModusToolbox™ tools. It is in the installation directory under *ModusToolbox/tools_<version>/modus-shell*. It is listed in the **Start** menu under **ModusToolbox <version>** or you can just start typing `modus-shell` in the Windows search box until it appears in the list.



Once you start the shell, it will look something like the following. You will need to change to the directory containing the ModusToolbox™ application that you want to operate on before running any command line operations.



2.2.2.2 macOS / Linux

To run the command line on macOS or Linux, just open a terminal window and change to the application's root directory.

2.2.2.3 Terminal Window Tab

If you are inside the Eclipse IDE for ModusToolbox™, there is a **Terminal** tab adjacent to the **Console** tab. The **Terminal** tab contains a command line interface window that is already changed to the application's root directory. On the Windows operating system, this will be a modus-shell while on MacOS or Linux, it will be a standard command terminal.

2.2.2.4 Make targets (commands)

In order to have the command line commands work, you need to be at the top level of a ModusToolbox™ application or project where the *Makefile* is located.

The following table lists a few helpful make targets (i.e. commands). Refer also to the ModusToolbox™ tools package user guide which can be found here: www.infineon.com/ModusToolboxUserGuide

Make command	Description
make help	This command will print out a list of all the make targets. To get help on a specific target type <code>make help CY_HELP=getlibs</code> (or whichever target you want help with).
make getlibs	Process all the <i>mtb</i> files (or <i>lib</i> files) and bring all the libraries into your project. If you receive a shared application from someone, this is typically the next step.
make build	Build your project.
make debug	Build your project, program it to the device, and then launch a GDB server for debugging.
make program	Build and program your project.
make qprogram	Program without building.
make device-cofigurator	This command will open the Device Configurator.
make get_app_info	Prints all variable settings for the app.
make get_env_info	Prints environment information such as the tool versions being used.
make printlibs	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 command or variable use `make help CY_HELP=<command or variable>`. For example:

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

```
$ make help CY_HELP=TARGET
Searching installed tools in progress...
Searching installed tools complete

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

Current target in this application is, [ ../bsps/TARGET_APP_CY8CPROTO-062-4343W/APP_CY8CPROTO-062-4343W.mk ].

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

When running a command that takes significant time - such as build, debug, and program – it is highly recommended to use a make option to allow multiple parallel jobs. The option is "-j". It can be added by itself, which allows make to determine the optimal number of jobs, or you can specify a number of parallel jobs to allow. For example:

```
make -j build
make -j8 build
```

Note: The default configuration in Eclipse is setup to run 8 jobs (-j8).

2.2.2.5 Accessing tools from the command line

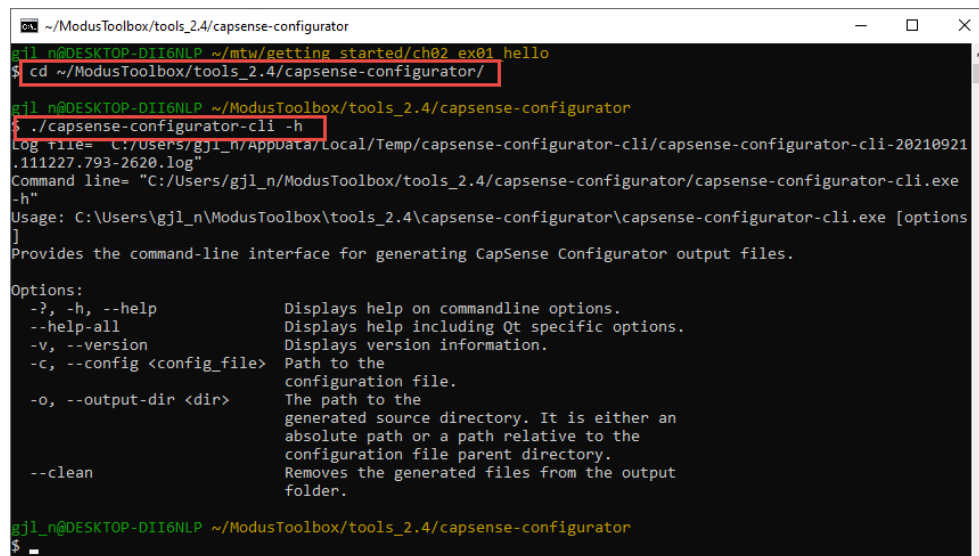
Most of the tools can be launched from an application's directory where the *Makefile* is located. Use `make` along with the executable name of the tool. A couple useful examples:

- `make device-configurator` launches the Device Configurator and opens the application's configuration file.
- `make library-manager` launches the Library Manager and opens the application's library settings.
- `make bsp-assistant` launches the BSP Assistant and opens the application's active BSP.

As described earlier, all tools can also be run directly from the tool's directory. 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
```



```
gjl_n@DESKTOP-DII6NLP ~/ModusToolbox/tools_2.4/capsense-configurator
$ cd ~/ModusToolbox/tools_2.4/capsense-configurator/

gjl_n@DESKTOP-DII6NLP ~/ModusToolbox/tools_2.4/capsense-configurator
$ ./capsense-configurator-cli -h
Log file= "C:/Users/gjl_n/AppData/Local/Temp/capsense-configurator-cli/capsense-configurator-cli-20210921
.111227.793-2620.log"
Command line= "C:/Users/gjl_n/ModusToolbox/tools_2.4/capsense-configurator/capsense-configurator-cli.exe
-h"
Usage: C:\Users\gjl_n\ModusToolbox\tools_2.4\capsense-configurator\capsense-configurator-cli.exe [options
]
Provides the command-line interface for generating CapSense Configurator output files.

Options:
-?, -h, --help           Displays help on commandline options.
--help-all              Displays help including Qt specific options.
-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.
--clean                  Removes the generated files from the output
                        folder.

gjl_n@DESKTOP-DII6NLP ~/ModusToolbox/tools_2.4/capsense-configurator
$
```

2.3 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.

Note: This section describes creating a new application from a template. The process to import an existing application or to share an existing application with another team member is different. That process is covered in section [2.12](#).

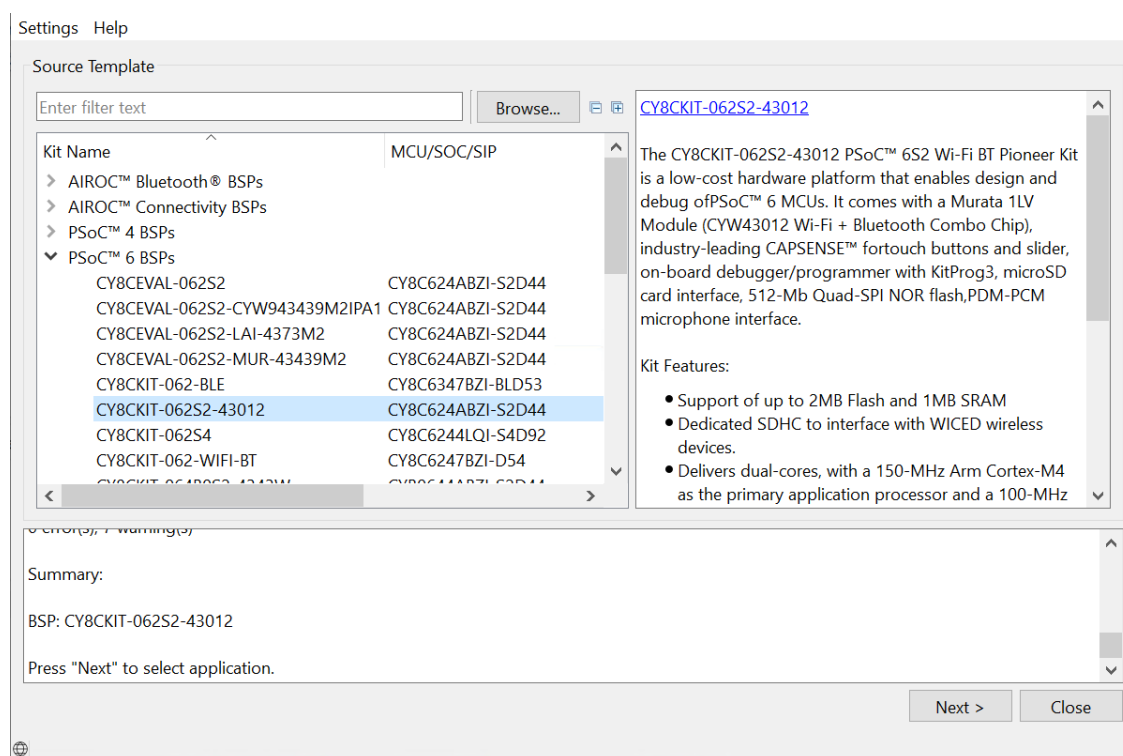
2.3.1 Project Creator tool GUI

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 section [2.13](#).

Note: You can also create projects in offline mode. See section [2.14.3](#) 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 below.



Note that there is a **Browse** button, so you can specify your own custom BSP if you have one.

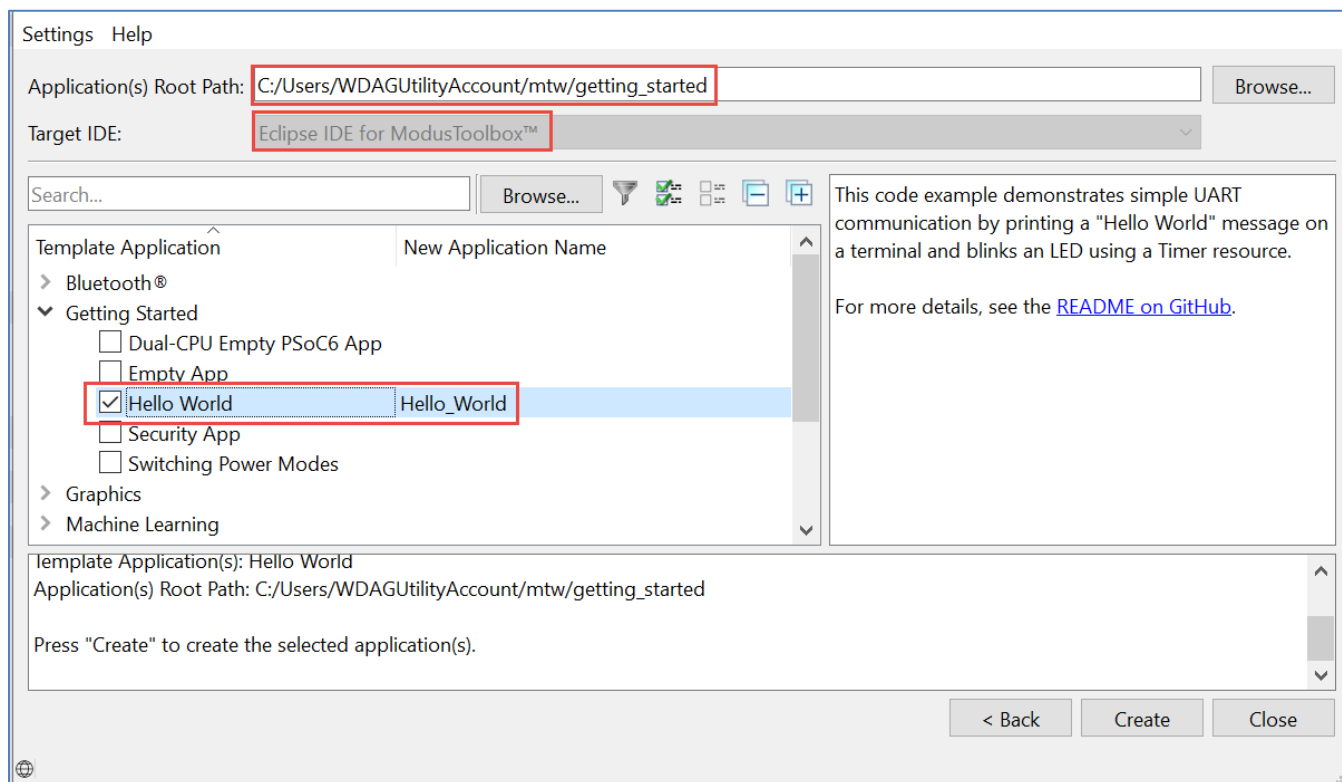
Select a kit and click **Next >**.

Now you will be presented with all the template code examples supported by the BSP you chose. Pick an application to start from and give it a name. The applications are arranged by categories to make it easier to find what you are looking for. Only applications that are valid for the BSP you chose in the previous step will appear.

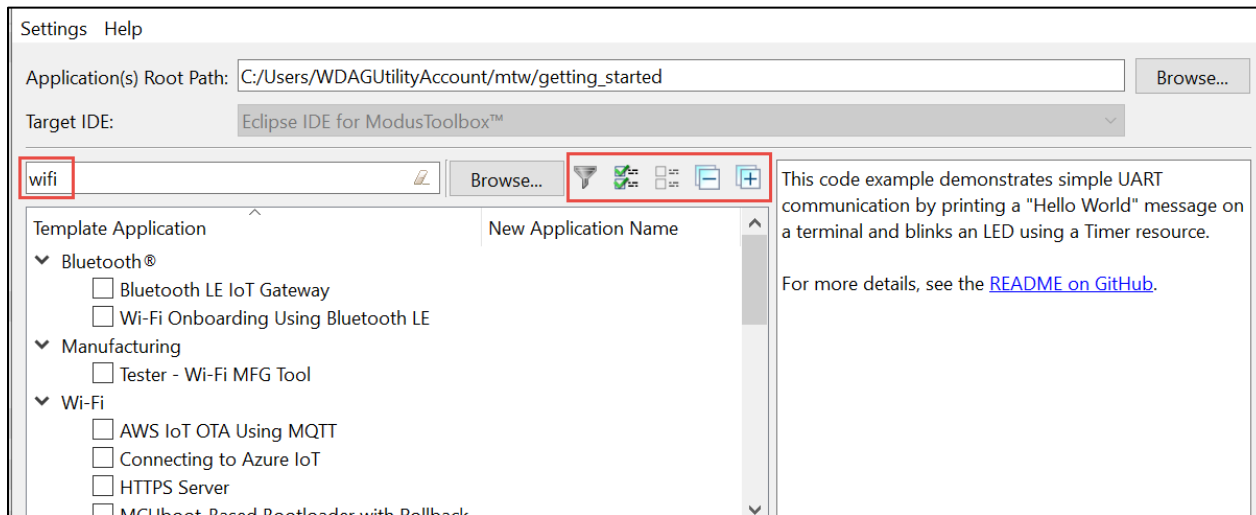
You can specify a different **Application Root Path** if you don't want to use the default value. A directory with the name of your application(s) will be created inside the specified Application Root Path. The value for the Application Root Path is remembered from the previous invocation, so by default 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.

Note: If you choose IAR Embedded Workbench, the project will be set up to use the IAR toolchain, but the application's Makefile will not be modified. Therefore, builds inside IAR Embedded Workbench will use the IAR toolchain while builds from the command line will continue to use the toolchain that was previously specified in the Makefile. You can edit the Makefile's TOOLCHAIN variable if you also want command line builds to use the IAR toolchain.



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). Only categories containing matching applications will be listed.

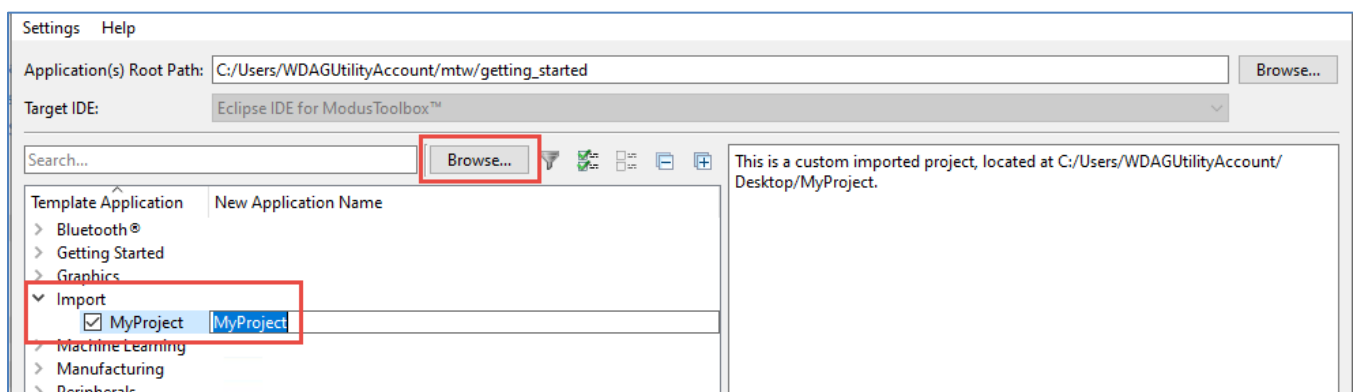


If desired, you can select all the applications in the search result by using the **Select All** button or you can unselect them with the **Unselect All** button. You can also expand or collapse all categories with the **Expand All** and **Collapse All** buttons.

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 template application instead of one of the code examples that we provide, click the **Browse** button. This allows you to browse to an application or application template on your computer and then create a new application based on it. Make sure the path you select is to the directory that contains the Makefile, otherwise application creation will fail.

Once you select an application directory, it will show up in the **Import** category.



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**. (If you run the Project Creator from the Eclipse IDE, it will close automatically).

2.3.2 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.

2.3.3 Using Git (manual creation)

Instead of using the Project Creator tool, you can use git commands directly to create a ModusToolbox application from a template.

Note: It is more complicated to create a new application manually, so it is typically better to use the Project Creator tool using the GUI or command line interface. This is NOT true if you have an existing application as a Git repo that you want to share. See section 2.12 for details on that use case.

You can get to the Infineon 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/Infineon/Code-Examples-for-ModusToolbox-Software>

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/Infineon/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
```

The next step is to determine if the BSP you want to use is the default for the application. You can do this by checking if the BSP you want to use is in the `deps` directory of the application repo that you cloned. If it is not, you need to add it to the application and set it as the active BSP. You can do this either by running the Library Manager tool, or you can run the Library Manager command line interface version. Using the CLI version requires two steps. For example:

```
library-manager-cli.exe --add-bsp-name TARGET_CY8CKIT-062S2-43012 --add-bsp-location local  
library-manager-cli.exe --set-active-bsp APP_CY8CKIT-062S2-43012
```

Then you can run the following to get all of the libraries required by the application:

```
make getlibs
```

At this point, the application is ready for use. You can do any operations you want such as build and program. You can also list all of the possible make operations with the help command:

```
make build  
make program  
make help
```

2.3.3.1 Differences when using Git directly for project creation

After using `git clone` followed by `make getlibs` to create a project, there will a few differences in the result compared to using the Project Creator tool. The most notable differences that you must be aware of are:

- When using the Project Creator tool, all dependencies – both direct and indirect - 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, indirect dependencies will be locked but 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 section [2.7](#) for more details.

- If the *mtb* file for the BSP indicates a shared location (`$$ASSET_REPO$$`) rather than a local location (`$$LOCAL$$`), the BSP will end up in the shared location (e.g. *mtb_shared*) as a repo rather than an in-app BSP in the *bsps* directory. This is important if the application changes any BSP settings such as device configuration using the templates mechanism (discussed next).

2.3.4 Template processing

Some code examples have a directory called *templates* that contains files to override the default BSP configuration. When an application is created from such a code example, the files from the *templates* directory that apply to the selected BSP will be copied into the BSP in the *bsps* directory. This allows code examples to change the default BSP configuration without having to supply a complete custom BSP. If the BSP is not located in the *bsps* directory, the *templates* directory is ignored.

The *templates* directory is excluded from the build process by including it in a *.cyignore* file or the `CY_IGNORE` variable in the *Makefile*.

2.4 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
- Device configurations such as clocks, pins, and on chip-peripherals like PWMs, and ADCs
- The peripherals on the board such as LED, buttons, and sensors
- What the device pins are connected to on the board
- What libraries the BSP depends on

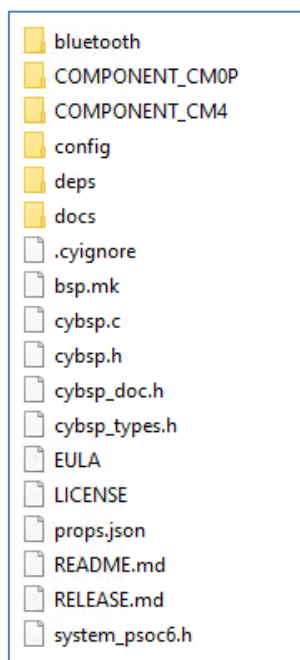
A BSP directory starts with the keyword "TARGET" and can be seen in the *bsps* directory of the application because the BSP is local to the application. In fact, when you create an application or add a new BSP to an application, the BSP that you select is copied into the application's *bsps* directory and its Git information (if any) is removed. At that point, the application "owns" the BSP for that application. This allows you to modify the BSP configuration for an application without dirtying a Git repo or affecting all other applications in the same workspace.

Note: ModusToolbox™ 3.0 format BSPs will either be in the bsp directory or will be shared in the mtb_shared directory. The location is determined by the manifest entry for that BSP and it cannot be changed once the application is created.

Note: Once a BSP is created inside an application's deps directory, its version cannot be changed. If you want a newer version of the BSP, use the Library Manager to add an additional BSP to the application using the desired version. Once the new BSP is added and changed to the active BSP, the previous one can be removed.

2.4.1 BSP directory structure

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



The various directories and files in a BSP include:

Directory/File	Description
<i>bluetooth/</i>	Contains information for a companion Bluetooth® device that is used in conjunction with the onboard MCU. This includes pin connections between the devices, and transport mechanism details.
<i>COMPONENT_CM4/</i> and <i>COMPONENT_CM0P/</i>	Startup code and linker scripts for all supported toolchains for each of the two cores - the CM4 and the CM0+.
<i>config/</i>	<p>Contains BSP configuration files. These are edited by using the appropriate configurator. For example, <i>design.modus</i> is edited using the device configurator.</p> <p>Contains the <i>cyreservedresources.list</i> file used to reserve resources so that the user cannot change settings or attempt to use them for other purposes in the device configurator. This is typically only used for devices that require specific UDB, routing, or clocking resources to implement an advanced feature.</p> <p>Contains the <i>GeneratedSource</i> that is created by the configurators.</p>
<i>deps/</i>	Contains <i>mtbx</i> files for libraries that the BSP requires as dependencies. These files are generated during application creation using the dependency information contained in a manifest file. Manifest files are covered in section 2.13 .
<i>docs/</i>	HTML based documentation for the BSP.
<i>.cyignore</i>	Used to exclude directories from the build if necessary for this BSP.
<i><bsp_name>.mk</i>	Defines the <i>DEVICE</i> and other BSP-specific make variables such as <i>COMPONENTS</i> .
<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>).
<i>cybsp_doc.h</i>	Used to generate documentation on the BSP's resources. It is not used for any functional purpose.
<i>cybsp_types.h</i>	Contains macros for using various board peripherals. For example, <i>CYBSP_LED_STATE_ON</i> and <i>CYBSP_LED_STATE_OFF</i> are set to the appropriate values based on the LED hardware on the board (e.g. active low or active high).
<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>props.json</i>	Contains BSP information including but not limited to capabilities that the BSP supports, version information, category for the Project Creator and Library Manager, and BSP capabilities.
<i>README.md</i>	BSP documentation in Markdown format.
<i>RELEASE.md</i>	BSP release notes.
<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.

2.4.2 BSP dependency version locking

The dependencies of a BSP are stored in *mtbx* files in the BSPs *deps* directory. When a BSP is created inside an application, the version of each dependency is locked to a specific version so that it does not change in that application unless the user explicitly requests a change. To change the version of a dependency inside a BSP, use the BSP assistant - which will be discussed in a minute. To change the version of a dependency inside an application without affecting the BSP itself, use the Library Manager.

2.4.3 BSP names

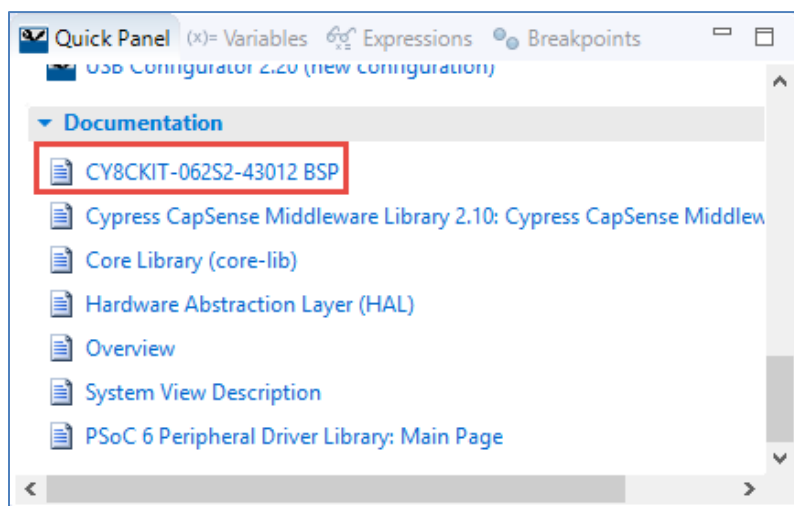
When creating an application from a manifest based BSP, the name of the in-app BSP created in the application will use the same name with a prefix of *APP* (e.g. *TARGET_APP_CY8CKIT-062S2-43012*). When creating an application using a local custom BSP using the **Browse...** mechanism, the name of the in-app BSP created in the application will be the same as the name of provided BSP.

If you want the BSP in your application to have a specific name that is different from the default, it is best to create it first using the BSP Assistant tool and then browse to it during application creation. You can also specify a name when you add a new manifest based BSP to an existing application using the Library Manager. Future versions of the tools may allow the BSP to be renamed during application creation.

After it is created, you can rename a BSP manually by just changing the name of the directory the BSP is contained in. It must always start with "TARGET_" but otherwise can have any name you want. Spaces and other special characters should not be used in a BSP name. Remember if you change the name of BSP in an application to update the *Makefile* and regenerate any launch targets to reflect the new name.

2.4.4 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:

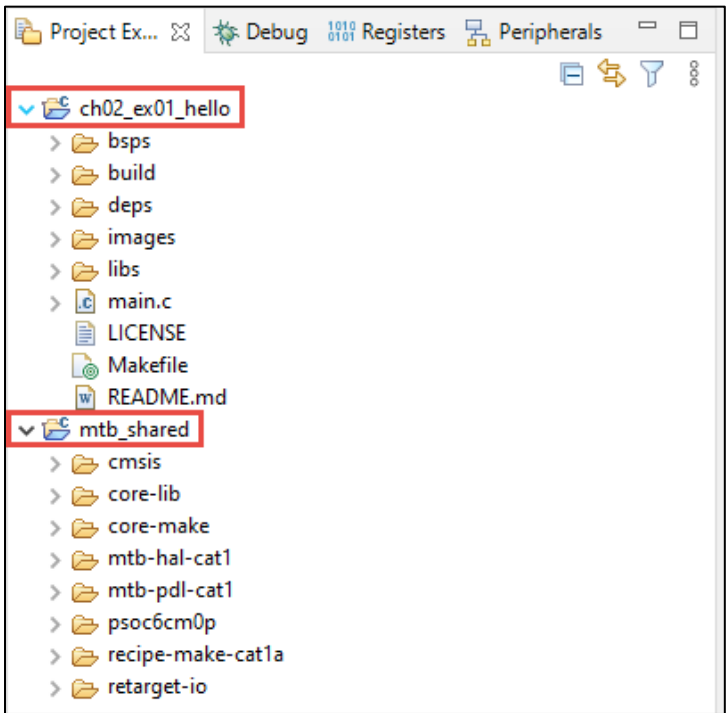


2.5 Application directory organization

A ModusToolbox™ application will contain one or more projects. In a simple application that uses only one CPU core, the application and the project will be the same directory. In a typical multi-core application, the top-level directory will be the application and it will have subdirectories containing the project for each core.

2.5.1 Single-core application

For applications that only use one CPU core, there is just one project. In that case, the top-level application directory and project directory are the same. There is usually another directory containing libraries that may be shared between applications. As an example, a typical PSoC™ 6 MCU single-core 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 directory may not appear until a build is done.
<i>bsps</i>	This directory contains the board support packages (i.e. BSPs) that are owned by the application. The BSPs in this directory are copies of the BSP selected during application creation and any added to the application using the Library Manager. They are no longer Git controlled libraries. Any edits done to these BSPs apply only to this application.
<i>build</i>	This directory contains build files, such as the <i>elf</i> and <i>hex</i> files.

Directory/File	Description
<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>
<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. 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.</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 version control system.</p>
<i>templates</i> (not shown)	Some code examples have a directory called <i>templates</i> that contains files to override the default BSP configuration. See section 2.3.4 for details.
<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 section 2.1 for details about editing the <i>Makefile</i> .
<i>README.md</i>	Almost every code example includes a <i>README.md</i> file (in Markdown format) that provides a high-level description of the application.

2.5.1.1 Makefile

Various build settings can be set in the *Makefile* or can be specified on the command line 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:

Application Type

For single core applications, the variable `MTB_TYPE` should be set to `COMBINED`. This indicates that the application and project are both located in the same directory since there is only one project that makes up the application.

Target

This variable sets the BSP that will be used for the build. A directory named `TARGET_<target variable value>` must be in the application's search path. If there is more than one directory that starts with `TARGET` in the search path, only the one specified in the `TARGET` variable will be used.

In the example below, the BSP must be contained in a directory named `TARGET_CY8CKIT-062S2-43012`.

```
# 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
```

Note: The `TARGET` board should have an `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.

Note: `TARGET` is used in the launch configurations (program, debug, etc.) so if you change this variable 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.

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.

Build configuration

The build configuration sets compiler optimization settings. What each selection does is toolchain specific. If you choose Custom, you can manually set any compiler optimization that you want using the `CFLAGS` variable.

```
# 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: `CONFIG` is used in the launch configurations (program, debug, etc.) so if you change this variable 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.

Components

The `COMPONENTS` variable is a way to enable code to be optionally included in a build. Any directory whose name is `COMPONENTS_<name>` will only be included in the build if `<name>` is included in the `COMPONENTS` variable. Use spaces between names if you need more than one COMPONENT.

There is also a variable called `DISABLE_COMPONENTS` which will remove a directory from the build even if it is included elsewhere using the `COMPONENTS` variable. This can be used to override the default behavior of a BSP or library.

```
#####
# 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=
```

Including Files and Directories

The make system automatically searches through the application's directory hierarchy for source files to include in the build. However, there are several mechanisms to include other code if necessary, such as from custom libraries that are located outside the application.

There are three main variables that you can use. Each one can use a relative path and can have a list of more than one item separated by spaces.

SOURCES	Use this to specify a list of C/C++ source files to include in the build.
INCLUDES	Use this to specify a directory containing header files to include in the build. Only the specific directory is included – not sub-directories.
SEARCH	Use this to specify a directory to add to the auto-discovery path. This means that the directory and all of its sub-directories will be searched by the make system. This is the variable that is typically used with custom libraries that are outside the application's directory.

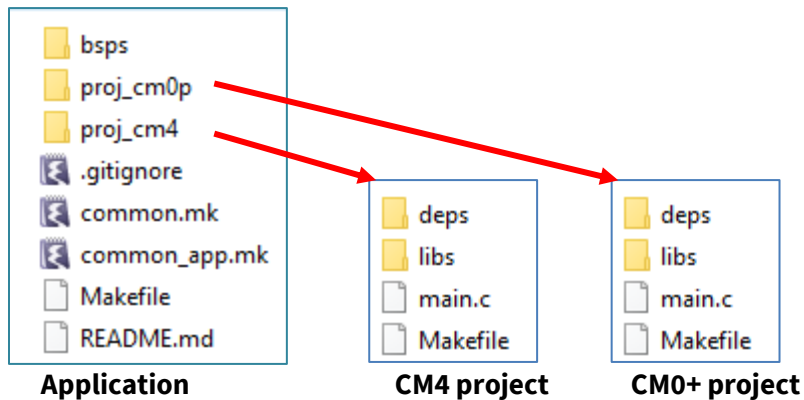
Note: The `SOURCES` and `INCLUDES` variables usually have an empty placeholder in the application's Makefile so you can just add to them.

Note: The `SEARCH` variable is used other places in the build system so you must use `+=` when adding it to the Makefile so that your path doesn't overwrite other paths that have been included. For example:

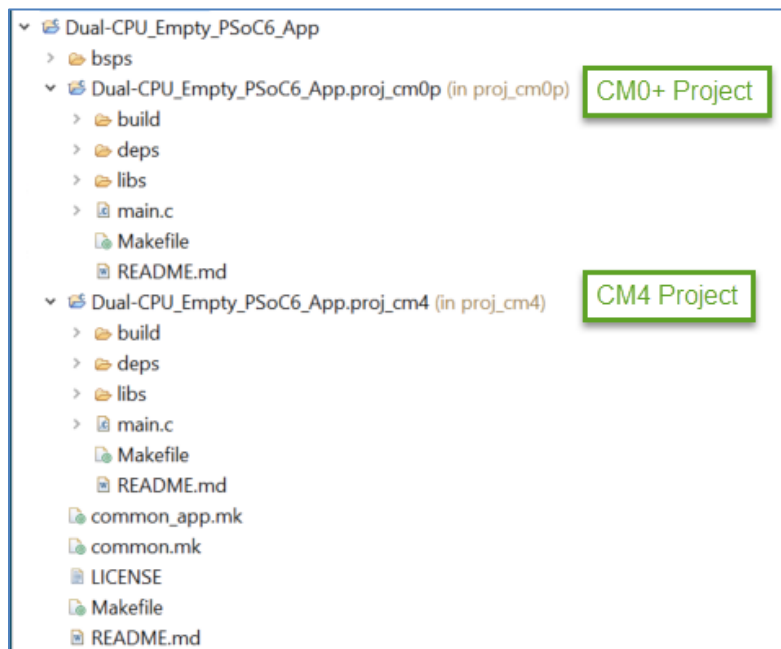
```
SEARCH += ../myLibraries
```

2.5.2 Multi-core applications

Multi-core applications have more than one project associated with a single application –one project per core. The top-level application directory contains application level files and a sub-directory for each of the projects. As an example, a dual-core application for an MCU with an Arm® Cortex®-M4 and an Arm® Cortex®-M0+ MCU may look like this on disk:



Inside an IDE, the application will look something like this:



Note that there is a *deps* and *libs* directory inside each project. That means each project has its own libraries. Likewise, each one has its own *main.c* file since each core will be running separate code. On the other hand, the *bsps* directory is at the application level since the entire application will be running on one physical board.

2.5.2.1 Makefile and make targets

For multi-core applications, there is a *Makefile* at the application level and one inside each project. There are also two files at the application level: one called *common_app.mk* that is included by the *Makefile* in the application and each project; and one called *common.mk* that is included by each project's *Makefile* but not by the application's *Makefile*.

Some operations can be done at both the application and project levels while other operations make sense only at the project level. A few of the common make targets and their behavior is shown here.

Make Targets

Target	Level	Description
help	Both	Print the list of make targets that are available at that level.
build	Both	Build all projects and generate a combined application hex image. The behavior is the same at the application and project levels.
build_proj	Project	Build only the selected project.
program	Both	Build all projects, generate a combined application hex image and program the combined image to the device. The behavior is the same at the application and project levels.
program_proj	Project	Build and program only the selected project to the associated core.
debug	Project	Build only the selected project, launch the debugger for the target core, program the target core, start the application and stop in the target core's <code>main</code> function. <i>Note:</i> From the command line, it is only possible to debug a single core at a time. However, some IDEs support simultaneous multi-core debugging. <i>Note:</i> You should program the other core before running the debug target since it only programs the core being debugged.
device-configurator	Both	Run the device configurator. It will show the configuration for both cores and will allow you to specify which core each peripheral belongs to. The behavior is the same at the application and project levels.
library-manager	Both	Run the library manager. It will show libraries for each core and will allow you to manage libraries for each core separately. The BSP section applies to both cores. The behavior is the same at the application and project levels.

Make Variables

Make variables to control the build are the same as for single-core applications but they are split between the different files depending on how they are used (application only, project only, or both).

The `MTB_TYPE` variable is set to a different value than single-core applications. For the application level *Makefile*, it is set to `APPLICATION` while for each project *Makefile*, it is set to `PROJECT`.

One other difference from single-core applications is that the top level *Makefile* has a variable called `MTB_PROJECTS` that lists the projects that are a part of the application. For example, the application shown above would have these lines in its *Makefile*:

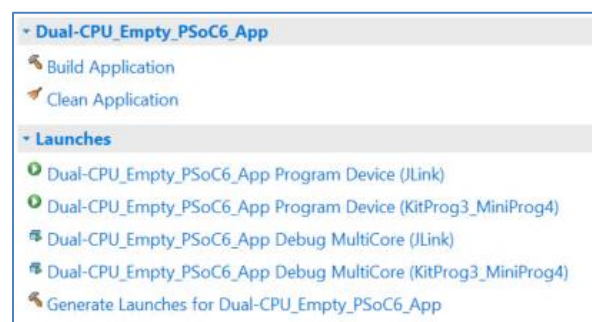
```
Application level:      MTB_TYPE=APPLICATION
                        MTB_PROJECTS=proj_cm0p proj_cm4
```

The *common.mk* file (included in the CM0+ and CM4 projects) would have:

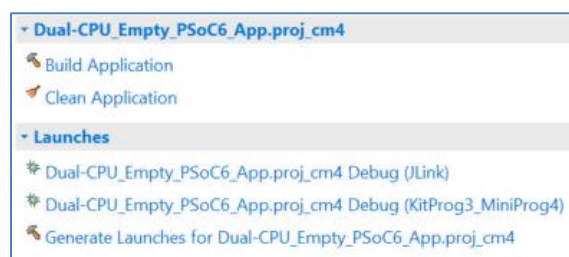
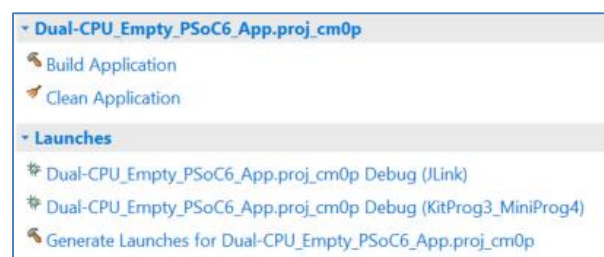
```
Project level:         MTB_TYPE=PROJECT
```

2.5.2.2 Build, Program and Debug in the Eclipse IDE for ModusToolbox™

If you are using the Eclipse IDE for ModusToolbox™, the application and each project will have a set of build and launch configurations like this:



The project levels will look like this:



From the Quick panel, you can build or program the entire application either from the application level or one of the sub-projects.

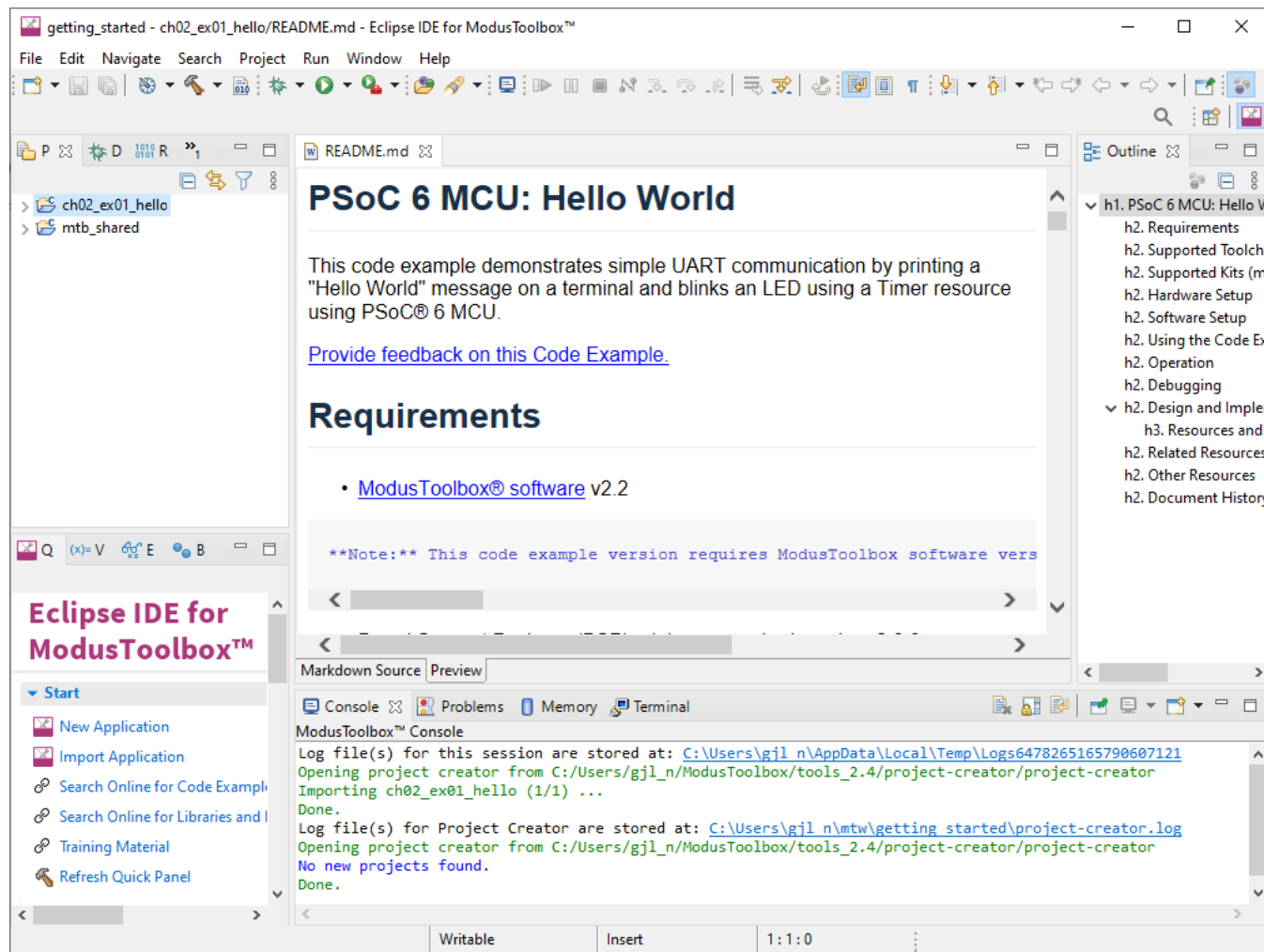
For debug, you can run multi-core debug from the application level or debug on one of the cores from the respective project level. In either case, the entire application is built and programmed before starting debug.

Other launch configurations such as Attach and Erase can be found in the **Run > Run Configurations** and **Run > Debug Configurations** menus.

2.6 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.

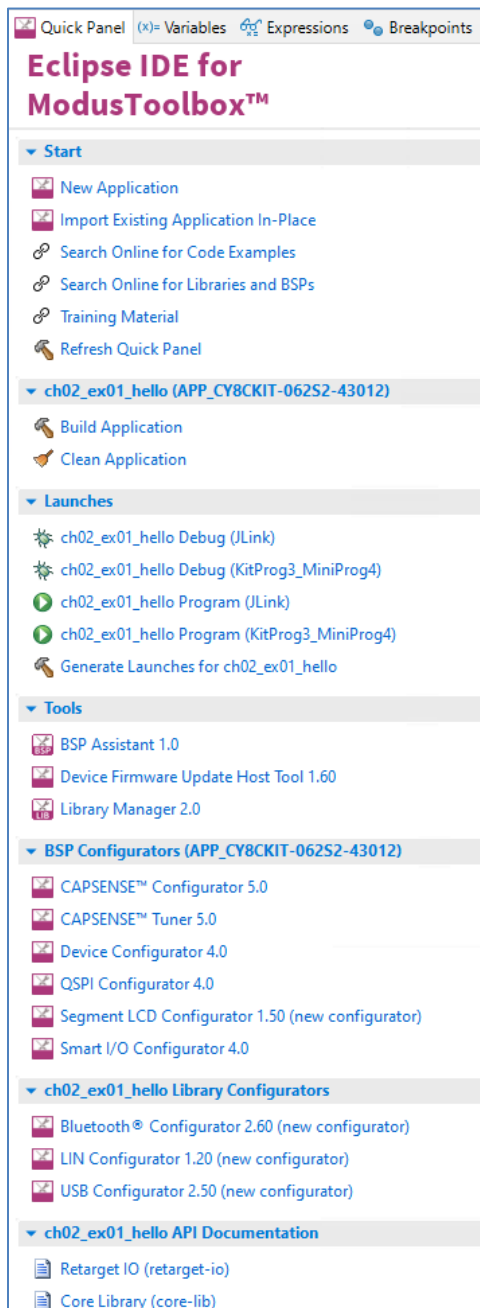


For more information about the IDE, refer to the Eclipse IDE for ModusToolbox™ User Guide which can be found at www.infineon.com/MTBEclipseIDEUserguide.

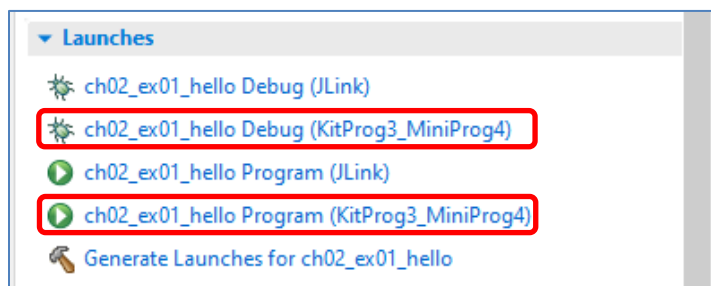
2.6.1 Quick Panel

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
- Import an existing ModusToolbox™ application in-place
- Clean & build
- Program/Debug launches
- Tools & Configurators
- Documentation

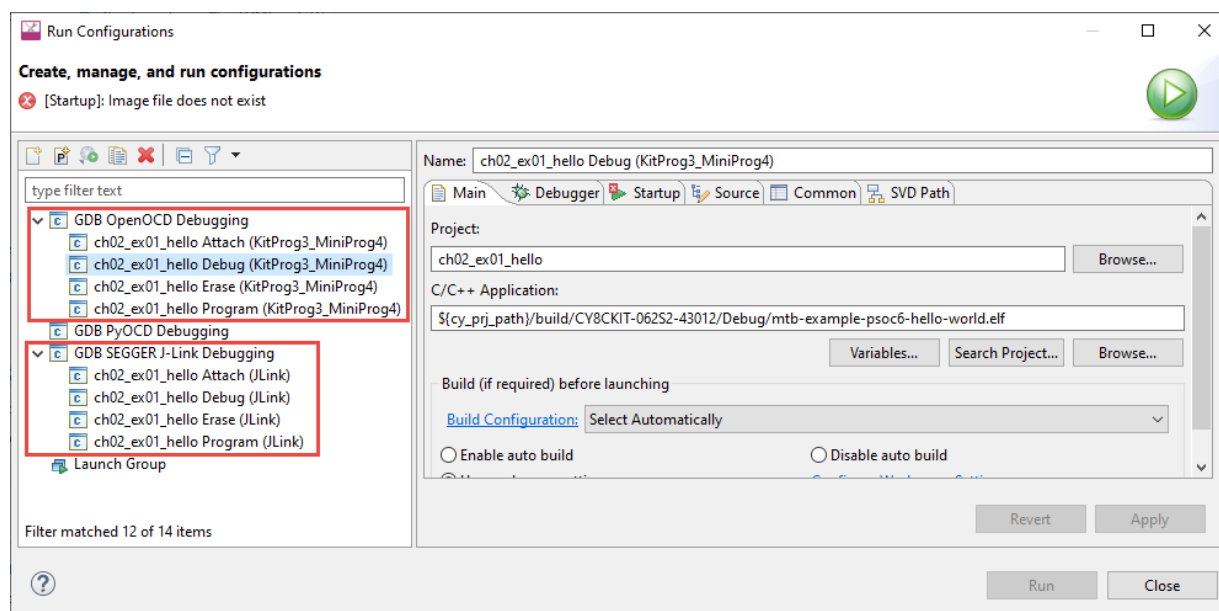


The launches section is what you should use when you want to program a kit or start the debugger. If you are using an Infineon PSoC™, you should use the selections that say "KitProg3_MiniProg4" since most kits contain an integrated KitProg3 programmer. For example:



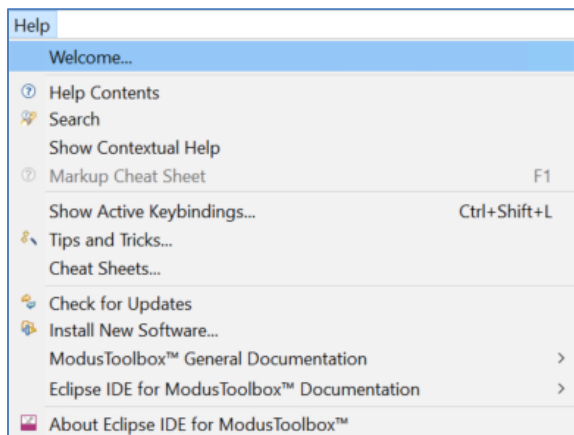
You can also access these using the bug and play icons along the top ribbon, but make sure you select the correct one from the drop down. For that reason, the Quick Panel is usually easier to use.

In addition to the launch configurations in the Quick Panel, there are others that you can find under **Run > Run Configurations** and **Run > Debug Configurations**. 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. If you want to launch the debugger, make sure you launch from the debug configurations. Otherwise, the CPU will not halt at the beginning of main when debugging starts.



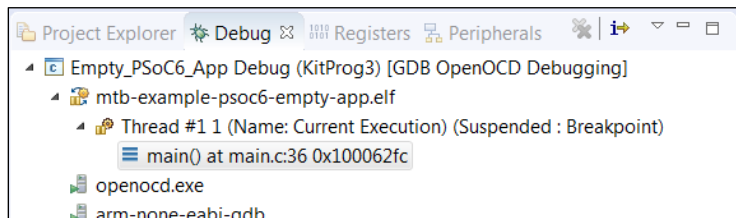
2.6.2 Help menu

The IDE Help menu provides links to general documentation, such as the ModusToolbox™ User Guide, ModusToolbox™ Release Notes, and training classes like this one as well as IDE-specific documentation, such as the Eclipse IDE for ModusToolbox™ Quick Start Guide.



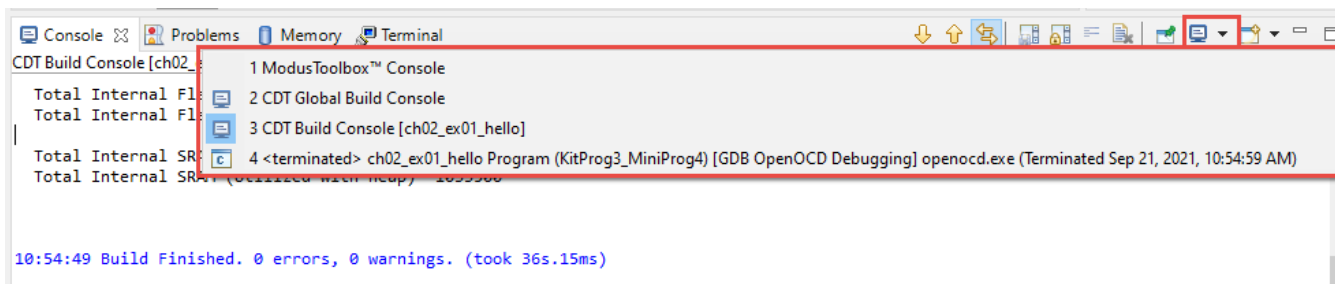
2.6.3 Integrated debugger

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.



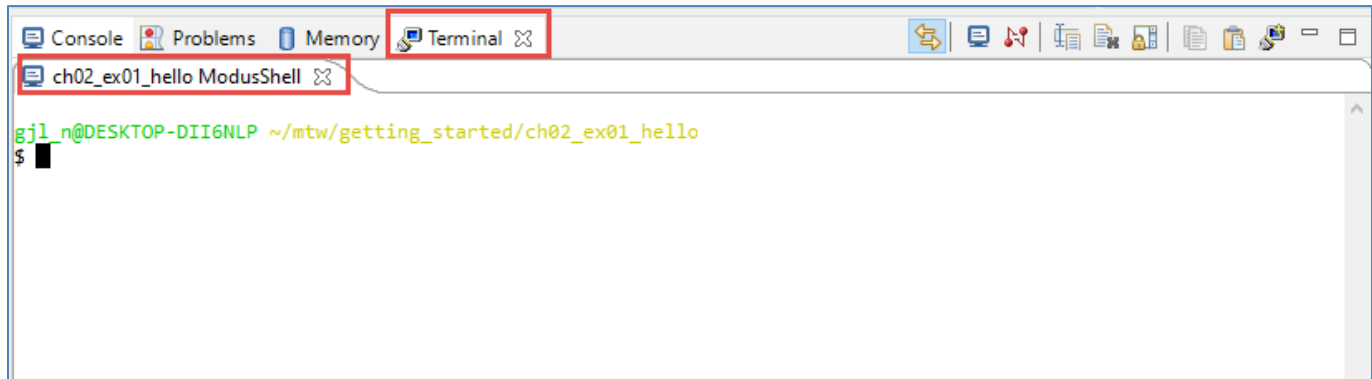
2.6.4 Console

Near the bottom of the default perspective is a console window that shows general log information as well as information from build and program operations. Use the **Display Selected Console** drop-down menu on the right side to choose which console is displayed. This is especially useful to see messages from a build that occurs before a program operation since the default will just show the result of the programming step.



2.6.5 Terminal

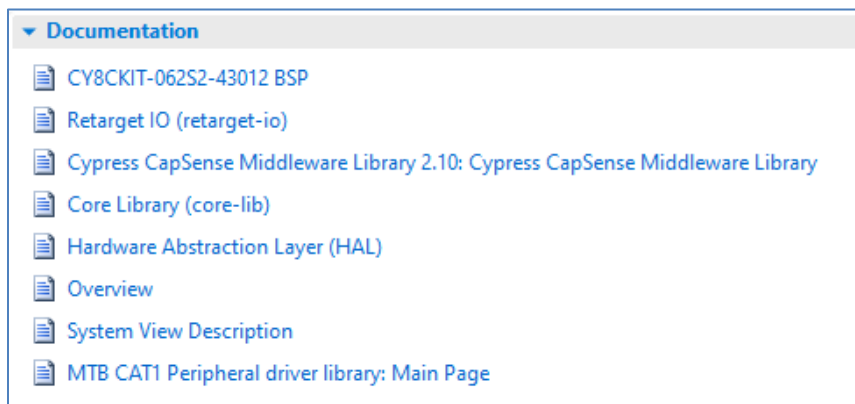
In the same window as the **Console**, there is a tab labeled **Terminal**. This tab provides a command line interface for running command line operations without having to leave the Eclipse environment. There will be one sub-tab for every project that you have accessed. Each sub-tab will automatically start out in the top-level directory for that specific project.



See section 2.2.2 for details on using the command line interface.

2.6.6 Documentation

After creating a project, assorted links to documentation are available directly from the Quick Panel, under the "Documentation" section. This will update to include documentation for new libraries as you add them to the application.

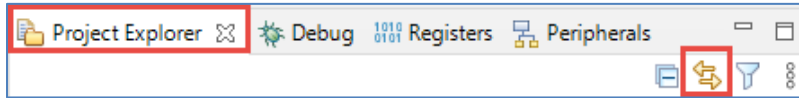


2.6.7 Eclipse IDE tips & tricks

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

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

- Set/clear breakpoints
- 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.

2.7

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 BSP 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 (shared vs. local, direct vs. indirect, and fixed vs. dynamic). Each classification is explained below.

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.

Starting with the ModusToolbox™ 2.2 tools release, the method by which applications reference libraries was modified such that applications typically share libraries by default. 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 the *mtb_shared* directory parallel 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 by using the Library Manager, which is discussed in section [2.7.3](#).

Prior to the ModusToolbox™ 2.2 tools release, a different library management method (called the LIB flow) was in use. This method is not supported in ModusToolbox 3.0 and later so 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.

As mentioned above, the source code for all libraries will be put in a shared directory by default. 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 sometimes be changed for applications with hierarchy such as multi-core applications. For most single-core 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 multi-core applications, there is an additional level of hierarchy so the path is set to `../../` in each of the project *Makefiles*.

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

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). For example, some applications require the *retarget-io* library, so they include that library as a direct dependency. In other applications, there will be additional direct dependencies such as Wi-Fi libraries or Bluetooth® libraries. BSPs require various libraries to work with the device such as the HAL and PDL, so those libraries are included as indirect dependencies.

2.7.1.3 Fixed vs. dynamic versions

You can specify a fixed version of a library to use in your application (e.g. release-v1.1.0), 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 using the Project Creator tool, the libraries will be converted to fixed versions, so the versions of libraries in a given application will never change unless you specifically request it. This is called "latest locking" and it is done automatically during project creation based on information in the manifest file (more on manifests in section 2.13). The Library Manager can be used to control fixed vs. dynamic versions for each library.

Note: If you don't use the Project Creator tool to create the project, latest locking is not done so it is important to use the Library Manager to select fixed version for each library unless you want them to dynamically change to the latest version whenever the Library Manager or `make getlibs` is run on the application.

2.7.2 Including libraries and dependencies

There are two basic ways of including a library and its dependencies into your application: using a manifest file or using manual methods. There is also a third method that is only used for custom BSPs. Each method is discussed separately below. The method you should use depends on whether or not the library and its dependencies are specified in a manifest file.

2.7.2.1 Including libraries that are specified in a manifest

All libraries provided by Infineon are specified in manifest files that are automatically found by the tools. The manifest files specify where each library can be found and what each library's dependencies are. The manifest can also specify specific versions of libraries and dependent libraries that are meant to work together.

You can create your own manifest file(s) for your libraries so that they will show up in the Library Manager and will work just like the Infineon libraries. More information on manifest files and how to create and use your own manifests is covered in section 2.13.

In order to manage libraries that are specified in a manifest file in the application, files with the extension *mtb* are used. Their location and behavior depend on whether the library is a direct dependency or an indirect dependency. Normally the Library Manager is used to create/modify *mtb* files so you won't need to change them or even view them, but it is worthwhile to understand what is in them.

For direct dependencies, there will be one or more *mtb* files somewhere in 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](https://github.com/cypresssemiconductorco/TARGET_CY8CKIT-062S2-43012/#latest-v1.X#$$LOCAL$$/TARGET_CY8CKIT-062S2-43012)

Note: The examples above specify dynamic library versions (e.g. latest-v1.X). Normally in your application, you will want fixed library versions (e.g. release-v1.0.0). This behavior can be controlled using the Library Manager.

For indirect dependencies, an *mtb* file for each library is automatically created in the *libs* directory. The version of each dependent library is also captured in the file *locking_commit.log* in the *deps* directory.

Once all the *mtb* files are in the application (both direct and indirect), the libraries they point to are pulled in from the specified Git repos and stored 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 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 by re-running `make getlibs`. This includes both shared libraries (in *mtb_shared*) and local libraries (in *libs*) - they all get pulled from GitHub when you run `make getlibs`.

You also don't typically need to check in the *mtb* files for indirect references and the *mtb.mk* file which are 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 - they can be recreated at any time.

2.7.2.2 Including libraries are not specified in a manifest

If you have your own custom libraries, you can create and include your own custom manifest files so that your libraries will show up in the Library Manager just like Infineon libraries. That method also allows you to specify the dependencies in the manifest so that they are automatically added as indirect dependencies when the dependent library is added to the application. Custom manifests are covered in section 2.13.

If you do not want to create a manifest file for your custom libraries then you must manually include the library and its dependencies in each application that uses the library. Including the library can be done several different ways including:

- `git clone` the version of the library that you want (or use some other revision control system) into the application's directory
- unzip an archive file of the library into the application's directory
- recursive copy the library into the application's directory
- modify the `SEARCH` variable in the application's *Makefile* to point to the library
- add an *mtb* file to the *deps* directory for the library and run `make getlibs` (requires that the library is in a Git repo)

For the first 4 methods, the library can go anywhere in the application's directory tree. Remember that if you put it in the *libs* directory it will not be checked into version control by default, so you will normally want put manually added libraries somewhere else.

If you use the *mtb* file method, there are two things to be aware of: (1) the library will still not show up in the Library Manager; and (2) latest locking will not be performed on that library so you should use a fixed version in the *mtb* file unless you want the dynamic update behavior.

Adding dependencies can be done most easily by using the Library Manager tool (for dependencies that are in a manifest file such as Infineon libraries). For dependencies that are not in a manifest file, you can use the same manual methods that are used for including the library itself.

2.7.2.3 Specifying dependencies for custom BSPs

Custom BSPs use an alternate method to specify dependencies. The advantage is that the specification of the dependencies is self-contained within the custom BSP itself.

This method is accomplished by having files with the extension *mtbx* for each dependency in the *deps* directory inside the custom BSP. An *mtbx* file has the exact same content as an *mtb* file with a different file extension.

The process of getting dependencies from *mtbx* files only searches in the target BSP for the application so if your application contains multiple custom BSPs, only the dependencies for the active target (as specified in the `TARGET` variable in the *Makefile* or on the command line) will be included in the application.

The *mtbx* files are automatically created for you so you typically don't need to do anything with them unless you want to add or remove dependencies from your custom BSP. The BSP Assistant can be used to do this. It will be discussed in section 2.1.

During `make getlibs`, *mtbx* files are treated just like indirect dependencies that are specified in a manifest file. That is, the process will copy each *mtbx* file to the application's *libs* directory and will lock the version in the *locking_commit.log* file in the *deps* directory. The libraries are then pulled down according to the information contained in the *mtb* files.

2.7.3 Library Manager tool

The ModusToolbox™ ecosystem provides a GUI for helping you manage library files. There are many ways to run it:

1. From the Eclipse IDE, use the Quick Panel link "Library Manager"
2. From the CLI, run `make library-manager` from the application directory
3. Launch from the Windows **Start** menu
4. Entering `library-manager` in the Windows search box.
5. Launch from the installation directory. For example, `ModusToolbox/tools_<version>/library-manager`

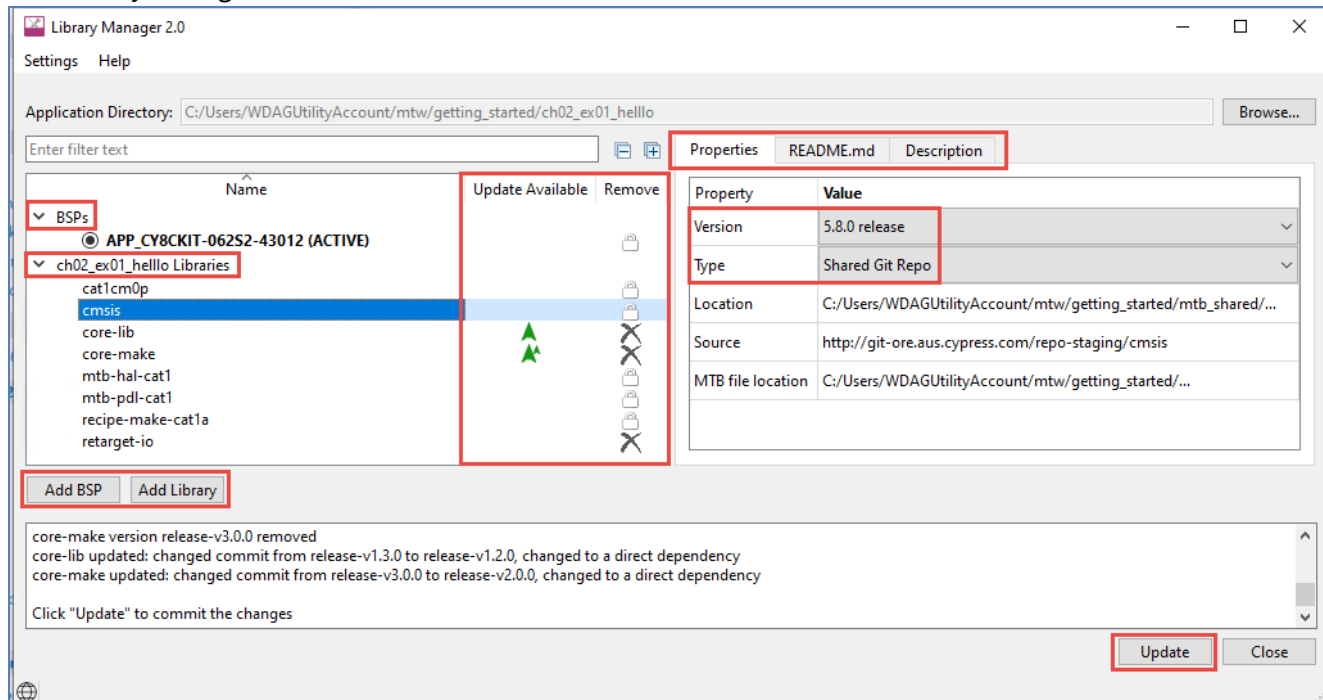
The Library Manager knows where to find libraries and their dependencies by using manifest files, which will be discussed in section 2.13. Therefore, *mtb* files included manually in your application that are not in a manifest file will NOT show up in the Library Manager.

One important note – the Library Manager never changes anything on disk until the **Update** button is clicked.

Behind the scenes, the Library Manager reads/creates/modifies *mtb* 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* file and then runs `make getlibs` to get the specified version.

The Library Manager GUI looks like this:



As you can see, there is a section for *BSPs* and a section for *Libraries*. If you are working with a multi-core application, there will be a section with libraries for each project.

The *BSPs* section allows you to specify which BSP 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 **X** symbol in the **Remove** column allows you to remove a BSP from the application but you always need at least one BSP in an application.

In the Libraries section, clicking the **X** symbol in the **Remove** column removes the library. A lock symbol means that the library is a dependency of the BSP or another library, so it cannot be removed.

The "Update Available" column shows when a library has a newer version available than the one being used in the application. A 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.



In the right panel, information about the selected BSP or library is shown. There are tabs for **Properties**, **README.md** and **Description** with different information. The **Properties** tab is where you can make changes to versions, locations, etc. Some of the commonly used items are:

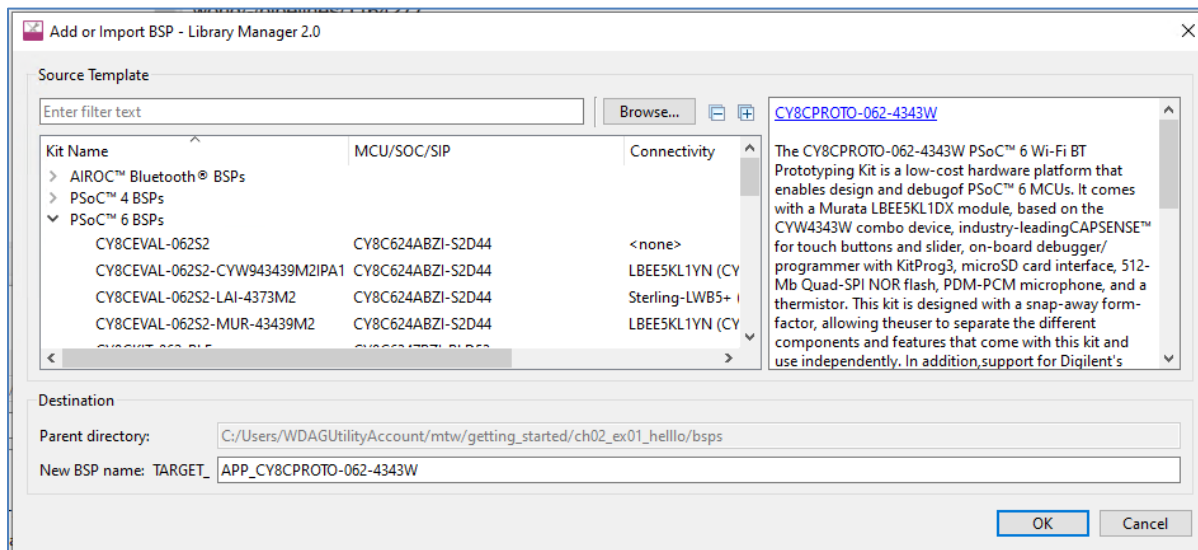
Version: You can select a specific fixed version of a library for your application or choose a dynamic tag that points to a major version but allows `make getlibs` to fetch the latest minor version (e.g. Latest 1.X). The drop-down will list all available versions of the library.

Type: Libraries can be "Shared Git Repo" or "Local Git Repo". By default, most libraries are shared, meaning that the source code for the libraries is placed in the workspace's shared location (e.g. `mtb_shared`).

If you change the value (either version or type) for an indirect library, it will become a direct dependency. To change it back to an indirect dependency, just click the **X** next to the library name to remove the direct dependency.

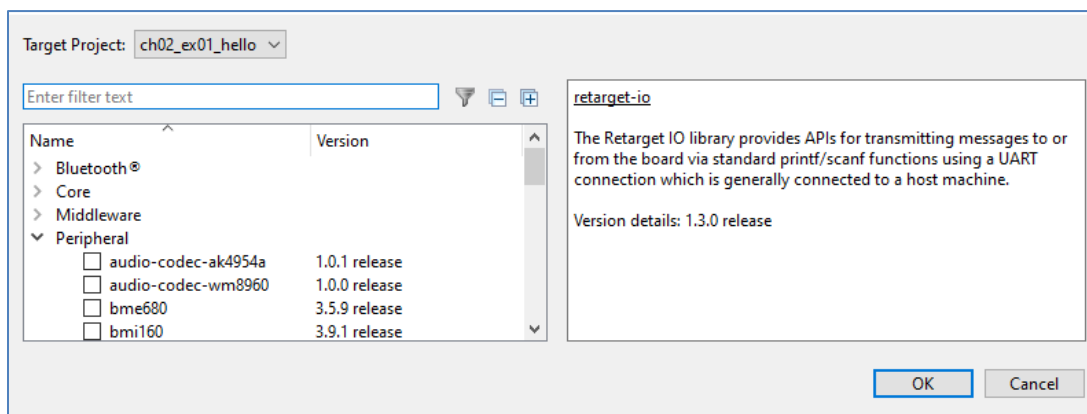
Note: The Version and Type cannot be changed for a ModusToolbox 3.X format BSP since it is flat data inside the application.

The **Add BSP** button allows you to pull an additional BSP into the application. It opens a window like this:



You can select a standard BSP and the version that you want to use, or you can click **Browse** to specify the path to a custom BSP. You can use the **+** and **-** buttons at the top to expand/collapse all categories and the text box to filter out specific BSPs. You can override the default name for the new BSP that will be created.

The **Add Library** button allows you to add new libraries. It opens a window like this:



You can use the + and - buttons at the top to expand/collapse all categories and the text box to filter out specific libraries.

Click the check box next to one or more libraries to add them to the project. You can select a different version for the library if you don't want the default. The filter button will show only libraries that are currently checked.

If you are working with a multi-core application, the drop-down menu at the top allows you to select which project you want to add the libraries to.

2.7.3.1 Active BSP

As described previously, one BSP is always selected as the active BSP. This is the target that will be used during a build. When the active BSP is changed, the Library Manager updates the `TARGET` variable in the application's *Makefile*.

If you are using the Eclipse IDE, the Library Manager will also update the Eclipse launch configs so that it will program the correct hex file onto the kit. If you were to manually edit the `TARGET` variable in the *Makefile*, there is a link in the Quick Panel that says "Generate Launches for <app-name>" that you can use to fix them.

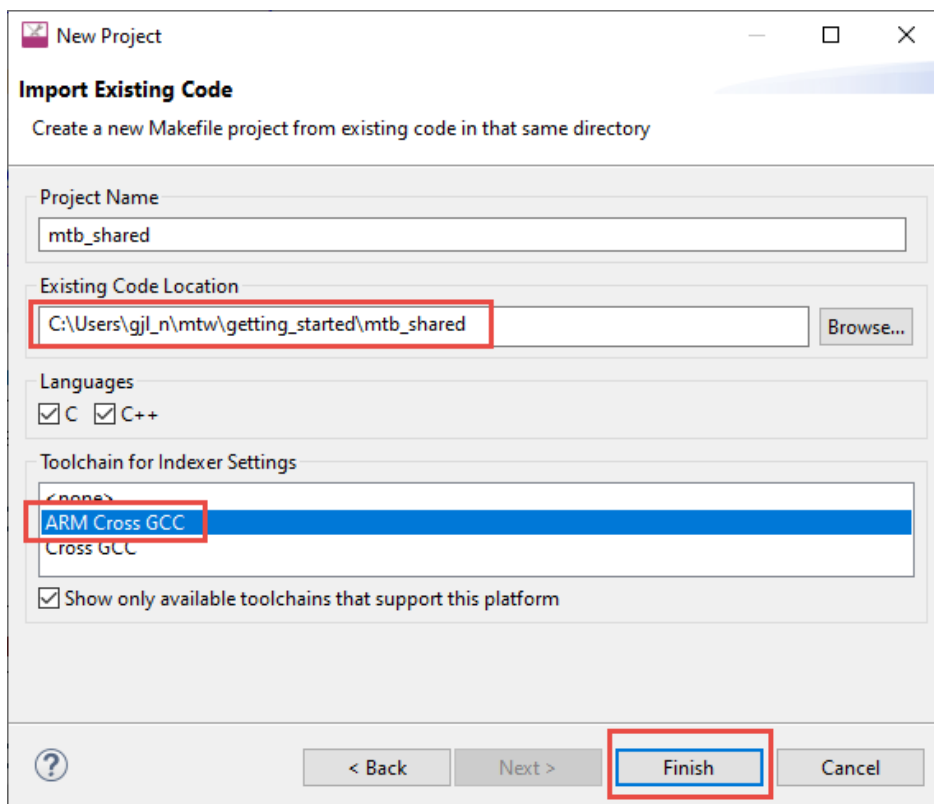
In the case of VS Code, after updating the Active BSP or after manually editing the `TARGET` in the *Makefile*, you must manually update the appropriate launch configurations. You can re-run `make vscode` to regenerate the launch configurations, but keep in mind that this may over-write other VS Code 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`.

2.7.4 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. However, the *mtb_shared* directory will not have Eclipse project information. This means 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.

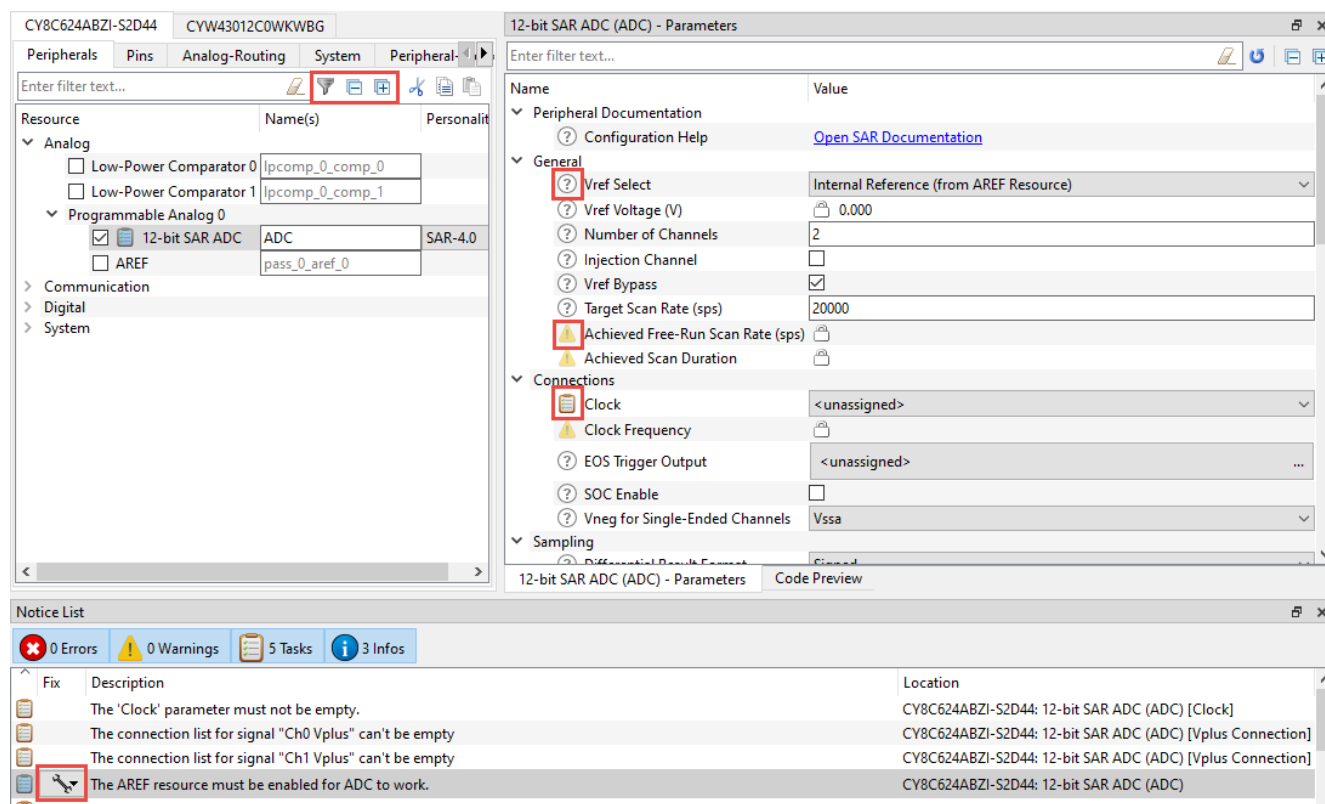
2.8.1 BSP configurators

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. These configurators can be launched from the BSP Assistant (discussed later) or from within the application. However you launch them, the configurators are modifying files that are a part of the BSP.

Note: All BSP configurators share the same file (design.modus). 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.



Note: 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.

Note: You can use copy/paste within the configurator to copy configurations for entire items. For example, you can copy/paste the entire configuration from one pin to another. To do this, just right-click on the item you want to copy from, select **Copy**, right-click the item you want to copy to, and select **Paste**. If you use cut/paste, the old item will be disabled and the new item will be enabled.

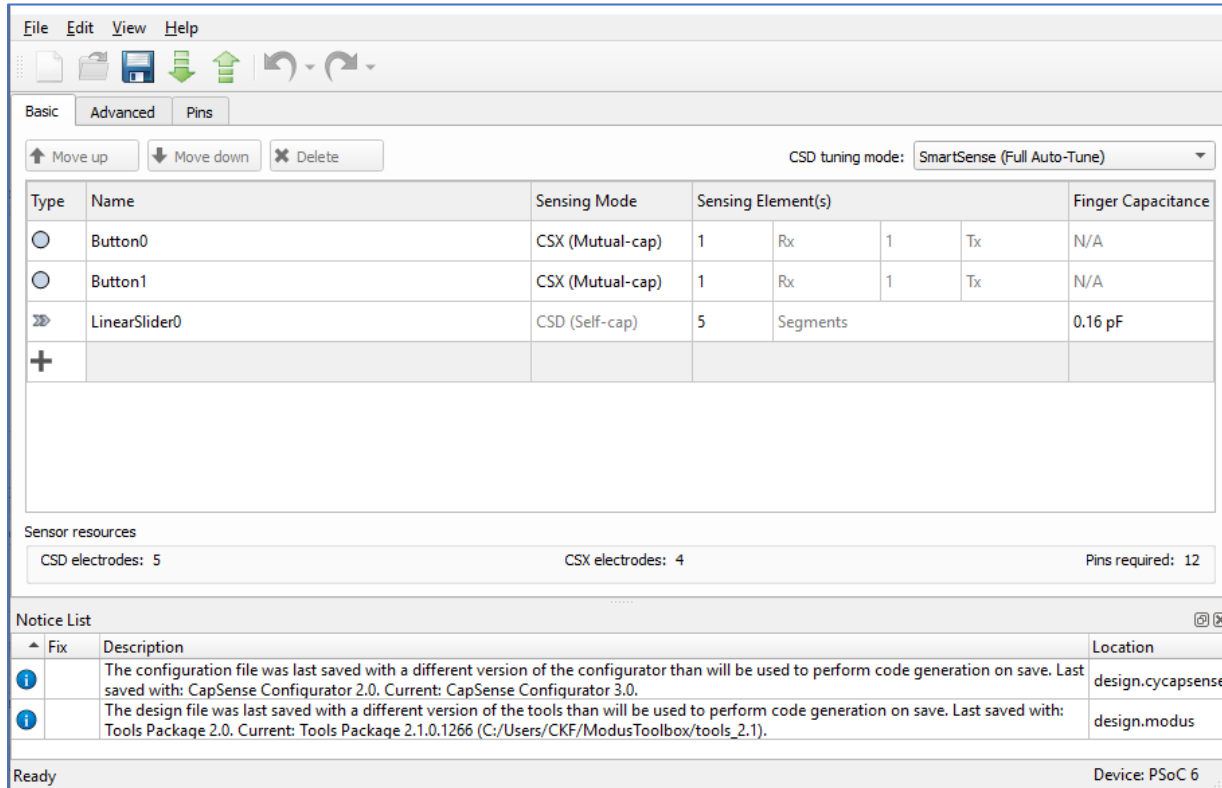
Several helpful icons will show up next to various items in the configurator. A few of interest are:

Icon	Name	Function	Example
	Chain	Clicking this icon will take you to the corresponding resource's configuration page.	When a peripheral has a pin associated with it, clicking the chain icon will take you to the configuration page for that pin.
	Notepad	This icon will appear when something needs to be configured.	If you enable a component that requires a clock, this icon will appear next to the clock selection row.
	Wrench	This icon appears in the log area when there are incompatible selections made across components. The drop-down will suggest fixes that can be applied automatically.	If you enable an ADC and select an internal reference, this icon will show up if the reference is not enabled. The drop down will allow you to enable the reference.
	Question Mark	This icon displays information about the parameter when you hover over it.	The Clock Frequency message may say "Source clock frequency".
	Exclamation Point	This icon displays an error message when you hover over it. Error icons will go away once items are properly configured.	If a clock has not been enabled for an ADC, calculated parameters such as scan rate cannot be determined.

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.

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 configurator interface shows a table of sensor elements and their configurations.

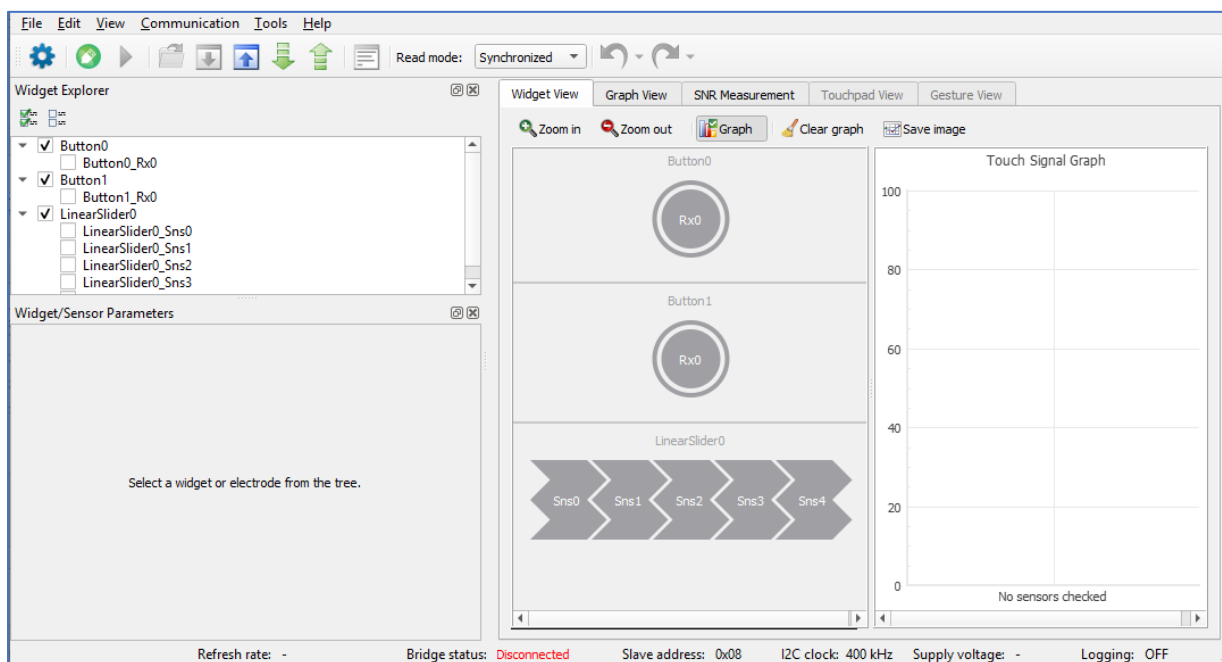
Type	Name	Sensing Mode	Sensing Element(s)				Finger Capacitance
	Button0	CSX (Mutual-cap)	1	Rx	1	Tx	N/A
	Button1	CSX (Mutual-cap)	1	Rx	1	Tx	N/A
	LinearSlider0	CSD (Self-cap)	5	Segments			0.16 pF

Sensor resources: CSD electrodes: 5, CSX electrodes: 4, Pins required: 12

Notice List:

Fix	Description	Location
	The configuration file was last saved with a different version of the configurator than will be used to perform code generation on save. Last saved with: CapSense Configurator 2.0. Current: CapSense Configurator 3.0.	design.cycapsense
	The design file was last saved with a different version of the tools than will be used to perform code generation on save. Last saved with: Tools Package 2.0. Current: Tools Package 2.1.0.1266 (C:/Users/CKF/ModusToolbox/tools_2.1).	design.modus

Ready Device: PSoC 6



The tuner interface displays sensor data in real-time. It includes a Widget Explorer, Widget/Sensor Parameters, and a Touch Signal Graph.

Widget Explorer:

- Button0
 - Button0_Rx0
- Button1
 - Button1_Rx0
- LinearSlider0
 - LinearSlider0_Sns0
 - LinearSlider0_Sns1
 - LinearSlider0_Sns2
 - LinearSlider0_Sns3
 - LinearSlider0_Sns4

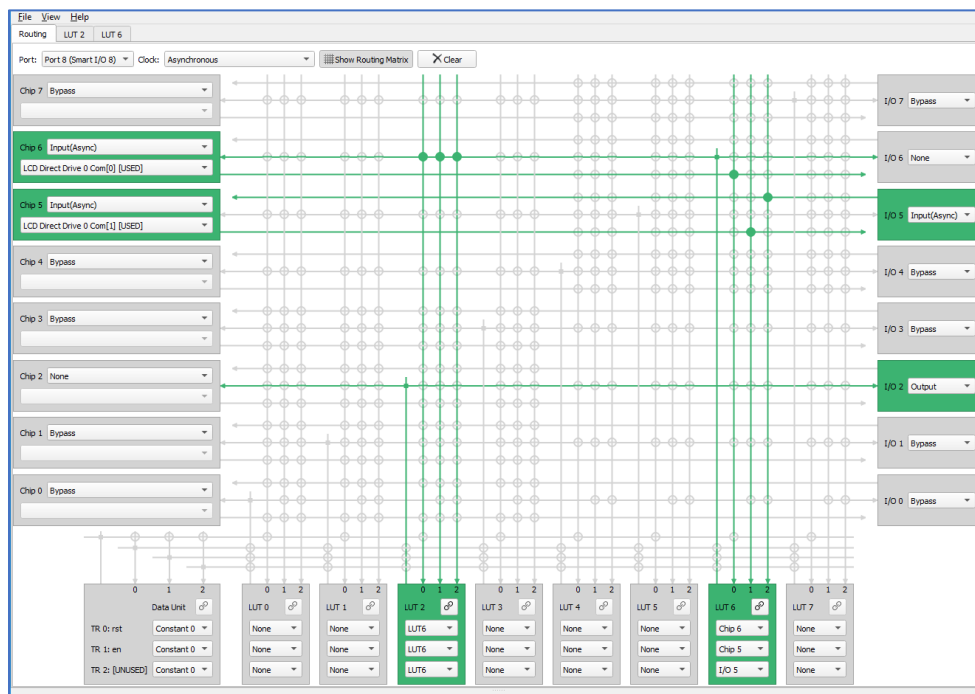
Widget/Sensor Parameters: Select a widget or electrode from the tree.

Touch Signal Graph: A graph showing signal levels for Rx0 and Sns0-Sns4. The Y-axis ranges from 0 to 100. The X-axis shows segments for Sns0-Sns4. The graph is currently empty, indicating no sensors are checked.

Bridge status: Disconnected. Slave address: 0x08. I2C clock: 400 kHz. Supply voltage: -. Logging: 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. 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].

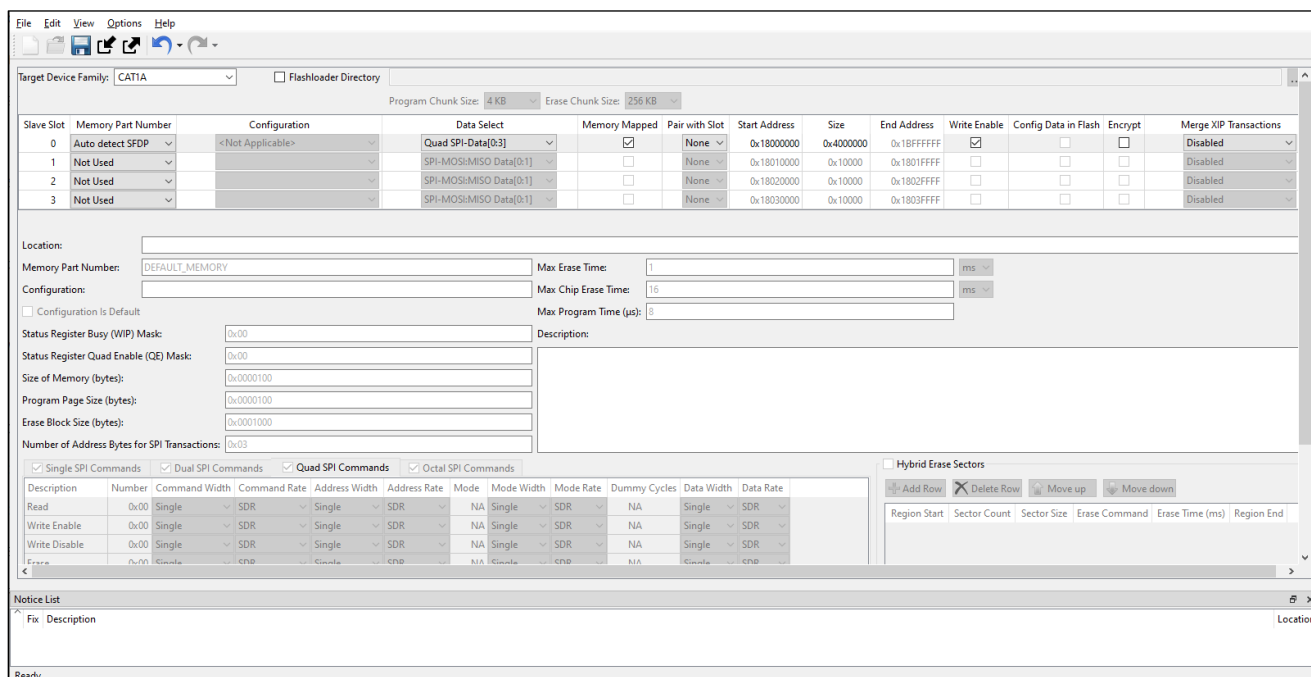
The bottom edge contains LUTs which can be used to combine and route signals to/from the chip's peripherals and to/from the chip's pins. The LUT inputs can come from peripherals, I/O pins, or from another LUT. Once you enable a LUT by selecting its inputs, a tab along the top appears for you to setup 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 Infineon GitHub site).

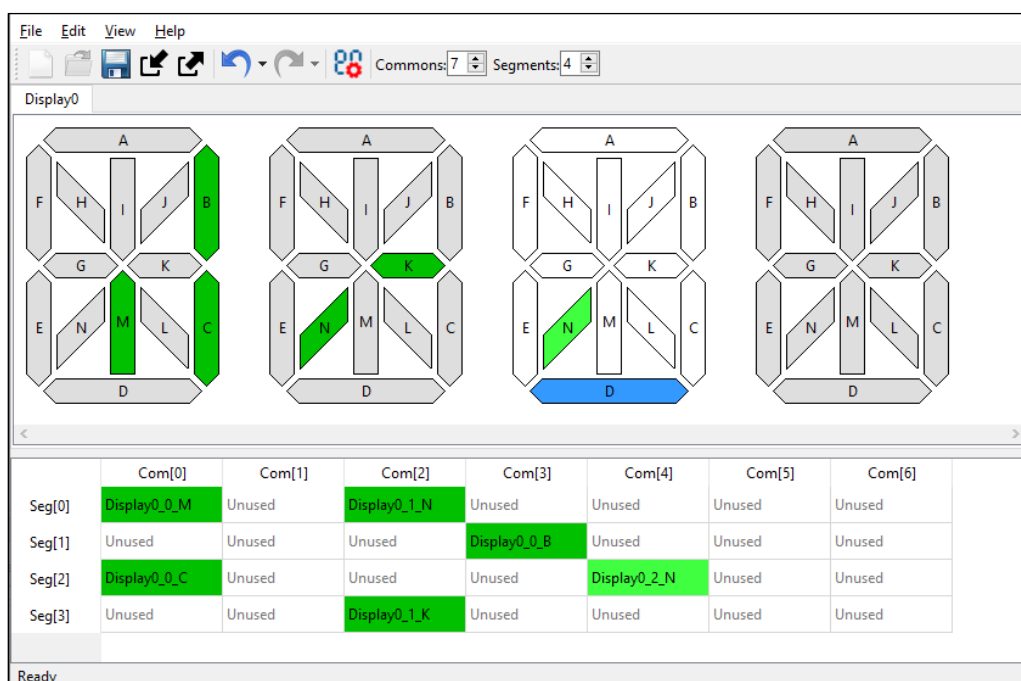
2.8.1.4 QSPI configurator

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.

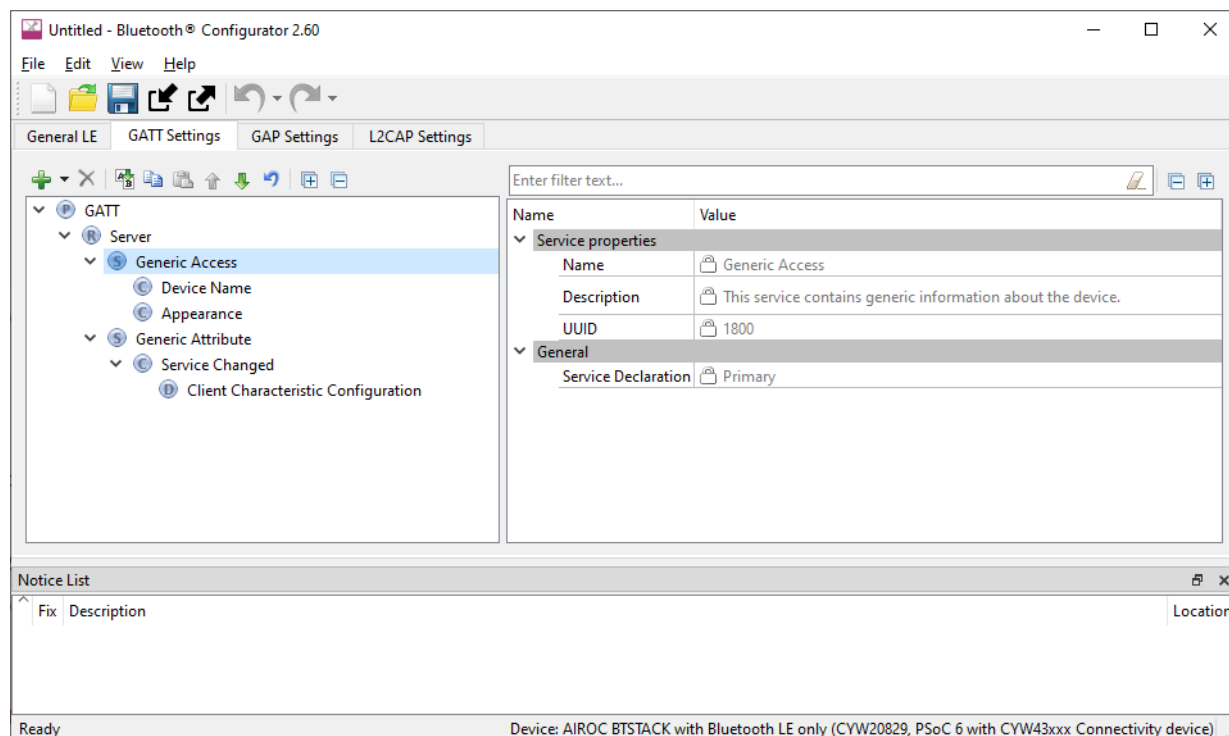


2.8.2 Library configurators

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.

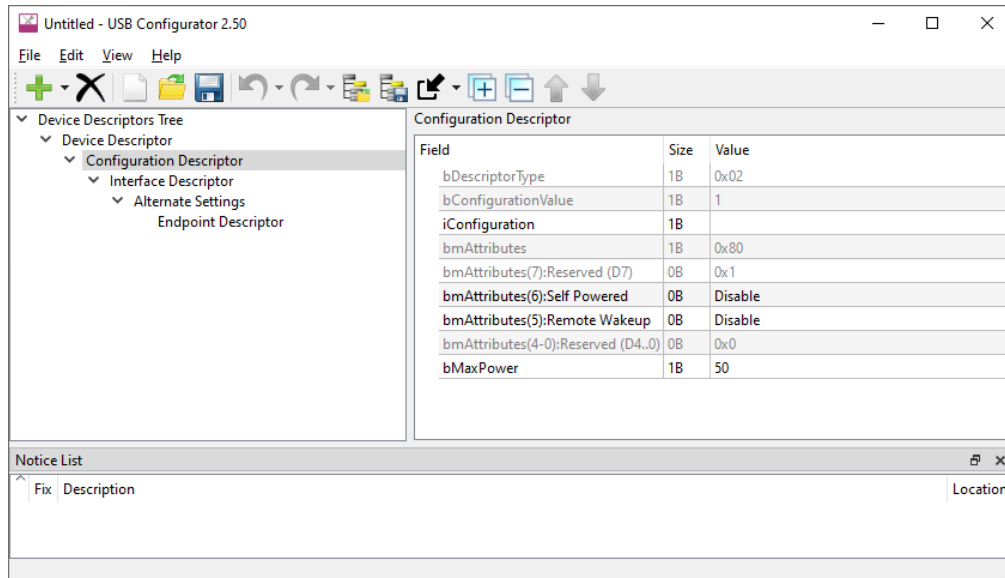
2.8.2.1 Bluetooth® configurator

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

2.9 Working with custom BSPs

2.9.1 BSP Assistant

The ModusToolbox™ ecosystem provides a GUI for helping you create and edit BSPs.

2.9.1.1 Opening the tool

There are several ways to run the BSP Assistant:

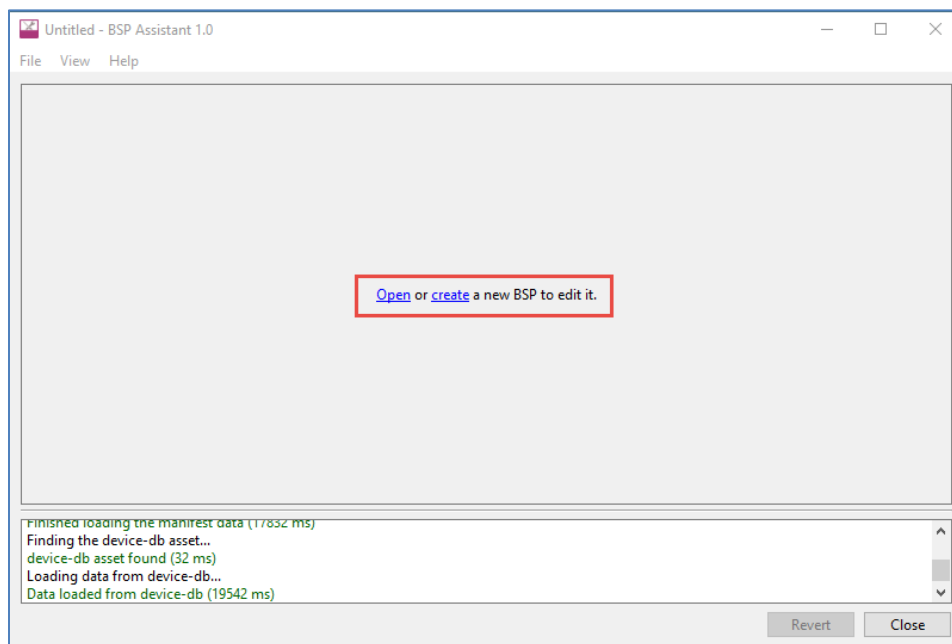
To open a BSP for an existing application:

- From the Eclipse IDE, use the Quick Panel link "BSP Assistant"
- From a make command terminal, run `make bsp-assistant`

To open an empty BSP Assistant GUI:

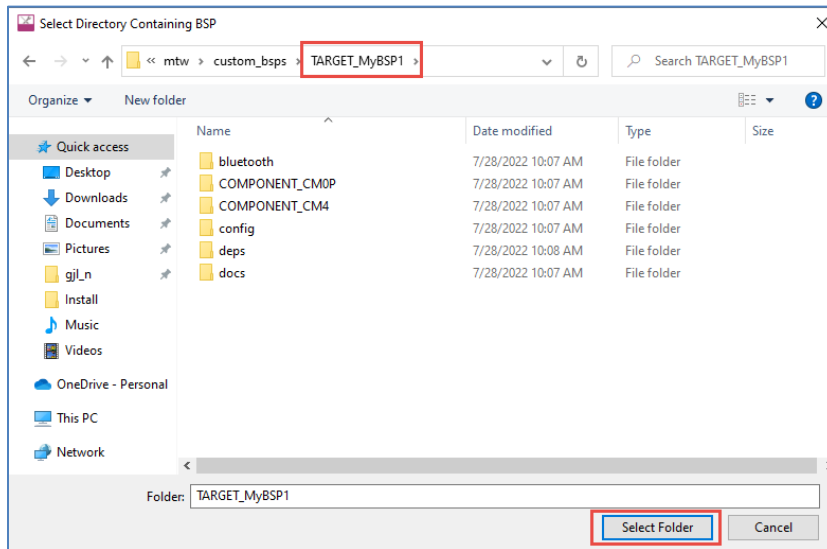
- Launch from the Windows **Start** menu
- Entering `bsp-assistant` in the Windows search box.
- Launch from the installation directory. For example, `ModusToolbox/tools_<version>/bsp-assistant`

When you launch the GUI, it first loads the technology packs, tools information, and manifest data. Once that's done, if you opened the tool without a BSP, it will look like the following image. You can either open an existing BSP or create a new one by clicking one of the links or using the **File** menu.



2.9.1.2 Open existing BSP

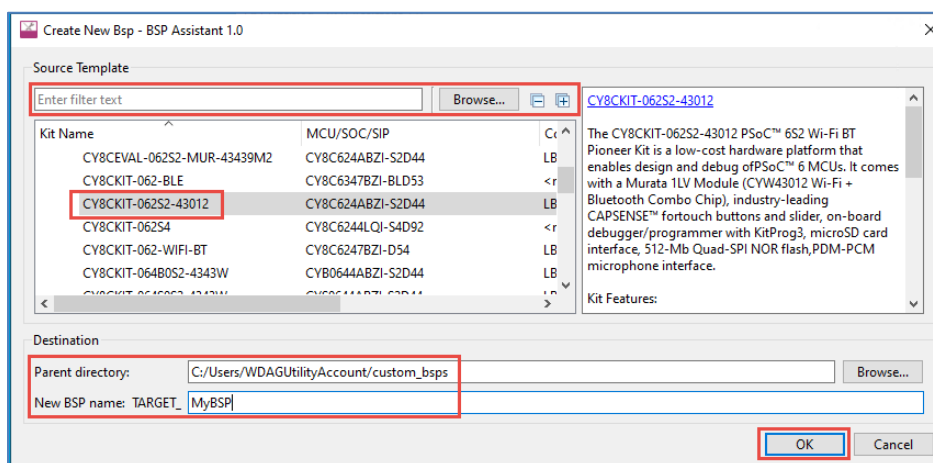
On the main BSP Assistant window, click the **Open** link or select **File > Open** to open the standard Open dialog. Then, navigate to the directory that contains the desired BSP and click **Select Folder**.



Note: If the specified BSP has a .git directory that points to an Infineon repository, a warning dialog will indicate that you might want to use the New... dialog to use the BSP as a template to create a new BSP in a new location that is not connected to Infineon's Git repositories before customizing it.

2.9.1.3 Create new BSP

If you create a new BSP, a dialog opens that is similar to the BSP page in the Project Creator tool. Select a standard BSP to use as the template, or use the **Browse** button at the top to specify an existing local BSP.

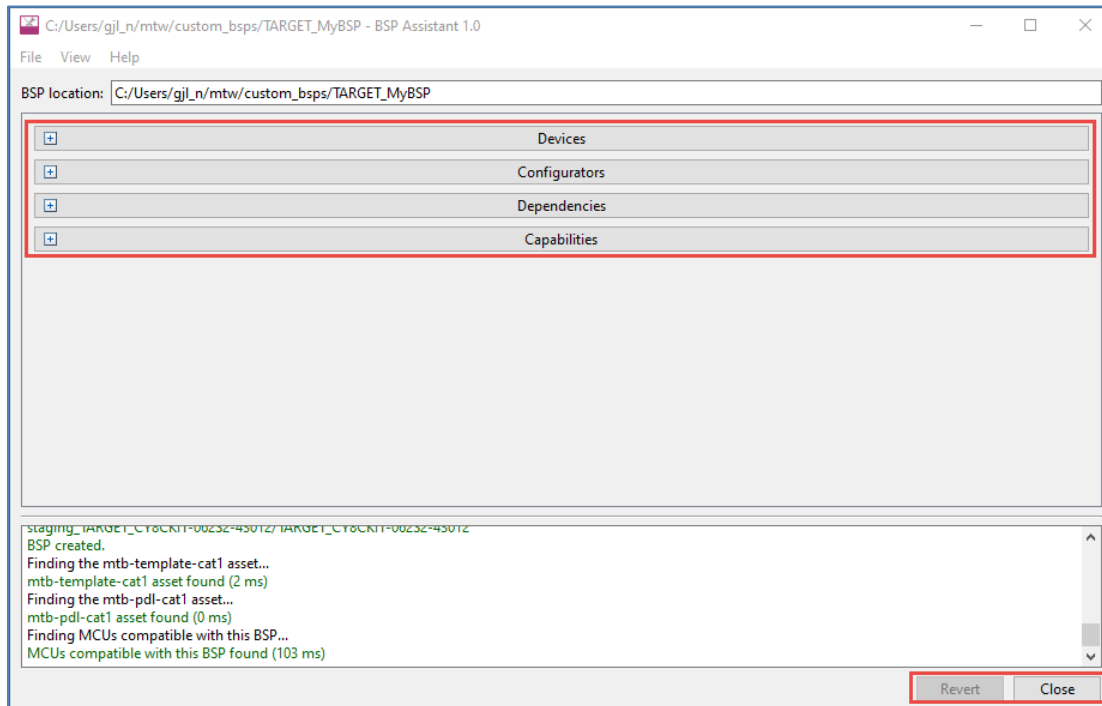


The filter box and buttons to expand/collapse all of the categories make it easier to find what you are looking for.

Specify where the BSP will be saved, and enter what it will be named.

2.9.1.4 GUI description

Whether you create a new BSP from a template or open an existing BSP, the main BSP Assistant window looks like the following image. There are four sections: Devices, Configurators, Dependencies, and Capabilities. Each of these can be collapsed or expanded (they are all shown collapsed).



Devices

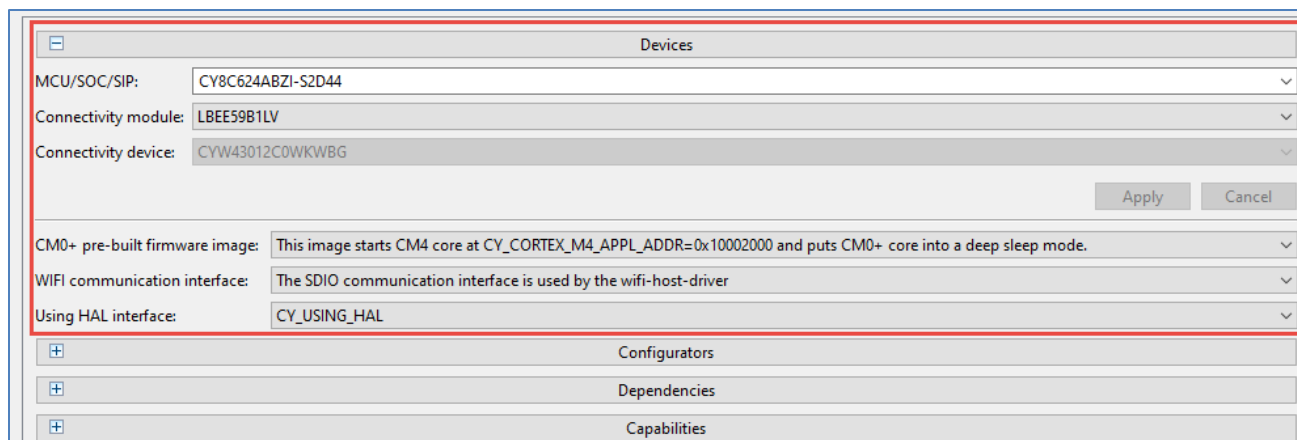
The Devices section has selections for the MCU/SOC/SIP and connectivity module or device on the board. The MCU/SOC/SIP can be changed to any device within the same family. If you want a device in a different family, you must first create a BSP for a device in that family.

The connectivity module can be set to "None," any module that is supported for the selected MCU/SOC/SIP, or "Custom." In the case of "Custom," you must then specify the exact connectivity device that you are using and you must supply the required Bluetooth® and/or Wi-Fi firmware files for your device and hardware. If you choose a supported module, the appropriate files will be automatically provided.

The remaining rows in the Devices section are options that are specific to the MCU/SOC/SIP and connectivity selections that you have made. These are used to select the settings required to get the desired behavior of the devices by setting the `COMPONENTS` variable to the appropriate string in `bsp.mk`.

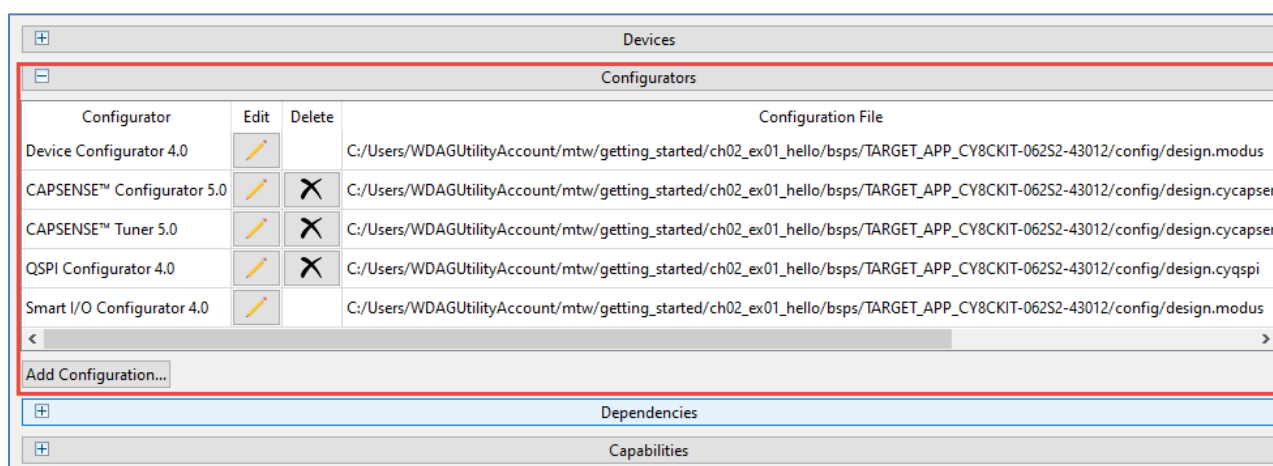
For example, "CM0+ pre-built firmware image" is used to select the firmware that is loaded onto a PSoC™ 6 MCU for the CM0+ core for a single-core application that only uses the CM4.

For a multi-core application, the CM0+ pre-built firmware image must be disabled so that it can be replaced with the desired multi-core functionality. The CM0+ pre-built firmware image will still show up in the BSP Assistant since it is the default image included by the BSP if it is not disabled by the application. The method used to disable the CM0+ pre-built firmware image is the `DISABLE_COMPONENTS` variable in the CM4's *Makefile*.



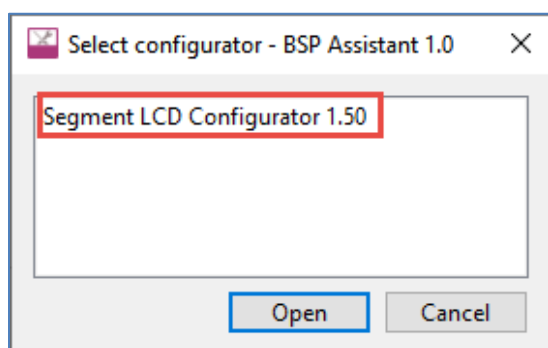
Configurators

The Configurators section allows you to edit, remove, or add BSP configuration settings for the selected MCU/SOC/SIP and connectivity device. The list shows all configurators that currently have files in the BSP. You can click the **Edit** button to open the associated configurator to make changes or click the **Delete** button to remove a configurator file (some configurators are required and cannot be removed).

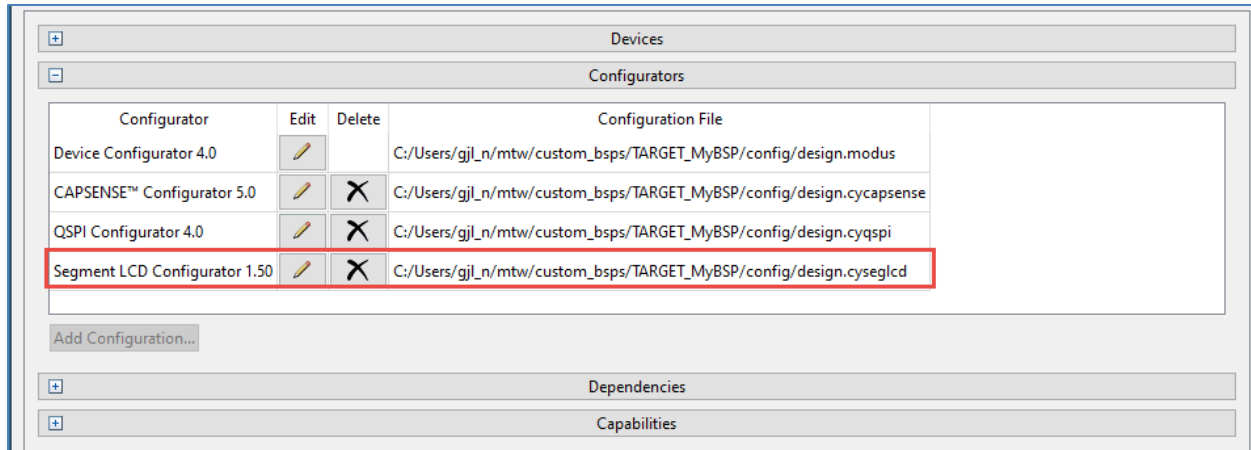


Configurator	Edit	Delete	Configuration File
Device Configurator 4.0			C:/Users/WDAGUtilityAccount/mtw/getting_started/ch02_ex01_hello/bsps/TARGET_APP_CY8CKIT-062S2-43012/config/design.modus
CAPSENSE™ Configurator 5.0			C:/Users/WDAGUtilityAccount/mtw/getting_started/ch02_ex01_hello/bsps/TARGET_APP_CY8CKIT-062S2-43012/config/design.cycapser
CAPSENSE™ Tuner 5.0			C:/Users/WDAGUtilityAccount/mtw/getting_started/ch02_ex01_hello/bsps/TARGET_APP_CY8CKIT-062S2-43012/config/design.cycapser
QSPI Configurator 4.0			C:/Users/WDAGUtilityAccount/mtw/getting_started/ch02_ex01_hello/bsps/TARGET_APP_CY8CKIT-062S2-43012/config/design.cyqspi
Smart I/O Configurator 4.0			C:/Users/WDAGUtilityAccount/mtw/getting_started/ch02_ex01_hello/bsps/TARGET_APP_CY8CKIT-062S2-43012/config/design.modus

If you click the **Add Configuration...** button, a list displays showing any available configurators not yet present for the selected device.



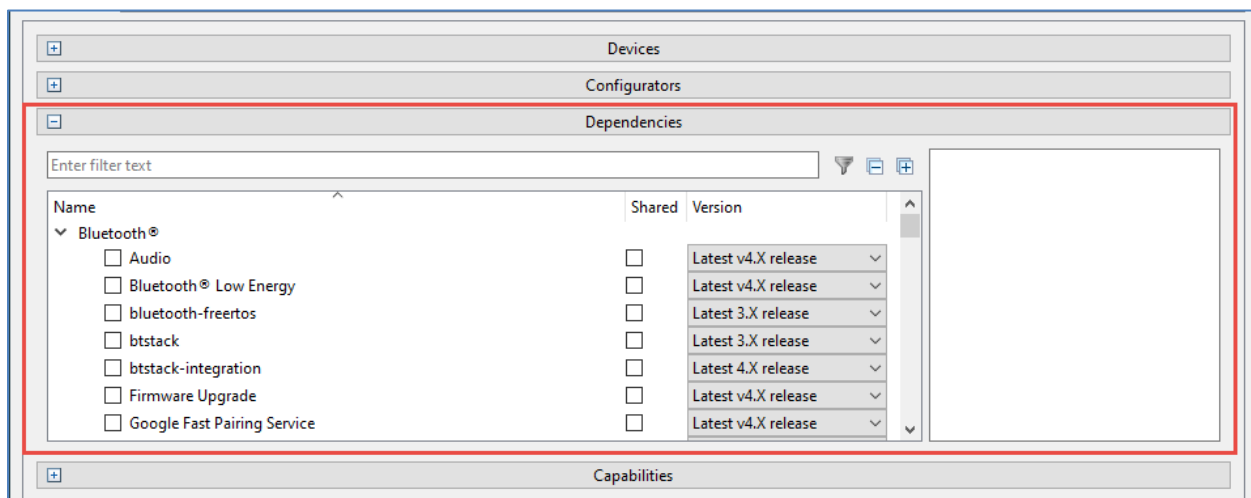
When you click **Open**, the associated configurator opens. Once you save changes from that configurator, it will appear in the BSP Assistant list.



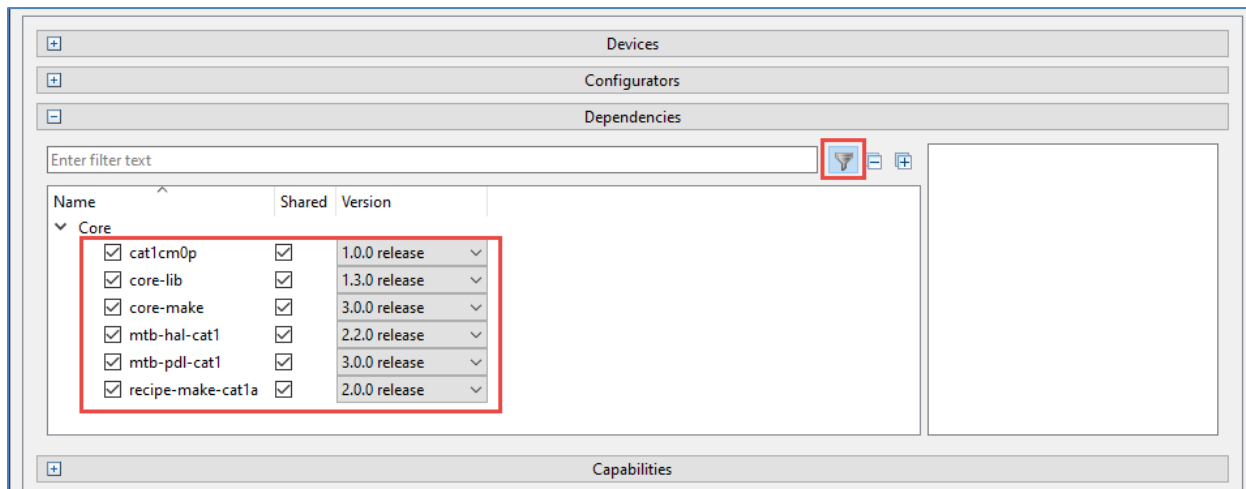
Note: The BSP assistant only operates on BSP configurator files. Library configurators such as the Bluetooth® configurator store their files in the application rather than the BSP, so they are not relevant to the BSP Assistant.

Dependencies

The Dependencies section shows all of the libraries that the BSP needs to operate along with any other libraries that can be added. When you first create a BSP, it will list all of the libraries that are required for your BSP, such as the core libraries, recipe libraries, PDL, and HAL. You can change the version of any dependency and the location where it will be placed for any applications that use the BSP. The location is either shared (i.e. in *mtb_shared*) or local (i.e. in the application's *libs* directory).



Select the **Filter** button to show only those libraries that are currently listed as dependencies. You can also use the filter box to search for libraries or the plus and minus buttons to expand or collapse all categories.



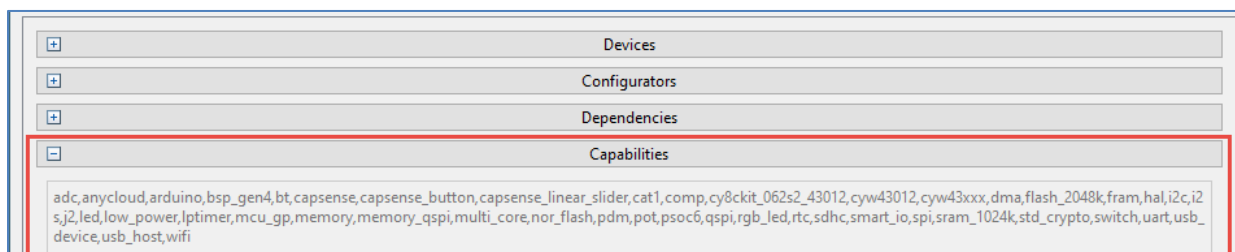
Note: The default versions for the required libraries are the latest release versions at the time the BSP was created. If you want the libraries to be set to the latest versions when the BSP is used to create an application, you can change the versions to "Latest V.X release" where "V" is the major version number.

Note: When you add additional dependencies, the default is "Latest V.X release" where "V" is the major version number. This means that those libraries will be set to the latest versions when the BSP is used to create an application. If you want fixed versions in the BSP, you can use the drop-down to select the release version that you require.

Note: If you modify the dependencies in a BSP, any application(s) that use the BSP must be updated to include those dependencies. This can be done by running the Library Manager and clicking **Update** or by running `make getlibs` from the application's root directory on the command line.

Capabilities

The Capabilities section is a read-only list of the capabilities that the BSP provides.



This list is used by the Project Creator to show template applications (i.e. code examples) that are compatible with your BSP and by the Library Manager to show libraries that are compatible with your BSP.

Specifically, the BSP contains "provided capabilities" while code examples and libraries contain "required capabilities". Project Creator and Library Manager match the contents of the provided and required

capabilities lists to determine what to show. A code example or library that requires a capability will only be shown if the BSP provides that capability.

For example, a code example that uses CAPSENSE™ sensing will only appear in the list of template applications if the BSP provides the `capsense` capability.

If you want to modify the provided capabilities in a BSP, you can edit the `props.json` file from the BSP. The capabilities are in the section `opt > props > capabilities`. For example:

```
"opt": {  
  "props": {  
    "capabilities": [  
      "adc",  
      "anyccloud",  
      "arduino",  
      ...  
    ]  
  }  
}
```

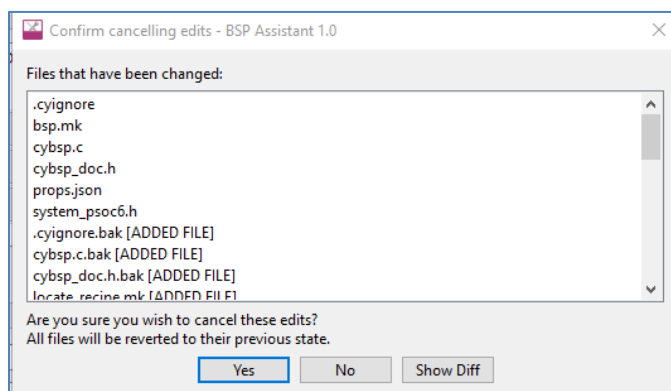
Save or revert changes

The BSP Assistant provides **Close** and **Revert** buttons, described as follows:

- **Close** – The BSP Assistant saves changes made to the Configurators and Dependencies sections automatically as you go. To save changes in the Devices section, you must click **Apply** in that section. If you click **Close** with changes already saved/applied, the BSP Assistant tool will quit. If there are unsaved changes, a dialog displays asking if you want to apply the changes.
- **Revert** – If you decide to back out all of your changes before closing the BSP Assistant (including applied changes in the Devices section), click the **Revert** button to review files that were changed and confirm the cancellation of the edits.

Confirm Cancelling Edits

When you click **Revert**, a dialog displays a list of all the files that have been modified since you opened the tool for this session or created the current BSP. You can view the differences in any of the files before accepting or cancelling the revert operation.



If you click **Yes**, the BSP will be restored to the state when you opened the tool for this session or created the current BSP. You cannot undo a revert operation.

Note: You can see the list of changed files at any time using the menu item **View > List Changed Files...**

2.9.1.5 Updating launch settings

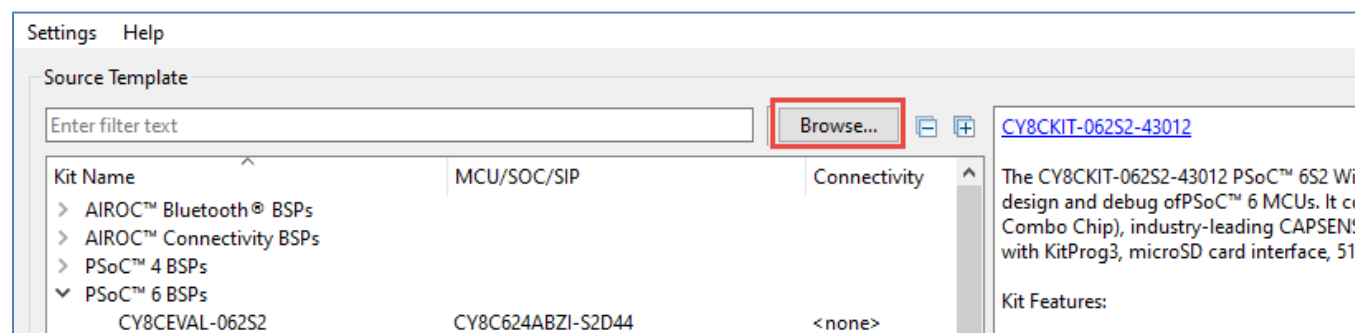
If you are using an IDE, you may need to regenerate the launch settings to reflect the new BSP. Use the appropriate command(s) for the IDE(s) that are being used. For example: `make vscode` for VS Code or click the **Generate Launches...** link in the Quick Panel for Eclipse. This step must be done whenever the target device is changed since the launch settings may change from device to device.

2.9.2 Using a custom BSP in a new application

When using a custom BSP that you created in a new application, you have a few options. Each option is discussed below.

2.9.2.1 Using a local BSP

The easiest way to include a custom BSP in a new application is to simply use the **Browse** button when selecting the BSP for the application in the project creator tool. Just browse to the directory containing your BSP and select it. That BSP will be copied into the application's `bsps` directory.



2.9.2.2 Using a custom manifest for the BSP

As stated above, the browse mechanism is by far the simplest way to include a custom BSP in an application, but you can also get your custom BSP to appear in the list of kit names in the Project Creator and Library Manager tools. For this method to work, the custom BSP must be in a Git repo. You must then create a custom super-manifest file and a manifest file so that the Project Creator and Library Manager tools will know how to find the Git repo. The manifest file includes not only the location of the BSP's Git repo, but also the list of dependencies for the BSP. See section 2.13 for additional details.

2.9.2.3 Using a version-controlled copy of the BSP

If your BSP is in a version control system (e.g. Git, Subversion, etc.), you can place a controlled copy of the BSP inside the application. You must first still create the application with a local BSP. That is, create the application and browse to the custom BSP as described above. Once the application is created, you can delete the local copy of the BSP from the application and then use Git clone or any other mechanism supported by your version control system to create a controlled version of the BSP in the application.

Note: Remember that if the BSP is a version-controlled copy, any edits (such as changes using the Device Configurator) will affect that version-controlled copy.

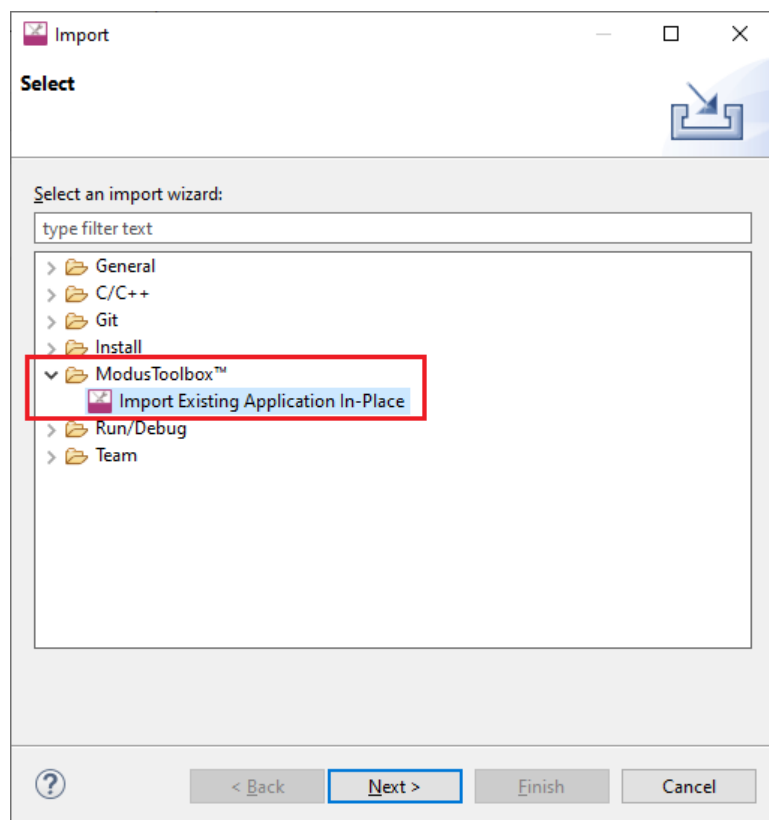
2.11 Importing/exporting applications for IDEs

Once you have an application, 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 an Eclipse Workspace 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 (where `<IDE>` depends on the IDE you are using). 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.

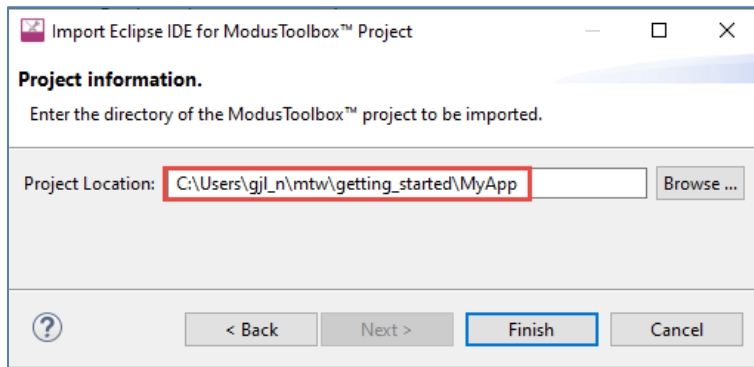
2.11.1 Import projects into the Eclipse IDE

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™ > Import Existing Application In-Place**. There is also a direct link for this in the Quick Panel with the same label.



Note: This imports the application so that the Eclipse workspace can see it but it does NOT copy it into your workspace directory. If you want the resulting application to be in your workspace, make sure you put it in the workspace directory 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™.

2.11.1.1 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.

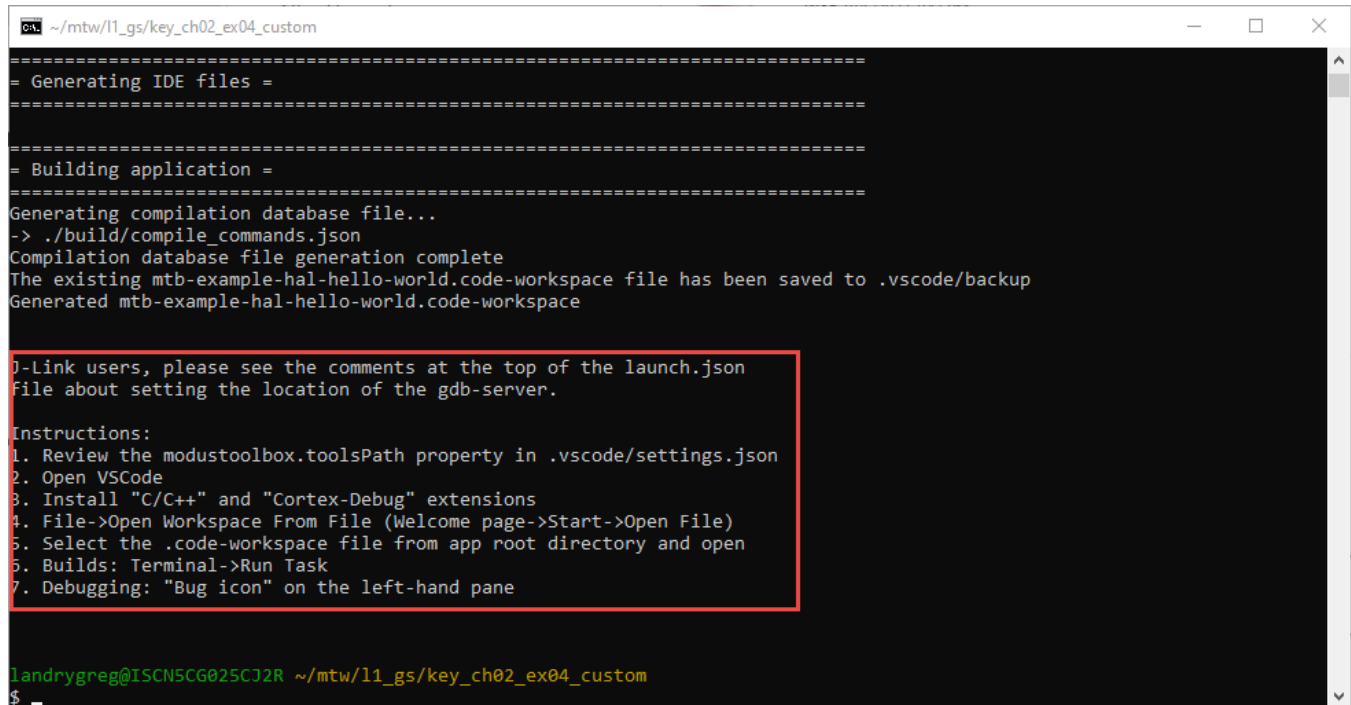
Some of the cases where you may end up with stale or multiple sets of launch configs:

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

2.11.2 Export for Microsoft Visual Studio Code (VS Code)

VS Code is quickly becoming a favorite editor for developers. The ModusToolbox™ ecosystem provides a mechanism to generate all the files required for VS Code to edit, build, and program a ModusToolbox™ application. If you didn't select Microsoft Visual Studio Code in the Project Creator tool when you created the application, 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
```



```
~/mtw/l1_gs/key_ch02_ex04_custom
=====
= Generating IDE files =
=====
= Building application =
=====
Generating compilation database file...
-> ./build/compile_commands.json
Compilation database file generation complete
The existing mtb-example-hal-hello-world.code-workspace file has been saved to .vscode/backup
Generated mtb-example-hal-hello-world.code-workspace

J-Link users, please see the comments at the top of the launch.json
file about setting the location of the gdb-server.

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

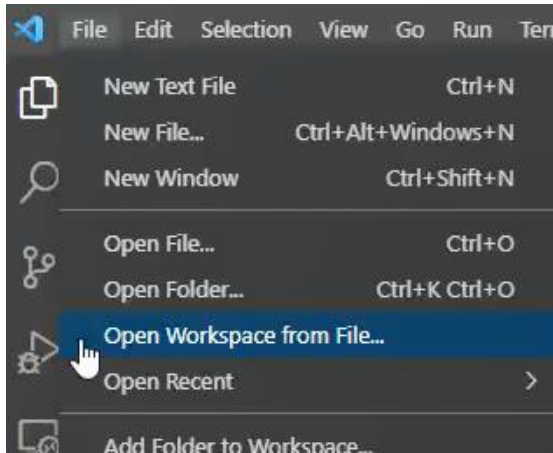
landrygreg@ISCN5CG025CJ2R ~/mtw/l1_gs/key_ch02_ex04_custom
$
```

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

J-Link users, please see the comments at the top of the launch.json file about setting the location of the gdb-server.

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

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



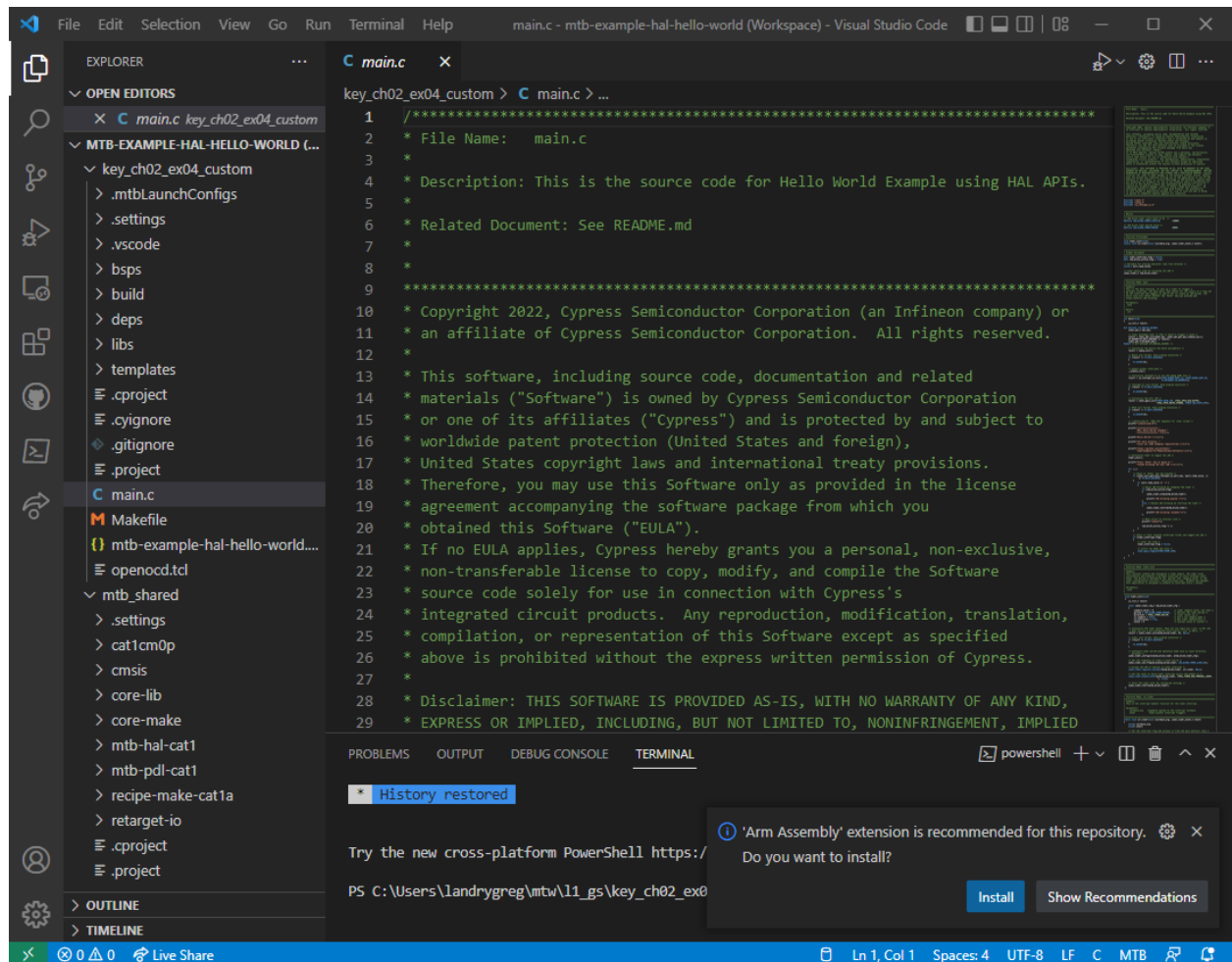
Note: We'll cover VSCode workspaces in more detail in a minute.

5. Then, navigate to the directory where your project resides, select the file with the extension `.code-workspace` and click **Open**.

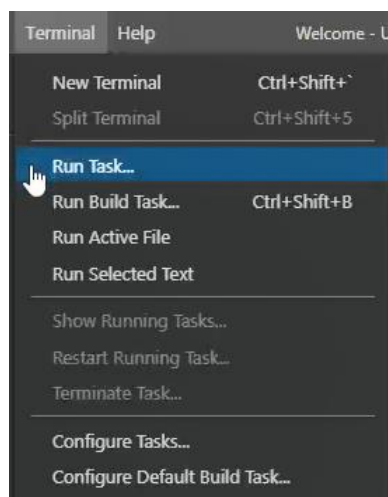
Note: Alternately, if it is in your path you can just enter `code <name>.code-workspace` on the command line after running `make vscode` and it will open VS Code and load the workspace all in one step. (Depending on how you installed VS Code, it may or may not be in your path by default.)

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

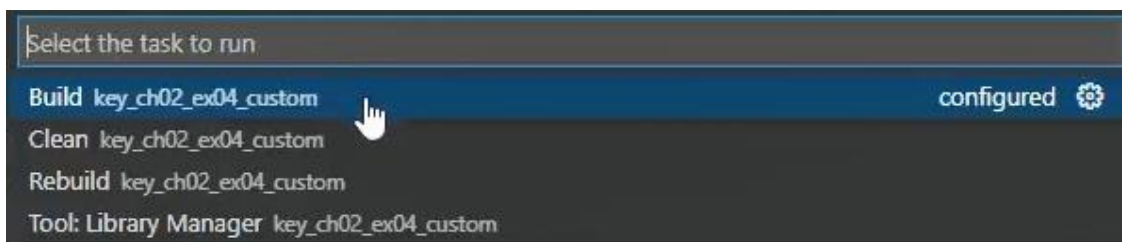
Now your windows should look something like this. You can open the files by clicking on them in the Explorer window.



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

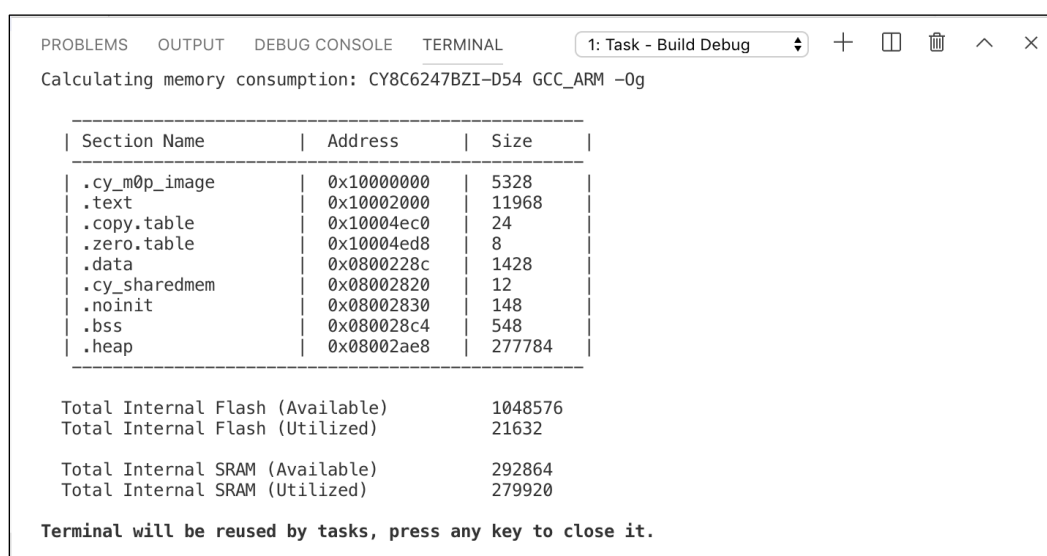


Then, select **Build <appname>**. This will essentially run `make build` at the top level of your project.



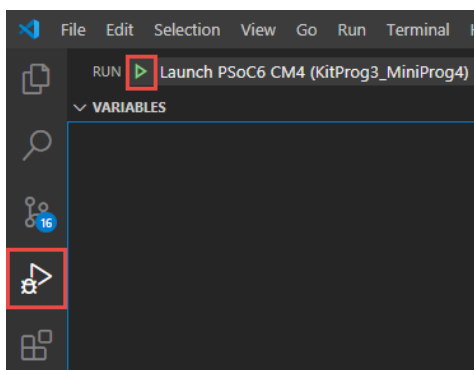
Note: You can also access the Library Manager from the Task list.

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

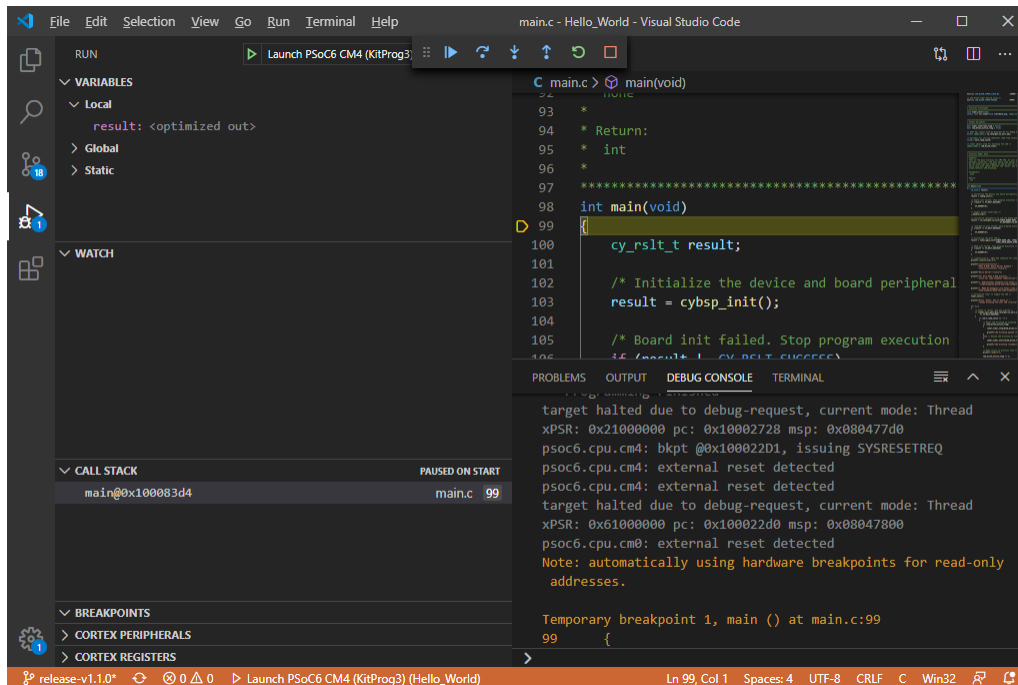


- To build and then program the kit or run the debugger, first make sure you have installed the "Cortex-Debug" VS Code extension from the marketplace. See the software installation exercise in the introduction chapter if you didn't do this when you installed VS Code.

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). Once you make a selection from the drop-down, click the green **Play** button to start the operation.



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>**.

2.11.2.1 Using workspaces in VS Code

Rather than just opening the application's directory in VS Code, we have been opening a workspace. The reason is that the workspace knows that the code in *mtb_shared* is part of the application and should be shown in the file Explorer window and used for Intellisense operations.

*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.*

In addition to the default workspace that is created by `make vscode`, you can create your own workspaces as well depending on what you want to see.

If you want to see a single application along with the shared directory (e.g. application + *mtb_shared*) you can use the default workspace that is created during `make vscode` as we have done up to this point.

If you want a workspace that will show all applications in a given directory instead of just one application at a time, 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...**
- If you add a library to the application, it is a good idea to build the application before adding any code for the new library. This will allow Intellisense to find code in the new library.

2.11.3 Export for IAR Embedded Workbench

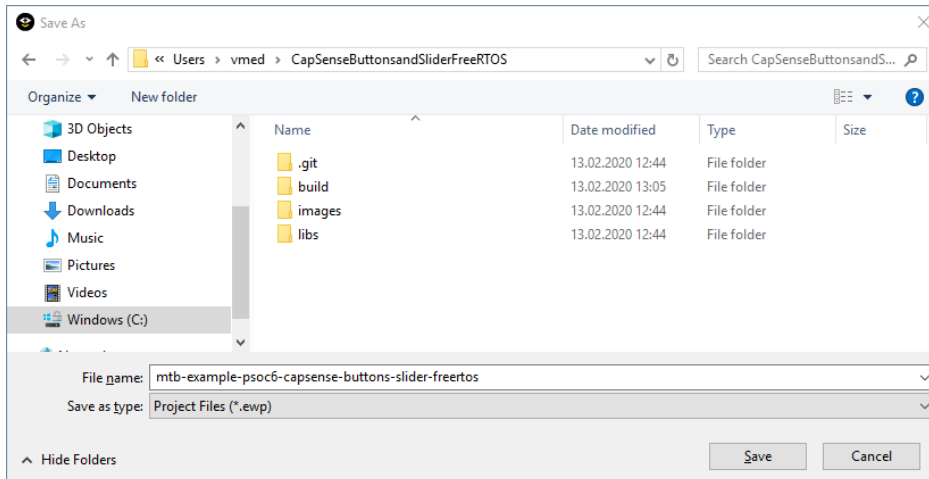
The ModusToolbox™ build system provides a make target for IAR Embedded Workbench. If you didn't generate the files during project creation you can run the following command:

```
make ewarm8 TOOLCHAIN=IAR
```

Note: *The command above will set the TOOLCHAIN to IAR in the Embedded Workbench configuration files but not in the application's Makefile. Therefore, builds inside IAR Embedded Workbench will use the IAR toolchain while builds from the command line will continue to use the toolchain that was previously specified in the Makefile. You can edit the Makefile's TOOLCHAIN variable if you also want command line builds to use the IAR toolchain. However, if you make any changes to the settings inside the IDE that affect the build, they will not be reflected in command line builds. The converse is also true – changes in the Makefile will not affect the IDE builds.*

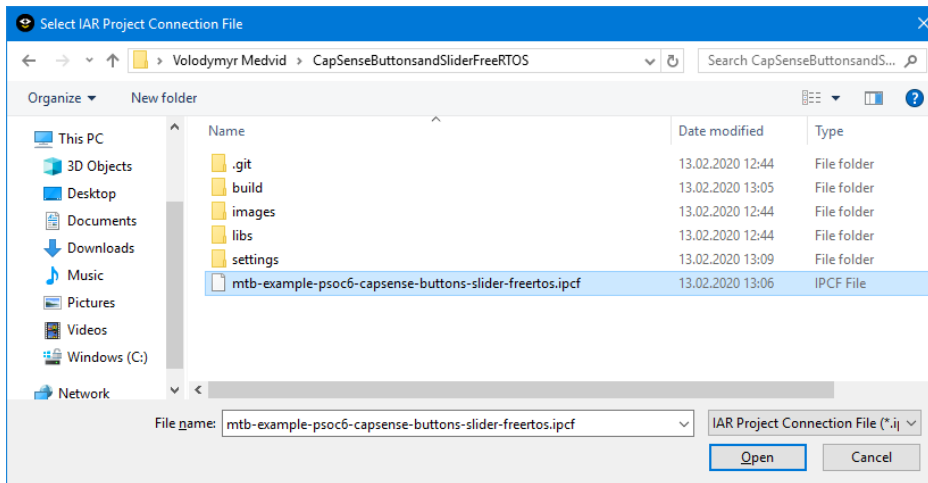
An IAR connection file appears in the application directory:

- `<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™ tools package user guide which can be found at www.infineon.com/ModusToolboxUserGuide.

2.11.4 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:

```
make uvision5 TOOLCHAIN=ARM
```

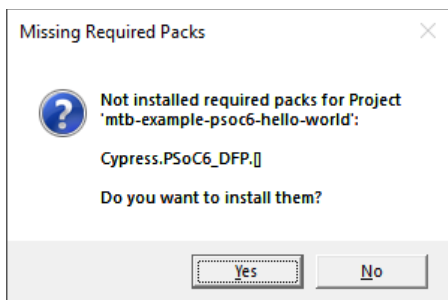
Note: The command above will set the TOOLCHAIN to ARM in the µVision configuration files but not in the application's Makefile. Therefore, builds inside Keil µVision will use the ARM toolchain while builds from the command line will continue to use the toolchain that was previously specified in the Makefile. You can edit the Makefile's TOOLCHAIN variable if you also want command line builds to use the ARM toolchain. However, if you make any changes to the settings inside the IDE that affect the build, they will not be reflected in command line builds. The converse is also true – changes in the Makefile will not affect the IDE builds.

This generates three files in the application directory:

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

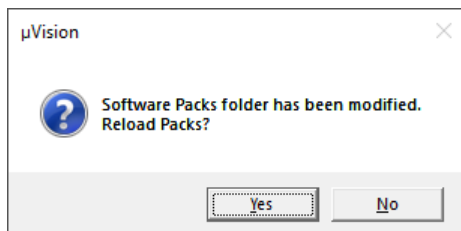
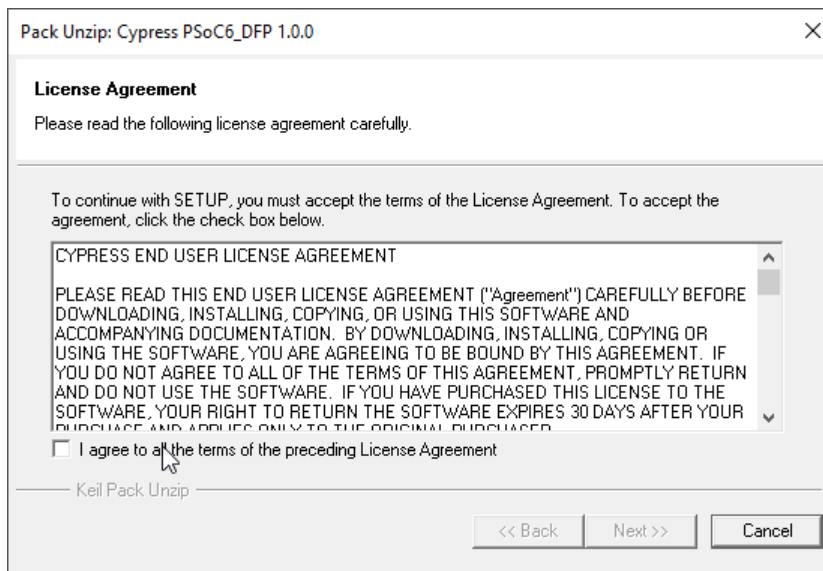
The *cpdsc* file extension should have the association enabled to open it in Keil µVision.

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™ tools package user guide which can be found at www.infineon.com/ModusToolboxUserGuide.

2.12 Version control and sharing applications

If you are working on a design with more than one person, it is common to want to share an application using some type of version control system such as Git. Even if you are not using version control, you may want to share applications manually copying files or using your IDE's export/import mechanisms.

2.12.1 Files to include/exclude

No matter which method you choose, you will want to know what is critical to check in to version control or copy, and what can be regenerated easily. The main files to consider when sharing or copying include anything that you have changed or added, and that will not be regenerated. These files include source code, BSPs and configurations, *Makefile*, etc. There are several directories in the application that can be re-created and therefore do not need to be copied or checked into version control. These include the *libs*, *mtb_shared*, *build*, and *GeneratedSource* directories. The processes that create them are:

libs and *mtb_shared*: Created by running `make getlibs` or by running the Library Manager and clicking the **Update** button. Either one will clone the libraries from GitHub to the appropriate locations.

build: Created during the build process.

GeneratedSource: Generated by running the associated configurator such as the Device Configurator or Bluetooth® configurator. The configurators are run automatically by the build process if the *GeneratedSource* files are out of date.

2.12.2 Using version control software

If you are working on a production design, you probably use version control software to manage the design and any potential revisions. This allows all users to stay synchronized with the latest version of an application. ModusToolbox™ assets are provided using Git, but you can use any version control method or software that you prefer.

ModusToolbox™ code examples have a default *.gitignore* file that excludes the directories that can be easily recreated and files containing IDE-specific information that need not be checked in. If you are using Git as your version control software, that file can often be used as-is, but you are free to change it if you want to change what gets checked in. For example, you may decide that you want to check in all of the libraries from *libs* and *mtb_shared* even though they are available on Infineon's GitHub site.

If you are using different version control software, the *.gitignore* file can be used as a guide for configuring the software that you are using.

Once you have an application checked into your desired version control software, sharing the application with a new user is straight-forward. The steps are:

1. Get a copy of the checked-in data. This will vary depending on the version control software (for example, `git clone <url>`).
2. Run the Library Manager and click the **Update** button, or open a terminal and run the command `make getlibs`. Either one will get all of the libraries required by the application.

3. Use the application as usual – the *build* and *GeneratedSource* files will be created automatically when needed.
4. When finished with your changes, check in your updates following your version control process.

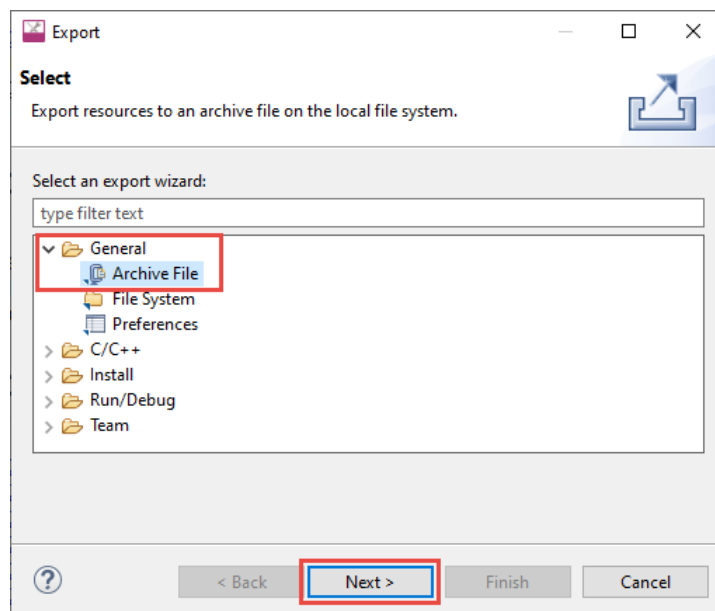
Note: You can import the project into an existing Eclipse workspace using the **Import Existing Application In-Place** link in the Quick Panel or by using **File > Import > ModusToolbox > Import Existing Application In-Place**. This imports the application in-place; it does NOT create a new copy of the application in your workspace so you should move it to your workspace first before importing it.

2.12.3 Manual file copy

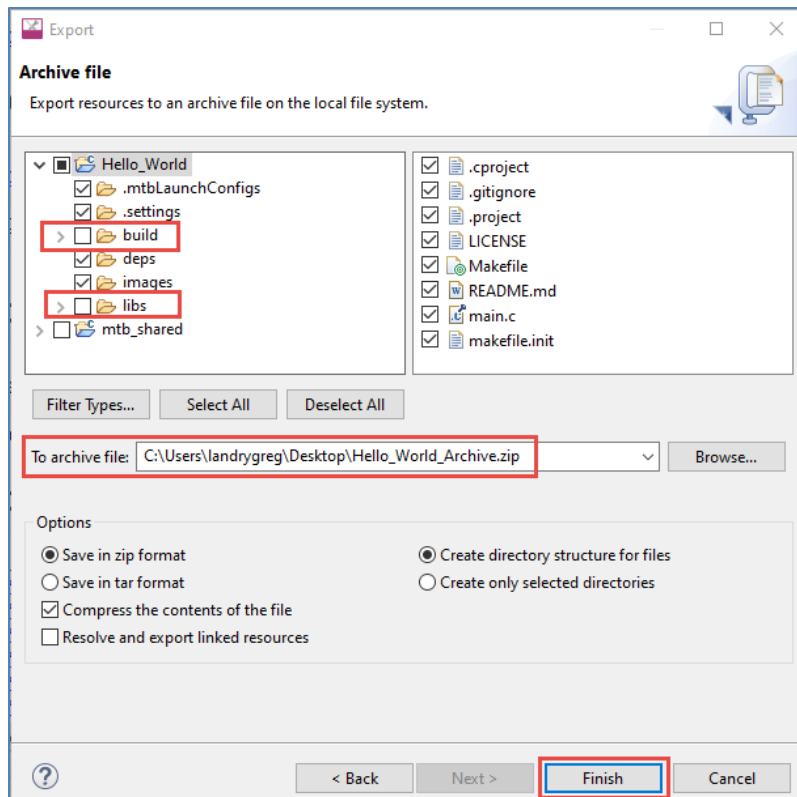
If you are not using version control software, you can just copy a complete application directory from one user to another. If desired, you can exclude the directories listed in 2.12.1 since the libraries can be downloaded by running `make getlibs` or running the Library Manager and clicking **Update**. The other files are regenerated when the application is built.

2.12.4 Using Eclipse archive files

If you are using the Eclipse IDE for ModusToolbox™, it has an archive feature that allows you to create a zip file that can be given to another user. To do this, use **File > Export > General > Archive File** and click **Next**.



When you get to the next page, you can select just the application that you want to archive along with the files required. That is, you don't need to archive *mtb_shared*, *libs*, *Generated Source*, or *build* directories.



For multi-core applications, the Eclipse IDE export function shows the project folders for each core alongside the main application folder. These project folders are included in that main application folder, so you should exclude them from the export process as well. On the Export dialog, select only the main application folder(s) to export. You may need to review your application in Eclipse to determine which folders you should exclude. By default, project folders include an extension after the application name.

Once you have the correct file selected, click **Finish** to generate the archive file.

The recommended procedure to import an application into the Eclipse IDE for ModusToolbox™ is as follows:

1. Unzip the archive file and place it into the desired Eclipse workspace directory.
2. Either run `make getlibs` or run the Library Manager tool and click **Update**.
3. From the Eclipse IDE for ModusToolbox™ Quick Panel, click on the link for **Import Existing Application In-Place**.
4. Click the **Browse...** button, navigate to the application directory, and click **Finish** to begin the import process. This will take a few moments, and then the application will display in the Eclipse IDE Project Explorer.

2.13 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 an accessible location, 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: The super-manifest and manifest files must be accessible via a URL but. They do NOT need to be stored on a Git server such as GitHub – they can use any URL that points to the xml files. For example:

<https://myserver.com/my-super-manifest.xml>

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

<file:///C:/MyManifests/my-super-manifest.xml>

To see examples of the syntax of super-manifest and manifest files, you can look at the Infineon manifest files on GitHub. The following is a subset of the full set of manifest repos that exist. Each repo may also contain multiple manifest files.

- 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>

2.14 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.

2.14.1 Network proxy settings

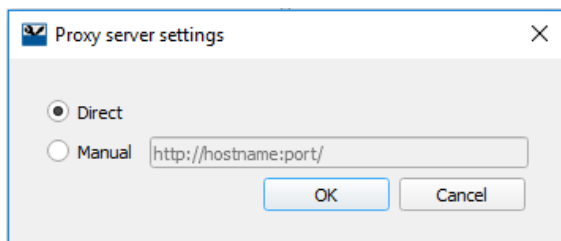
If you have environment variables for *http_proxy* and *https_proxy*, those same variables will work for ModusToolbox™ tools. However, there is a simpler way to enable/disable the proxy settings in the Project Creator and Library Manager tools that doesn't require you to create/delete the environment 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.

2.14.2 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:

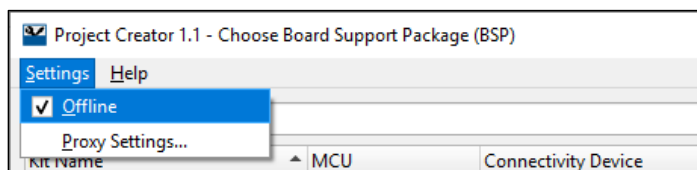
<https://community.infineon.com/t5/Knowledge-Base-Articles/Offloading-the-Manifest-Files-of-ModusToolbox-KBA230953/ta-p/252973>

2.14.3 Offline content

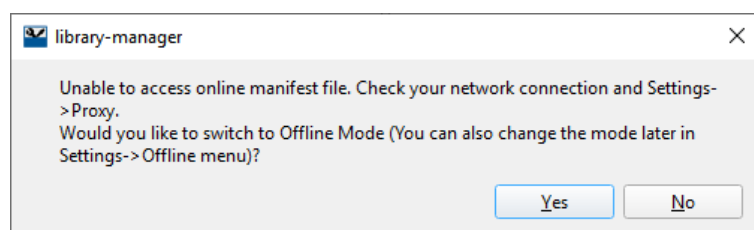
In the case where network access is just not possible, Infineon provides a zipped-up bundle with all of our GitHub repos to allow you to work offline, such as on an airplane or if for some reason you don't have access to GitHub. To set this up, you must have access to [infineon.com](https://community.infineon.com/t5/Resource-Library/ModusToolbox-offline-libraries/ta-p/252265) in order to download the zip file. After that, you can work offline.

Go to <https://community.infineon.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.



2.15 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*:

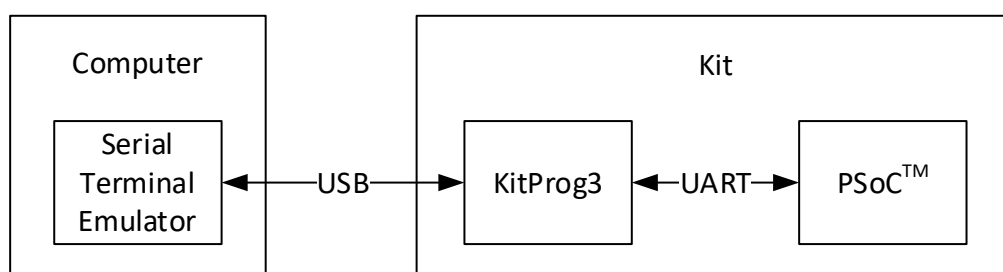
```
ifeq ($(CONFIG),Debug)
CY_TOOLCHAIN_DEBUG_FLAG=-DDEBUG
CY_TOOLCHAIN_OPTIMIZATION=-Og
else ifeq ($(CONFIG),Release)
CY_TOOLCHAIN_DEBUG_FLAG=-DNDEBUG
CY_TOOLCHAIN_OPTIMIZATION=-Os
else
CY_TOOLCHAIN_DEBUG_FLAG=
CY_TOOLCHAIN_OPTIMIZATION=
endif
```

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.

2.16 Serial Terminal Emulator

Most Infineon kits have an integrated programming device (such as KitProg3 on PSoC™ kits) which also have a UART to USB interface. This is very useful for printing messages from the firmware onto a serial terminal window on your computer. For example, with PSoC™ this can be done either using the *retarget-io* library with the standard `C printf` function or using the UART API directly. Either way, the UART signals from the PSoC™ go to the KitProg3 device on the kit. The KitProg3 device converts the UART messages into USB and sends them to the PC over the USB connector. On the PC side, a serial terminal emulator is used to display the messages.

Note: The UART is bi-directional so it is possible to type into the serial terminal emulator on the PC and have those messages be sent to the PSoC™.



There are various serial terminal emulators available for use. On Windows and Linux, a program called *PuTTY* is often used. On MacOS, you can use a program called *screen*. There are many other options available so if you are familiar with a different serial terminal emulator, feel free to use that instead.

If you are using the Eclipse IDE for ModusToolbox™, you can even run a terminal window right from the IDE.

2.16.1 Determine the Port

On Windows you can find the correct serial line in the device manager under **Ports (COM & LPT)** as "KitProg3 USB-UART". It will be something like COM91.

On Linux, issue the following in a command window. Your kit will be something like `/dev/ttyACM0`.

```
ls /dev/tty*
```

On MacOS, issue the following in a command window to look for a USB serial device. Your kit will have "usb" in the name.

```
ls /dev/tty.*
```

2.16.2 PuTTY (Windows and Linux)

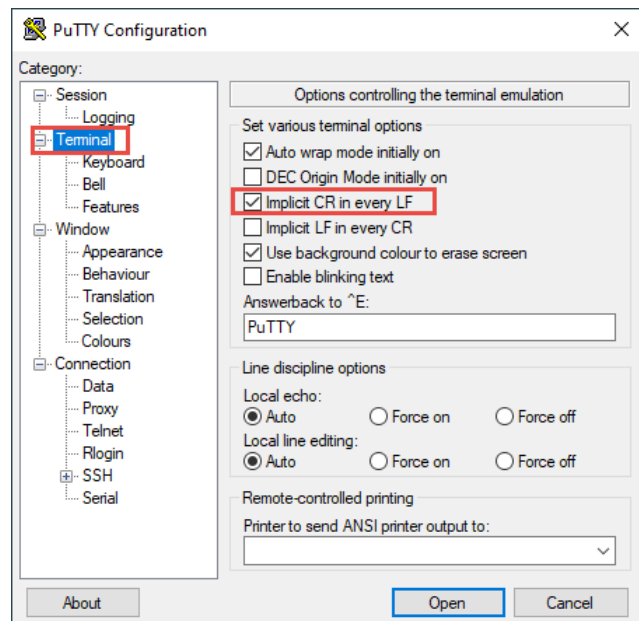
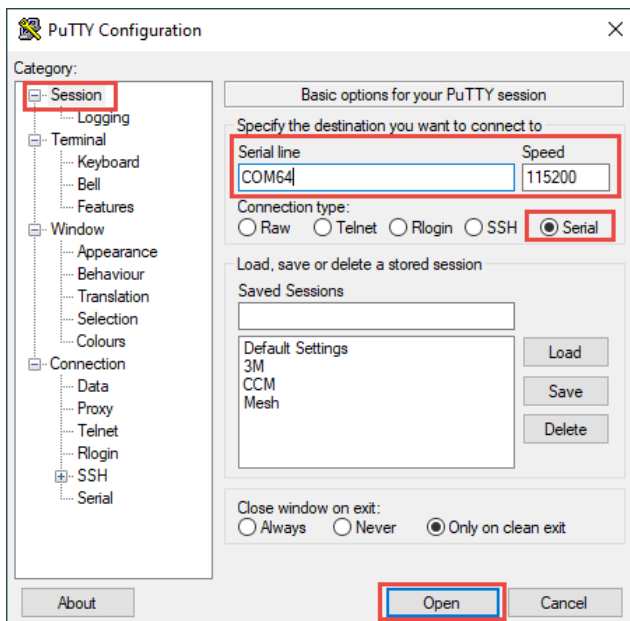
PuTTY for Windows can be found online. Once it is downloaded, you just run the executable `putty.exe` – it does not need to be installed.

For Linux (Ubuntu), PuTTY can be installed by running the following commands from a command window.

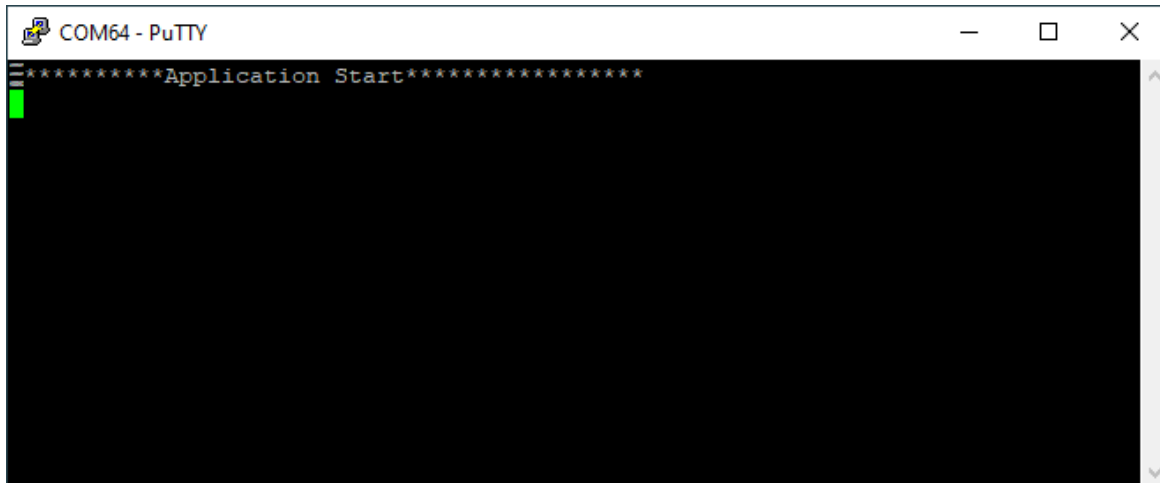
```
sudo apt-get update
sudo apt-get install -y putty
```

When putty starts up, first go to the **Session** settings. Click the "Serial" button if it is not already selected. Set the correct Serial line and set the Speed to 115200. Other settings such as flow control and parity can be found in the **Serial** settings but the defaults should be correct. Finally, in the **Terminal** settings, check the box for "Implicit CR in every LF". This will ensure that printing the `\n` character will send both a line feed and a carriage return. Once you have everything set the way you want it, click Open.

Note: Before opening the terminal, you can use the Save button to save your settings as the default settings or you can save to a new name if you want to keep more than one set.



Once it is running, right-clicking on the top banner shows a list of useful commands. You can change settings, clear the screen, and even restart the session if the kit if it was disconnected.



2.16.3 Screen (MacOS)

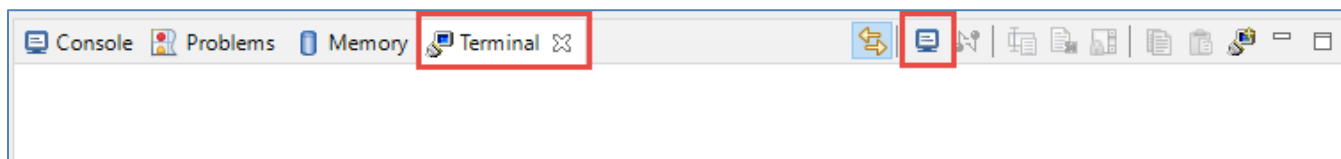
For MacOS, the `screen` program is a good serial terminal emulator.

Once you find your kit, use the following command to start `screen`:

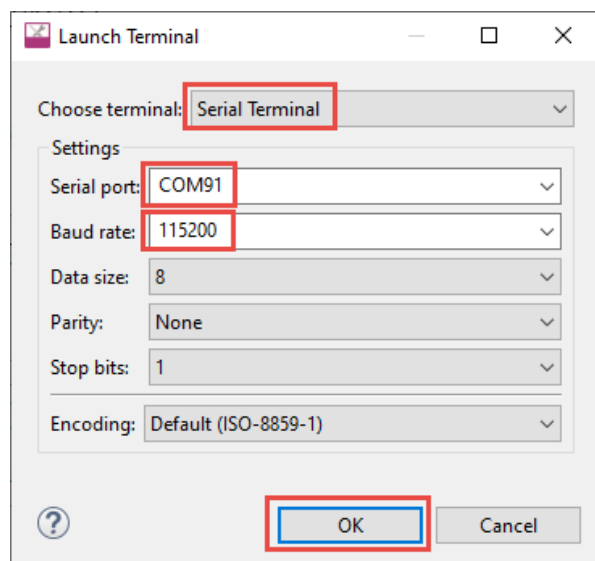
```
screen /dev/tty.<your_device> 115200
```

2.16.4 Eclipse IDE for ModusToolbox™ built-in terminal

If you want to use the serial terminal capability in ModusToolbox™, first select to the **Terminal** tab in the bottom panel and click the button to open a new terminal.



In the window that appears, select **Serial Terminal**, then select the correct port, and other settings. Finally, click **OK**.



Once you start it, the output will be shown in the Terminal tab in ModusToolbox. Click the **X** on the tab for the serial terminal to close it (Not the Terminal tab itself – if you accidentally close the top-level terminal tab, you can use **Window > Perspective > Reset Perspective** to reset the window layout).

2.17 Version 2.x BSPs/applications versus 3.x BSPs/applications

At the time of the ModusToolbox™ 3.0 tools package release, some code examples still create ModusToolbox™ 2.x format BSPs and applications. These 2.x applications, as well as any you created using ModusToolbox™ versions 2.2 through 2.4, fully function in the 3.x ecosystem. The following table highlights a few key differences between 2.x BSPs/applications and 3.x BSPs/applications:

Item	Version 2.x	Version 3.x
BSP Assistant usage	Not applicable	Creates and updates 3.x BSPs
Default BSP type	Git repo, to make changes requires custom BSP	Application-owned, can be directly modified
Local BSP location	Under the <i>libs</i> directory	Under the <i>bsps</i> directory
<i>design.modus</i> file location	<i>libs/COMPONENT_BSP_DESIGN_MODUS</i> subdirectory	<i>bsps/config</i> subdirectory
<i>Makefile</i> MTB_TYPE variable	Not applicable	Identifies single-core vs. multi-core applications
Custom device configuration for an application	Custom files are in the application directory <i>COMPONENT_CUSTOM_DESIGN_MODUS</i> and are compiled into the application via <i>Makefile</i> COMPONENT settings.	Custom files are in the application directory <i>templates</i> and are copied into the BSP during application creation.

This training focuses on the 3.x application structure. For more details about 2.x applications and BSPs, refer to an older version of this training material. All versions are available on the training GitHub repo as branches.

To take full advantage of the newest features, you can easily migrate version 2.x applications to the 3.x structure following Knowledge Base Article KBA236134. This KBA provides instructions for replacing your BSP and associated libraries with compatible versions for 3.x.

Note: You cannot mix and match version 2.x format applications with 3.x format BSPs, or vice-versa.

2.18 Exercises

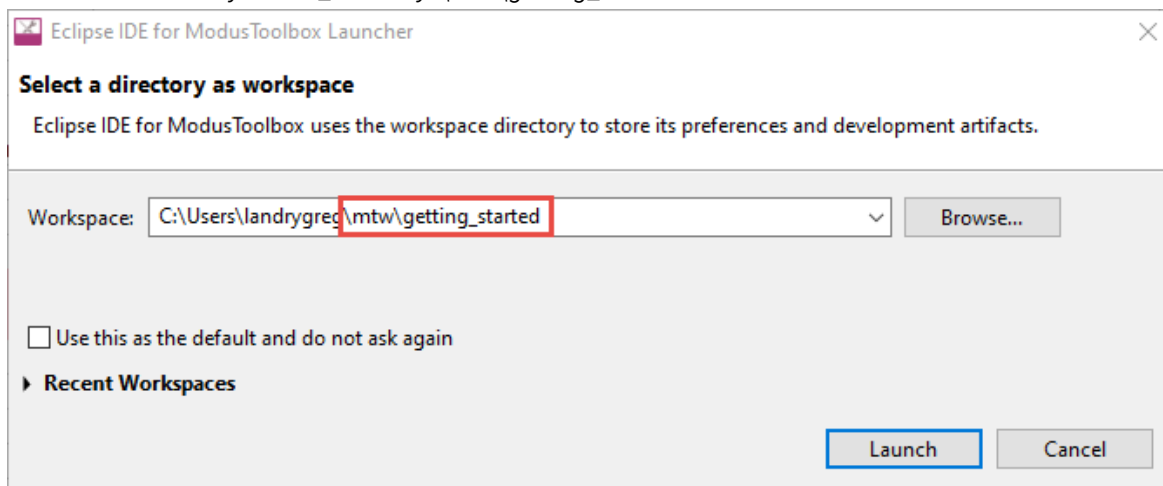
In the following exercises, the CY8CKIT-062S2-43012 kit is used but feel free to use any available BSP to explore the family or solution of your choice. The exact steps and starting application that you use may be different than what's shown here, but the concepts will be the same.

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.

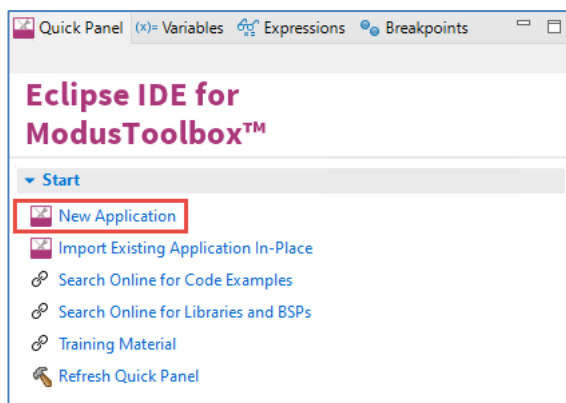
- ☐ 1. Start the Eclipse IDE for ModusToolbox™.

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`.

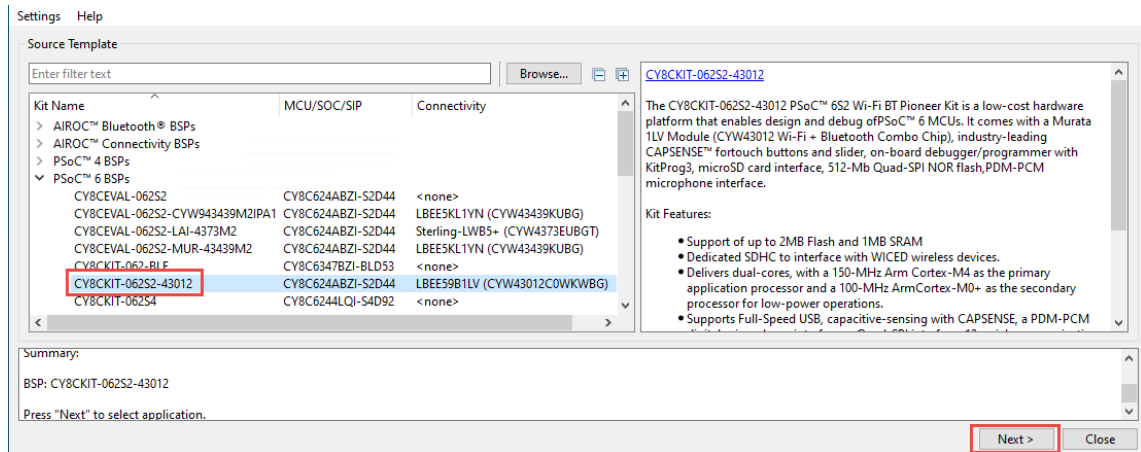


You can change workspaces at any time and you can have as many workspaces as you want.

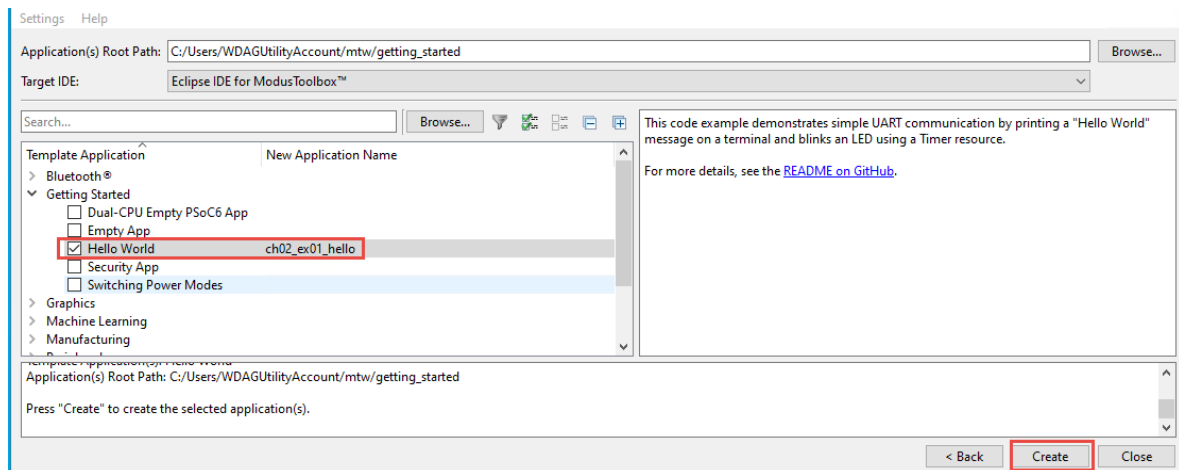
- ☐ 2. On the Quick Panel, click the **New Application** link or select either **File > New > ModusToolbox™ Application**. Either option launches the Project Creator tool.



- ☐ 3. 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.



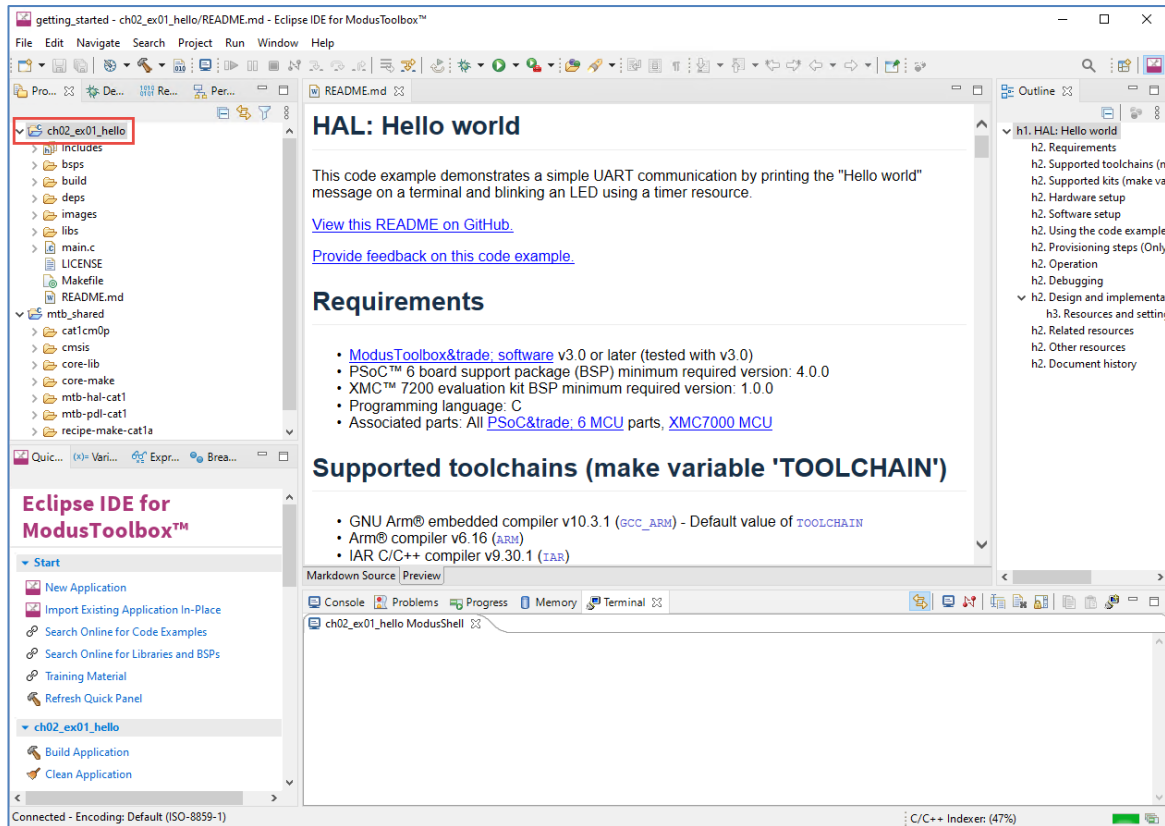
4. Check the box next to *Hello World* from the *Getting Started* category and give your project a name (in this case I'll use **ch02_ex01_hello**). Then, click **Create**.



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.

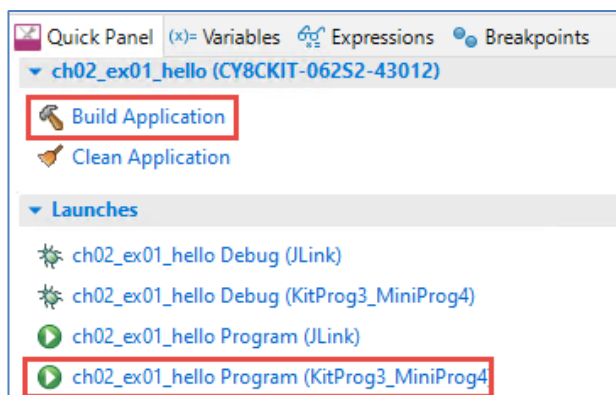
Once the application has been created, the new project creation window will close (unless there are warnings or errors) and the new application is imported into the Eclipse IDE.



- ☐ 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. You can click the build link in the quick panel to build the project without programming but in this case just click the appropriate program link to build first 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.

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.

- ☐ 8. Open a serial terminal emulator window and then reset the kit (press the black reset button) to see the messages. Follow the instructions on the screen to stop/start the LED blinking.

Note: See [Serial Terminal Emulator](#) for details on using a serial terminal emulator program to view UART messages.

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



- ☐ 10. 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 VS Code

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

Note: *The instructions have you create a new copy of the application for use with VS Code. If you would rather use your existing application from exercise 1, just run the command `make vscode` from the command line and then skip to step 5.*

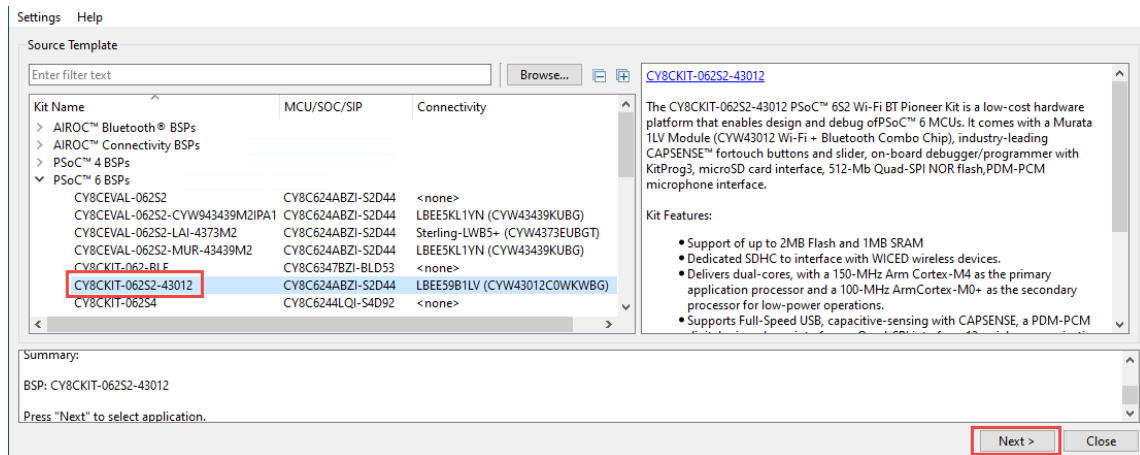


1. Run the Project Creator tool.

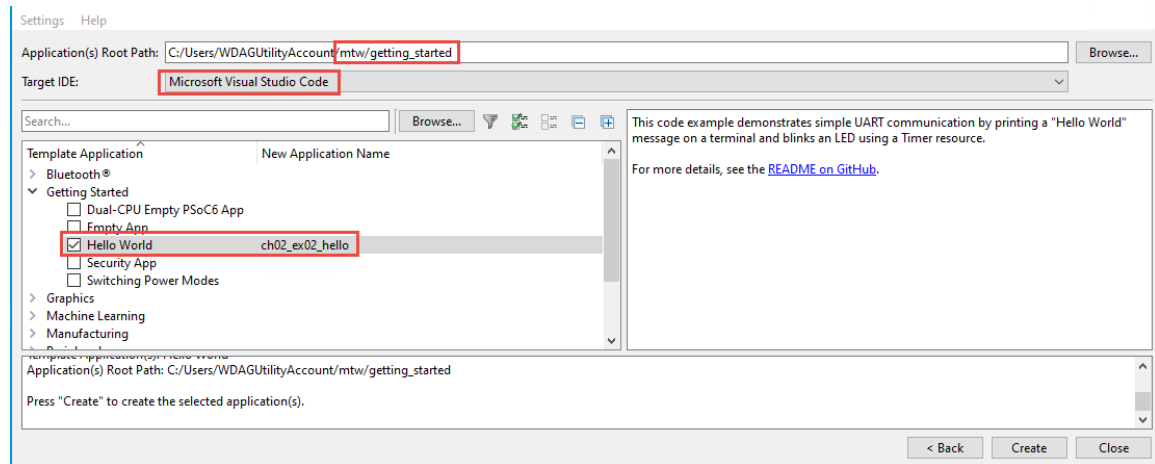
In Windows, you can open the search box and enter `project-creator`. On other operating systems, you can use a terminal to go to the directory `<install_path>/ModusToolbox/tools_<version>/project_creator` and run `project_creator.exe`.



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

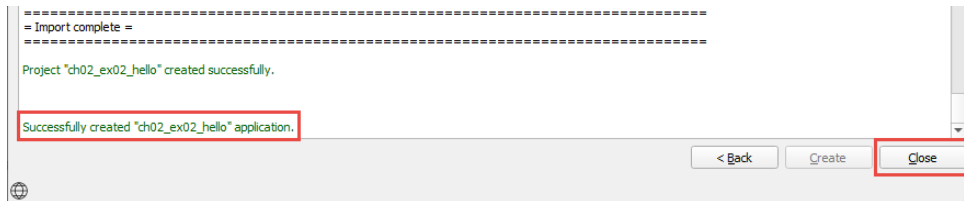


3. On the Application page, select the *Hello World* application and change the name to **ch02_ex02_hello**. Change the **Target IDE** to Microsoft Visual Studio Code. I'll leave the Application root path set as it is so that it will go in the same workspace as the previous exercise but you can change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



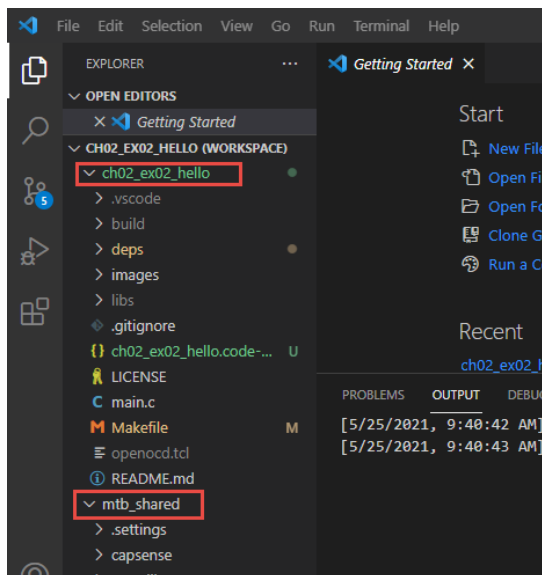
Note: *If you forget to select the **Target IDE** when you create the application, remember that you can run `make vscode` from the command line once application creation is done.*

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

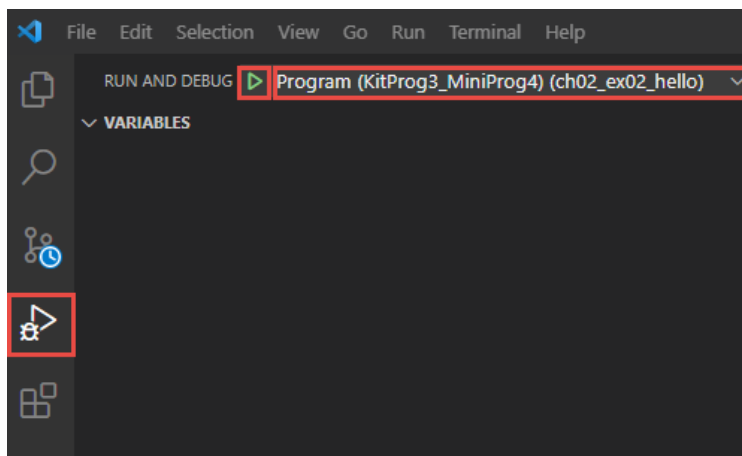


- ☐ 5. Open the application in VS Code and open the workspace. If you open it correctly, you will see the application directory and the *mtb_shared* directory.

See if you need a reminder.



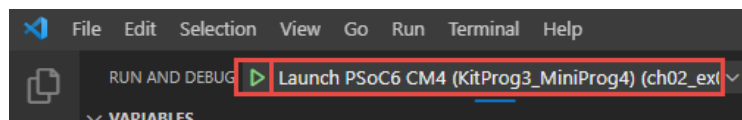
- ☐ 11. Right-click on the *README.md* file and select **Open Preview** to see the formatted Markdown file.
- ☐ 12. Open *main.c* and change `LED_BLINK_TIMER_PERIOD` to a smaller value (e.g. 999) so that the LED will blink faster this time.
- ☐ 13. Click on the Debug icon on the left side, select the 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.

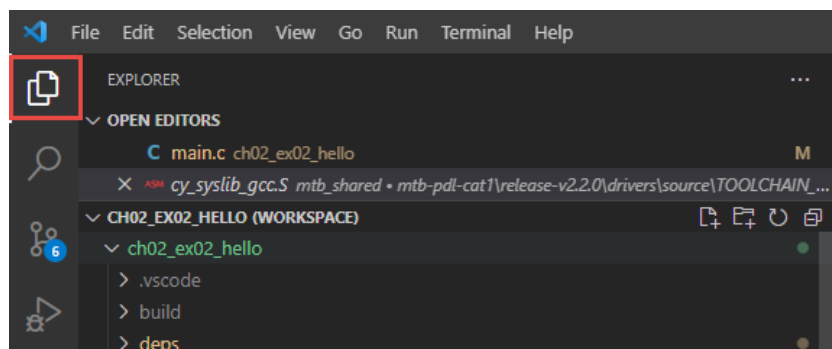
- ☐ 14. To debug, select the appropriate Launch item from the drop-down, and then click the green triangle to launch the operation.



- ☐ 15. 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.



- ☐ 16. 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.

*Note: If you are inside the Eclipse IDE for ModusToolbox™ you can use the **Terminal** tab instead of steps 1 and 2.*

- ☐ 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 Device Configurator to change a BSP setting

In this exercise, you will use the Device Configurator. The instructions 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_ex04_custom**.



2. Run the Device Configurator from within the application:

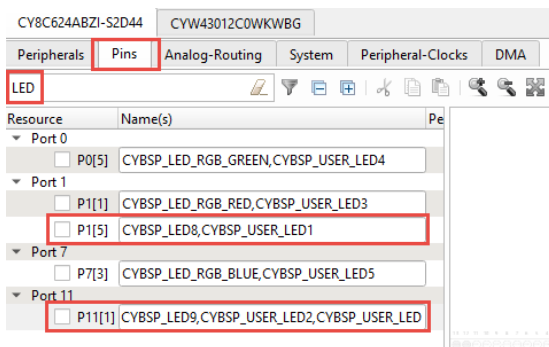
Eclipse: Click the link in the Quick Panel

VS Code: If there is no Run Task menu entry for the Device Configurator, use the CLI method instead.

CLI: Enter the command `make device-configurator`



3. Once the Device Configurator opens, make sure you are on the MCU tab (CY8C624ABZI-S2D44), 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.



4. Save the change and exit the Device Configurator.



5. Build and program. Observe the blinking LED is now the other user LED (red instead of orange).

Note: The solution project uses the templates mechanism described in section 2.3.4 to provide the updated device configurator file. That file is copied into the BSP during application creation.

Exercise 5: Use the BSP Assistant

In this exercise, you will create a custom BSP using the BSP Assistant and will use it in an application.

- ☐ 1. Start the BSP Assistant.
- ☐ 2. Create a new BSP called *MyBSP*. You can save it to any convenient location on disk.
- ☐ 3. Open the Device Configurator from the BSP assistant.
- ☐ 4. Once the Device Configurator opens, make sure you are on the MCU tab (CY8C624ABZI-S2D44), select the **Pins** tab and enter LED in the filter text box. This time, move the definition for `CYBSP_USER_LED` from `P1[5]` to `P7[3]`.
- ☐ 5. Exit the Device Configurator.
- ☐ 6. Explore other settings in the BSP such as dependencies and capabilities.
- ☐ 7. Exit the BSP Assistant.
- ☐ 8. Create a new application. Instead of selecting the CY8CKIT-062S2043012, use the **Browse** button to select your custom BSP.
- ☐ 9. Build and program the application. Observe the blinking LED is now the blue LED from the RGB LED.

Note: The solution has two directories for this exercise: `key_ch02_ex05_bsp` contains the custom BSP called *MyBSP* while `key_ch02_ex05_application` contains the application that uses the BSP. If you want to use the solution project, you will need to use the **Browse...** functionality inside Project Creator to specify both the BSP and the application.

Trademarks

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

Published by
Infineon Technologies AG
81726 Munich, Germany

© 2022 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.