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

Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	CY_ISR_PROTO(MyISR); make build
Italics	Displays file names and paths	*sourcefile.hex*
[**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 2.16.

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 similar to 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 (this will be explained in more detail 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 with the security of knowing that the results won't change.

The IAR Embedded Workbench and Keil μVision IDEs have their own internal build system. 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)

Each tool in ModusToolbox™ can be run using a command line interface that interacts directly with the make system for users that don't want to be constrained to working within an IDE.

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. The shell is in the installation directory under *ModusToolbox/tools_<version>/modus-shell.* It is listed in the **Start** menu under **ModusToolbox <version>** or discoverable by typing `modus-shell` in the Windows search box until it appears in the list.
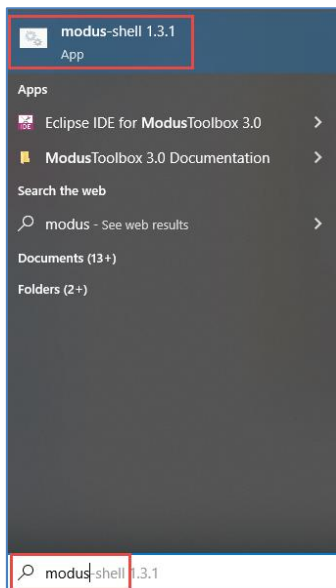
Once you start the shell, it will look something like the following image. You will need to switch to the directory containing the ModusToolbox™ application that you want to operate on before running any command line operations.
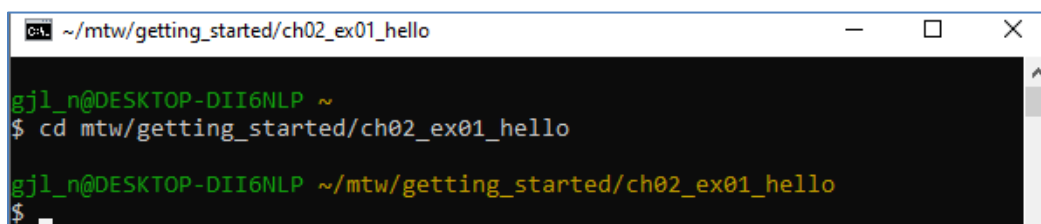
2.2.2.2   macOS / Linux

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 has already been changed to the application's root directory. On the Windows operating system, this will be a modus-shell. 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). For more information, refer to the ModusToolbox™ tools package user guide.

Make command	Description
`make help`	This command will print out a list of all the make targets. To get help on a specific target type `make help CY_HELP=getlibs` (or whichever target you want help with).
`make getlibs`	Process all the *mtb* files (or *lib* 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
```


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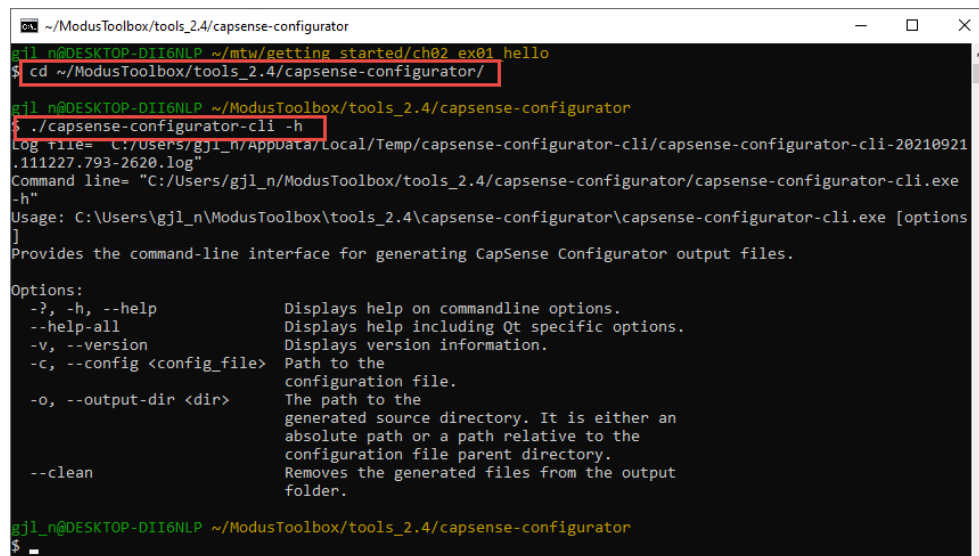
Most of the tools can be launched from an application's directory where the *Makefile* is located. Use `make` along with the executable name of the tool. A couple useful examples:

- `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. You can use the **Start** menu entries or the search box on Windows.

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
```

2.3 Application creation

The first step you need to follow 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 two pieces of information: (1) a starting BSP that specifies the hardware configuration; and (2) a starting template application. You can choose the standard options provided by Infineon or 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 2.11.*

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 located 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 2.12.

Note: *You can also create projects in offline mode. See section 2.13.3 for details.*

Once the manifests have been read, the first window asks you to select the target BSP. The target BSPs are categorized by device family, as shown below.
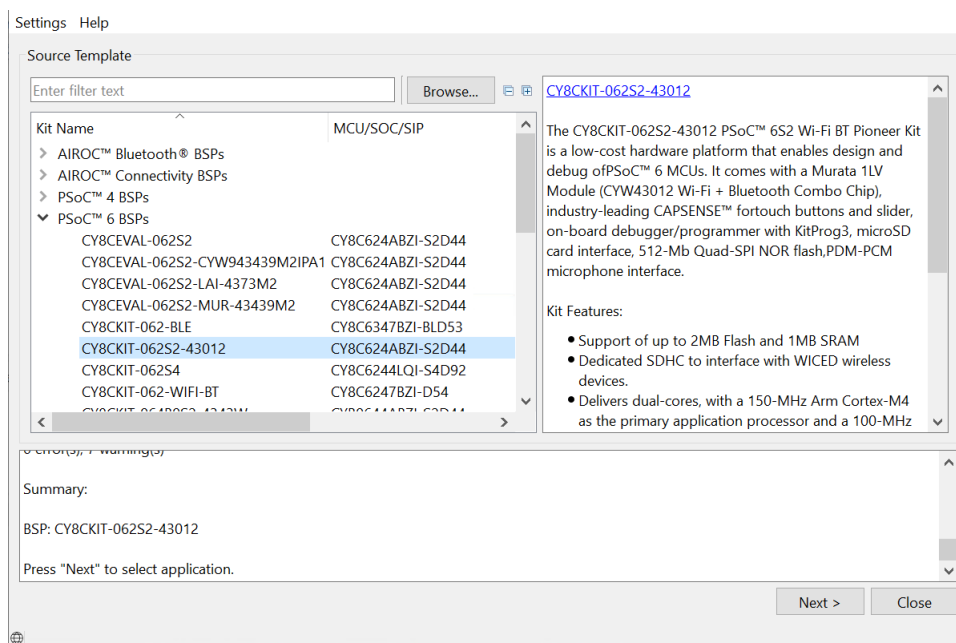
Note: *The **Browse** button allows you to specify your own custom BSP when applicable.*

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. More details on this feature will be available later in the chapter.

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.
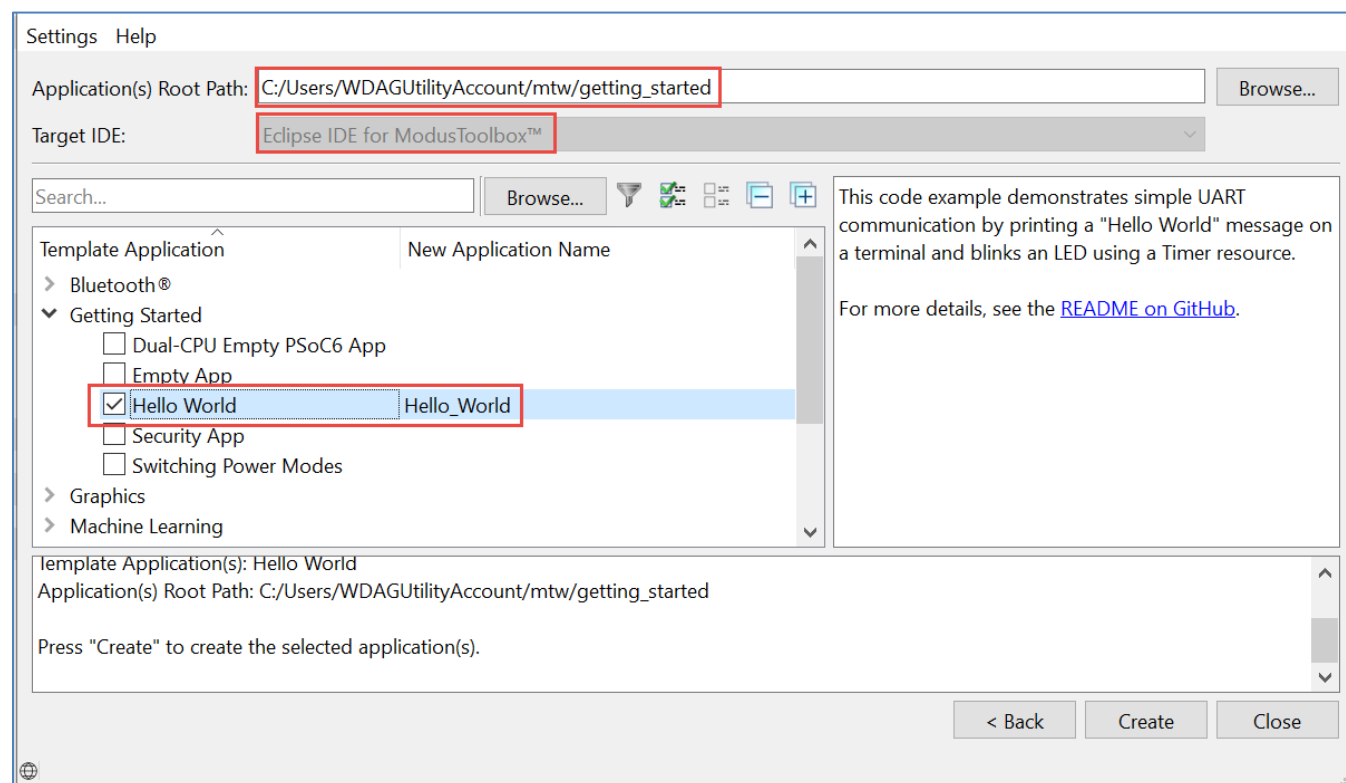
You can use the "Search…" box to enter a string to match from the application names and their descriptions. For example, if you enter *wifi*, you may see a list like this. (The exact list will change over time as new code examples are created). Only categories containing matching applications will be listed.
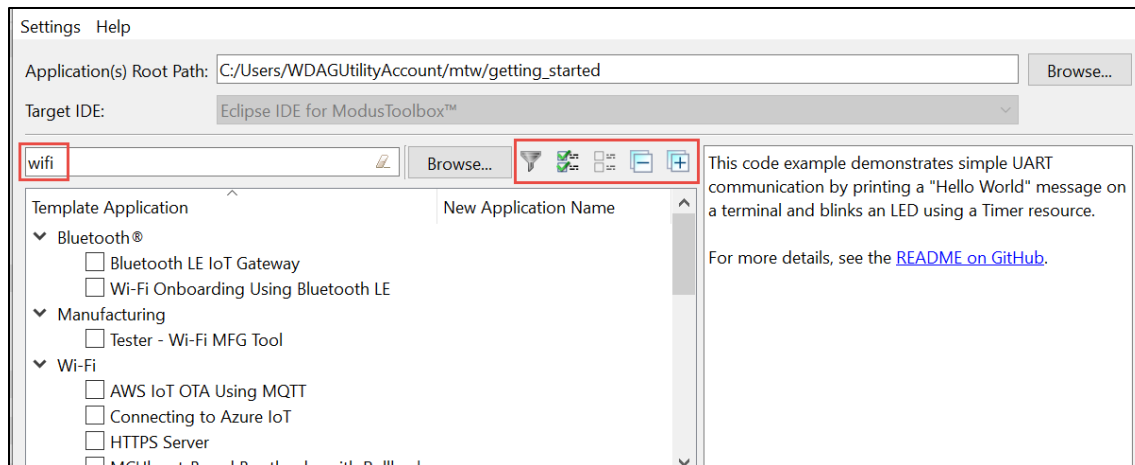
If desired, you can select all the applications in the search result by using the **Select All** button or you can unselect them with the **Unselect All** button. You can also expand or collapse all categories with the **Expand All** and **Collapse All** buttons.

The **Filter** button can be used to show only the applications which are currently checked. This can be useful if you have a large number of applications checked and want to view 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 leads to the directory that contains the Makefile, otherwise your application creation will fail.

Once you select an application directory, it will appear in the **Import** category.
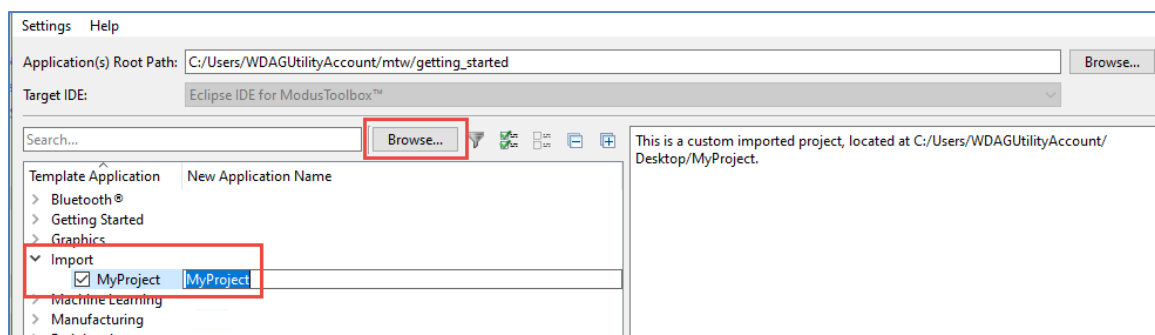
Note: *The existing project can either be located in your workspace or somewhere else. It does not need to be a full Eclipse (or another IDE) project. At the very least, it needs a* Makefile *and source code files, however it may contain other items such as configurator files and mtb files (which are 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.

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**.

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

2.3.2　　　Project Creator 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 view 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>
```

In order 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 "
    --user-app-name "my_app"
```

Note:　　　*If you are using a tilde (~) to specify the user's home directory, double-quotes must be placed around the target directory name.*

Other arguments are available to create an application using a local custom BSP (--board-path) or a local custom application (--app-path) if desired. This is equivalent to the **Browse** buttons in the GUI.

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:　　　*Since creating a new application manually is more complicated, it is typically better to use the Project Creator tool using the GUI or command line interface. However, this is not the case if you have an existing application as a Git repo that you want to share. See section 2.11 for further 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 to the following 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 add a dependency for the BSP you want to use into the *deps* directory and set it as the target BSP. You can do this by running the Library Manager tool, or you can run the Library Manager command line interface version. For example:

```
library-manager-cli.exe --add-bsp-name CY8CKIT-062S2-43012 --add-bsp-location local
                        --project .
```

If you do not want to use the library manager, you can manually add a *.mtb* file for your BSP to the *deps* directory, set the name of that BSP as the `TARGET` in the *Makefile*, and then run `make getlibs`. The *.mtb* file must have `$$LOCAL$$` as the destination for the BSP. For example:

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

If you use a standard BSP, the name will be prefixed with *APP* (e.g. *TARGET_APP_CY8CKIT-062S2-43012*) during application creation to indicate that the BSP is owned by the application (i.e. no longer a Git repo). The `TARGET` name in the *Makefile* is updated automatically during application creation to include the prefix either by the library manager or `make getlibs`. If you use a custom BSP, the name is not changed during application creation.

At this point, the application is ready for use. You can perform 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 final 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. As such, 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 Manger 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 For example, device configuration using the templates mechanism.

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 will be 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, and 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 exists) 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 bsps 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.
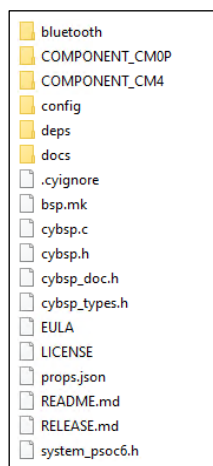
The various directories and files in a BSP include:

Directory/File	Description
bluetooth/	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.
COMPONENT_CM4/ and *COMPONENT_CM0P/*	Startup code and linker scripts for all supported toolchains for each of the two cores - the CM4 and the CM0+.
config/	Contains BSP configuration files. These are edited by using the appropriate configurator. For example, design.modus is edited using the Device Configurator. Contains the cyreservedresources.list 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. Contains the GeneratedSource that is created by the configurators.
deps/	Contains *mtbx* 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 explained in more detail in section 2.12.
docs/	Contains HTML based documentation for the BSP.
.cyignore	Used to exclude directories from the build if necessary for this BSP.
<bsp_name>.mk	Defines the DEVICE and other BSP-specific make variables such as `COMPONENTS`.
cybsp.h *cybsp.c*	These files contain the API interface to the board's resources. You need to include only *cybsp.h* in your application to use all the features of a BSP. Call the `cybsp_init` function from your code to initialize the board (that function is defined in *cybsp.c)*.
cybsp_doc.h	Used to generate documentation on the BSP's resources. It is not used for any functional purpose.
cybsp_types.h	Contains macros for using various board peripherals. For example, `CYBSP_LED_STATE_ON` and `CYBSP_LED_STATE_OFF` are set to the appropriate values based on the LED hardware on the boad (e.g. active low or active high).
locate_recipe.mk	Provides the path to the core and recipe make files that are needed by the build system.
*p*rops.json	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.
README.md	Contains BSP documentation in Markdown format.
RELEASE.md	Includes BSP release notes.
system_psoc6.h	Contains device system header information for PSoC™ 6 MCU BSPs. 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 via the **Browse...** mechanism, the name of the in-app BSP created in the application will be the same as the name of the 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 will allow the BSP to be renamed during the application creation stage.

After the BSP is created, you can rename it manually by just changing the name of the directory in which the BSP is contained.

Note: *The BSP name must always start with "TARGET_". Spaces and other special characters should not be used in the name. Pay attention when you change the name of BSP in an application so that you can 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 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:
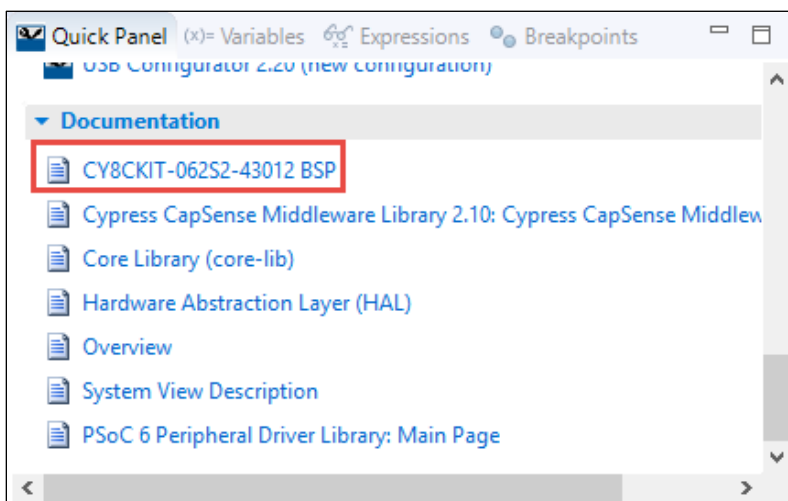
2.5 Application directory organization

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 part of the same directory. In a typical multi-core application, the top-level directory will be part of 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. A typical PSoC™ 6 MCU single-core application may look like the following example:
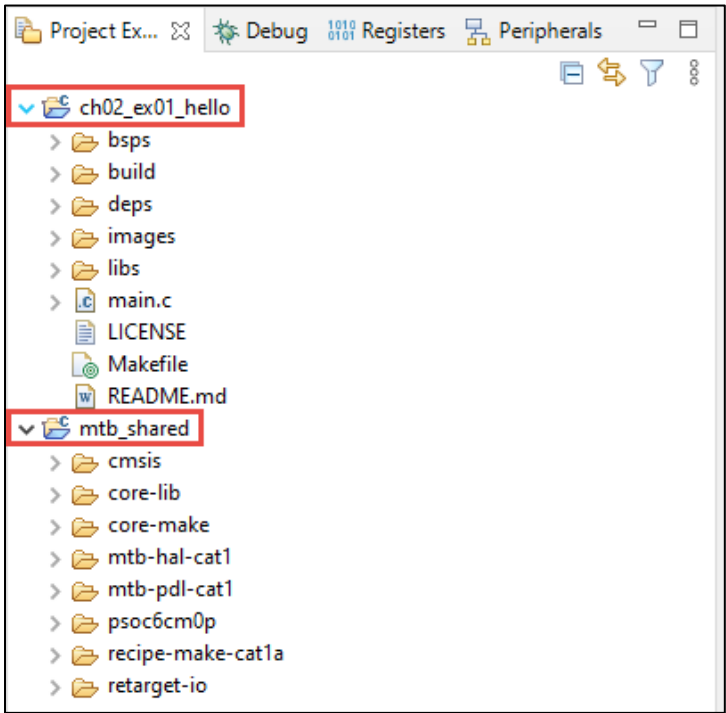
The exact files and directories may be different for other applications, device families or solutions but the basic structure will be the same. The directories and files in the example application's project directory include:

Directory/File	Description
Binaries	Virtual directory that points to the *elf* file from the build. This directory may not appear until a build is done.
bsps	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 BSPs 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.
build	This directory contains build files, such as the *elf* and *hex* files.

Directory/File	Description
deps	This directory contains *mtb* files that specify where the `make getlibs` command locates the libraries directly included by the project. Notice that libraries included via *mtb* files may have their own library dependencies listed in the manifest files. We'll talk more about dependencies, *mtb* files and manifests later in the chapter. As you will see, the source code for libraries either go in the shared repository (i.e. for shared libraries) or in the *libs* directory inside the application (i.e. for libraries that are not shared).
images	This directory contains artwork images used by the documentation.
libs	The *libs* directory contains source code for local libraries (i.e. those that are not shared) and *mtb* files for indirect dependencies (i.e. libraries that are included by other libraries). Most libraries are placed in a shared repo and are therefore not in the *libs* directory. The *libs* directory also includes a file called *mtb.mk* which specifies which shared libraries should be included in the application and where they can be found. *Note:* *As you will see, the libs directory only contains items that can be recreated by running* `make getlibs`*. Therefore, the libs directory should normally not be checked into a version control system.*
templates (not shown)	Some code examples have a directory called *templates* that contains files to override the default BSP configuration. See section 2.3.4 for details.
main.c	This is the primary application file that contains the application's code. Depending on the application, this file may have a different name or have multiple source code files. In more complex applications, the source code files might be in a subdirectory such as *source*.
LICENSE	This is the ModusToolbox™ software license agreement file.
Makefile	The *Makefile* 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, meaning that once you create an application, you can go back and forth between the IDE and CLI as you see fit. See section **Error! Reference source not found.** for details about editing the *Makefile*.
README.md	Almost every code example includes a *README.md* 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 directory 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.

Given the reasons listed above, it is better to use the Library Manager to update the TARGET BSP because it makes all of the necessary changes to the project. If you are working exclusively from the command line and do not 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 `COMPONETS_<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=
```

Note: *When you add a library to an application, a* COMPONENT *is automatically created for the name of the library with the prefix MW. For example, if you add the capsense library to an application, a* COMPONENT *is created called* MW_CAPSENSE*.*

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 variable to specify a list of C/C++ source files to include in the build.

INCLUDES   Use this variable to specify a directory containing header files to include in the build. Only the specific directory is included – not sub-directories.

SEARCH   Use this variable 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. SEARCH 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. 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. A dual-core application for an MCU with an Arm® Cortex®-M4 and an Arm® Cortex®-M0+ MCU may look like the following example:

Application   **CM4 project**   **CM0+ project**
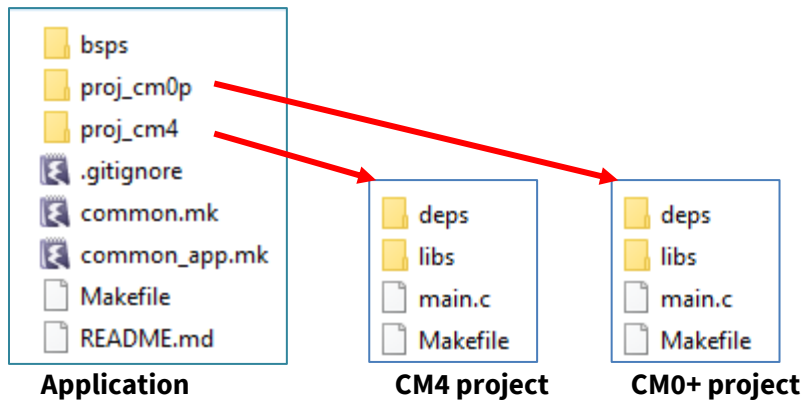
Inside an IDE, the application will look similar to this:
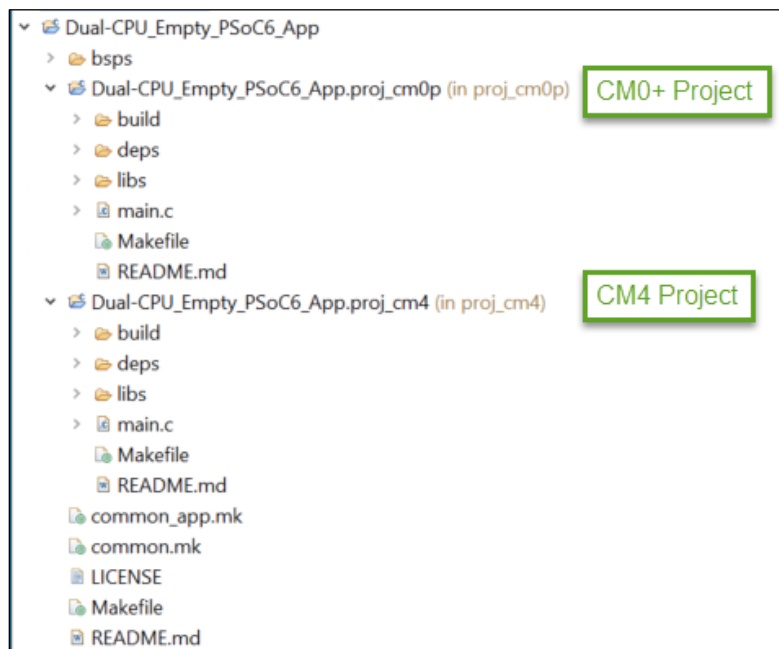
Note: *There is a* deps *and* libs *directory inside each project, meaning each project has its own libraries. Since each core will be running separate code, each one has its own* main.c *file. Alternatively, since the entire application will be running on one physical board the* bsps *directory is at the application level.*

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 are 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 main function. *Note:* *From the command line, it is only possible to debug a single core at a time. However, some IDEs support simultaneous multi-core debugging.* *Note:* *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 used to control the build operate the same as variables for single-core applications. However, the make variables are split between the different files depending on how they are used (i.e. 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 the variable for each project *Makefile,* is set to `PROJECT`.

One noticeable 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 include 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:
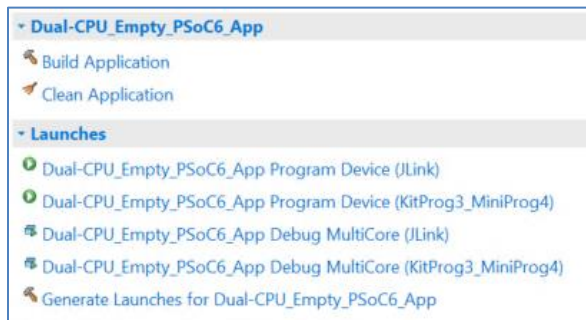
The project levels will look like this:
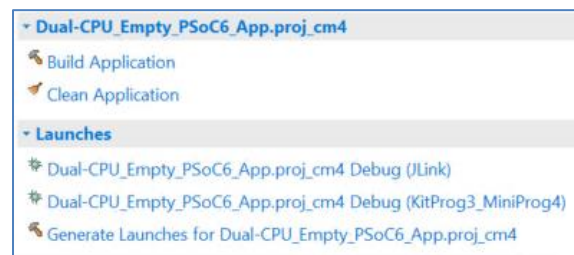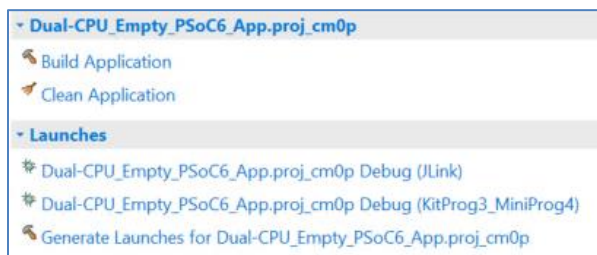
Using the Quick Panel, you can build or program the entire application either from the application level or one of the sub-projects.

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 the **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. This version of Eclipse works similarly to standard Eclipse, nevertheless a short introduction is helpful. While it is not necessarily recommended to use Eclipse over another IDE, the others are not customized for the ModusToolbox™ ecosystem and 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.
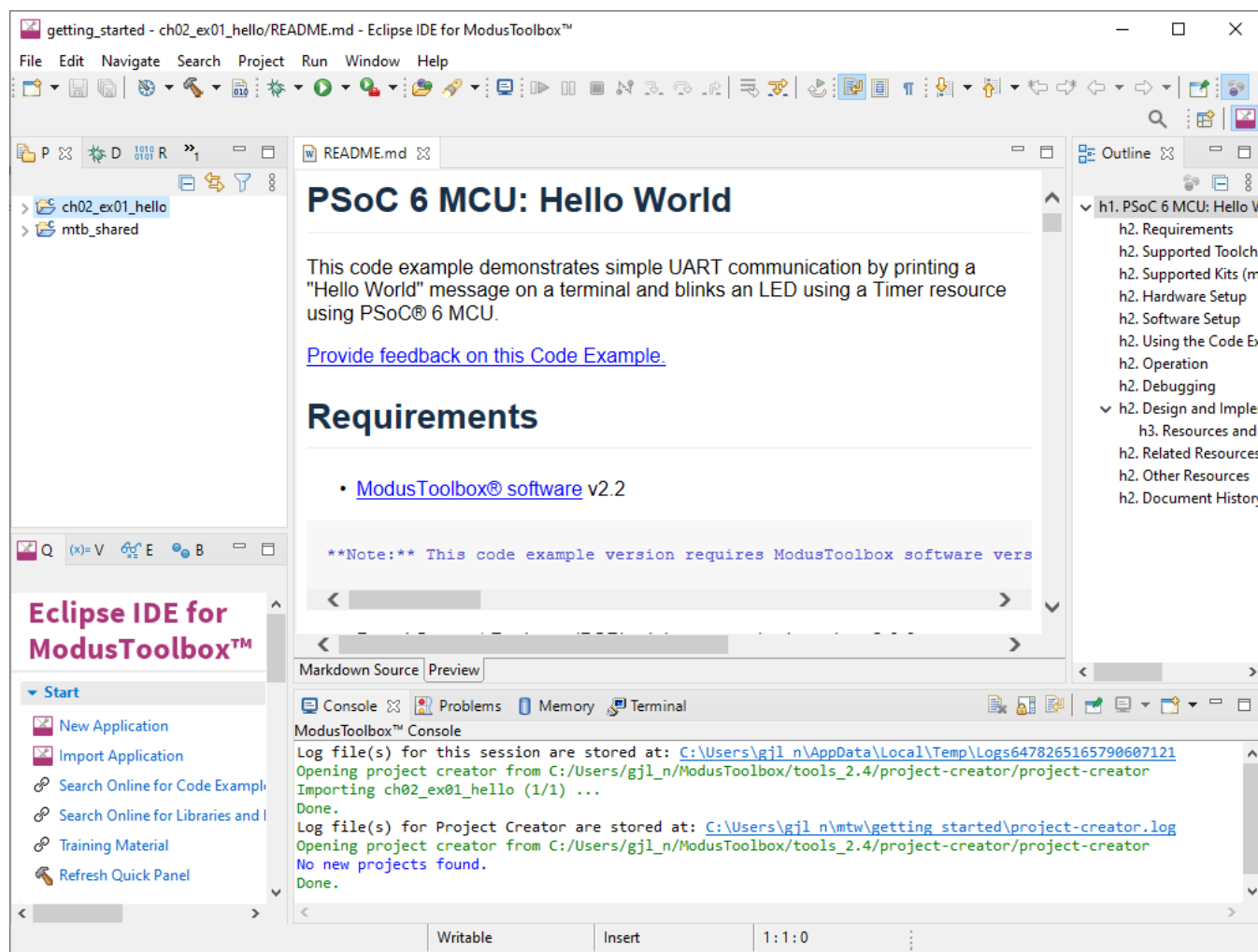
For more information about the IDE, refer to the Eclipse IDE for ModusToolbox™ User Guide.

2.6.1 Quick Panel

We have extended the Eclipse functionality with a "ModusToolbox" perspective. Among other new features (such as the addition of many 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
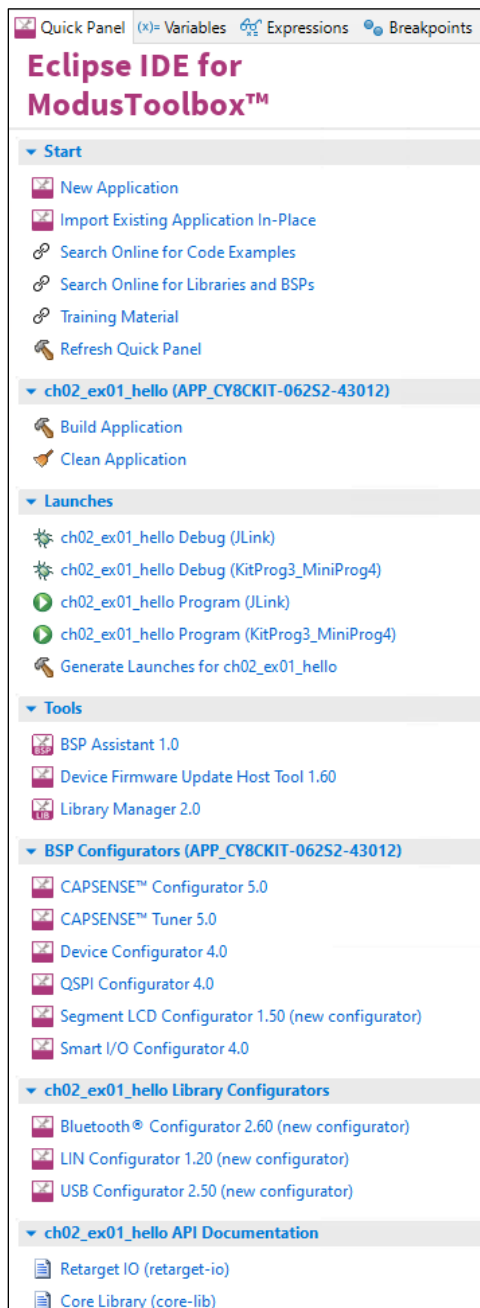
The launches section is what you should use when you want to program a kit or start the debugger. If you are using an Infineon PSoC™, you should use the selections that say "KitProg3_MiniProg4" since most kits contain an integrated KitProg3 programmer. For example:
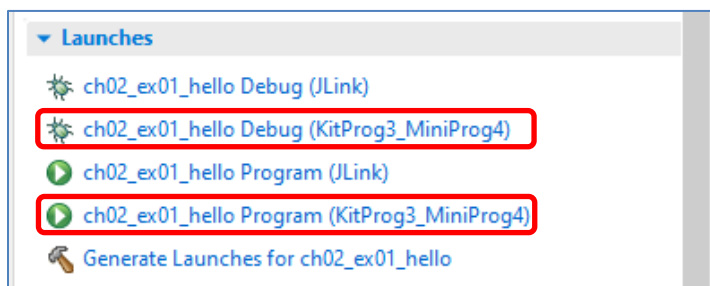
You can also access these using the bug and play icons along the top ribbon, but make sure you select the correct one from the drop down. For that reason, the Quick Panel is usually easier to use.

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.
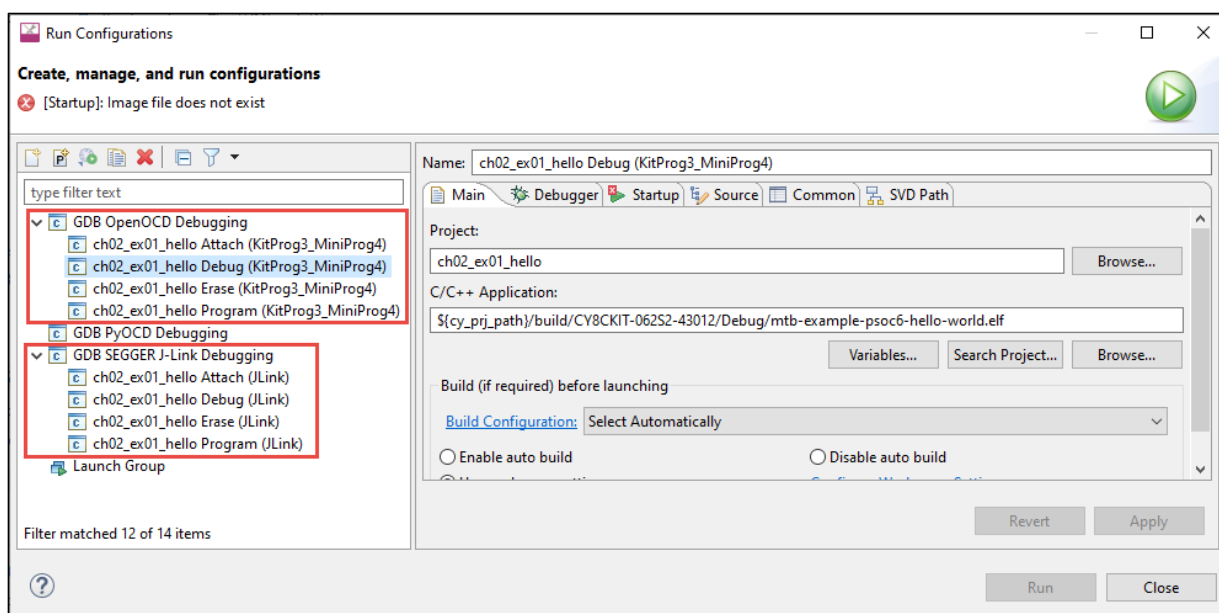
2.6.2　　　Help menu

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.
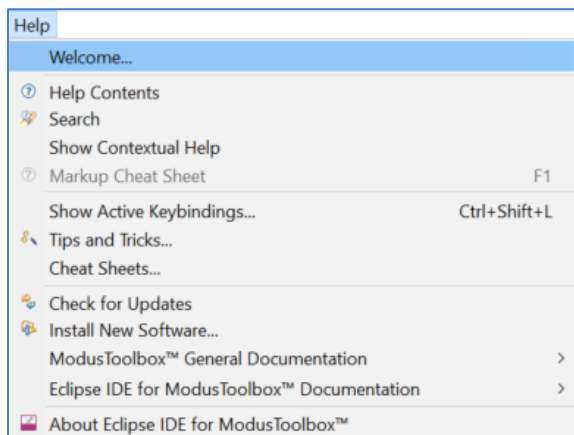
2.6.3　　　Integrated debugger

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.
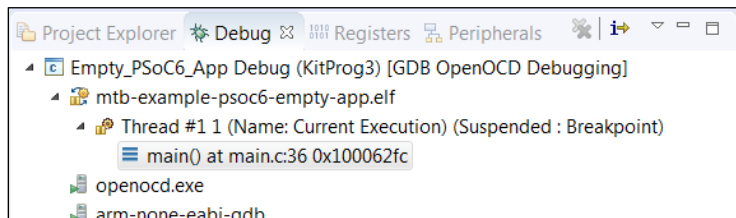
2.6.4　　　Console

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.
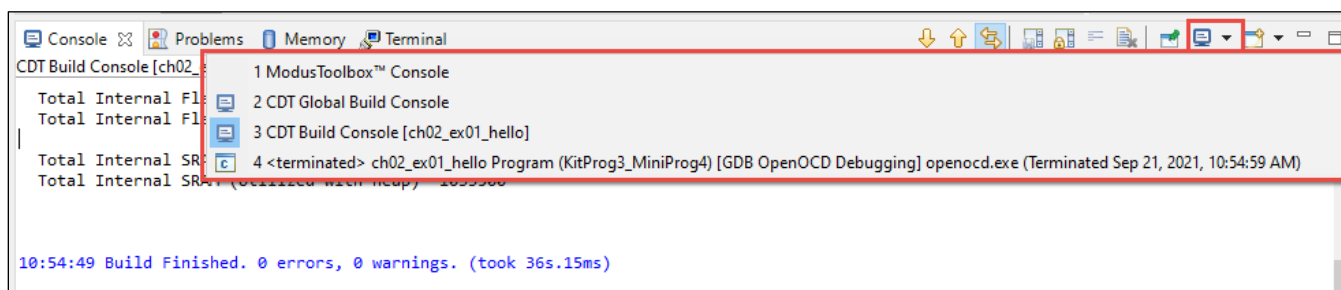
2.6.5 Terminal

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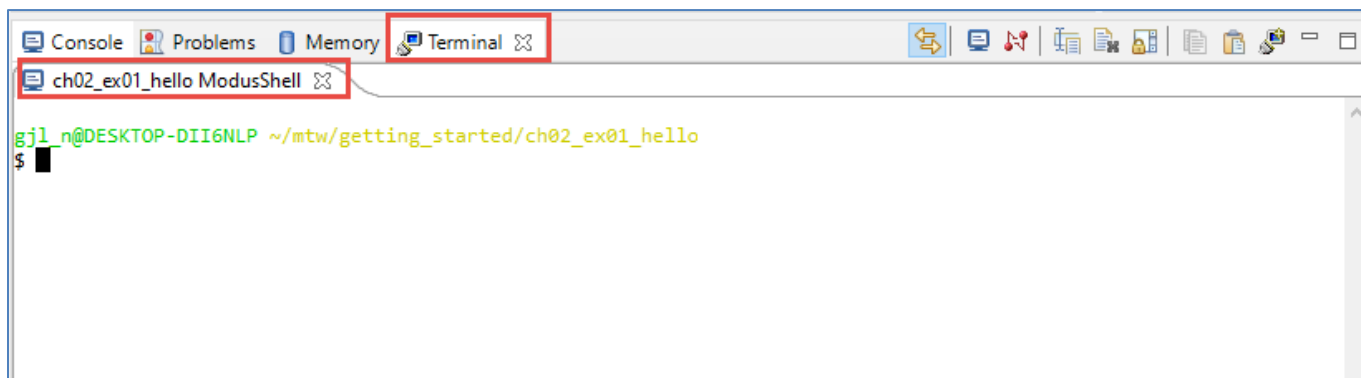
See section 2.2.2 for details on using the command line interface.

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.
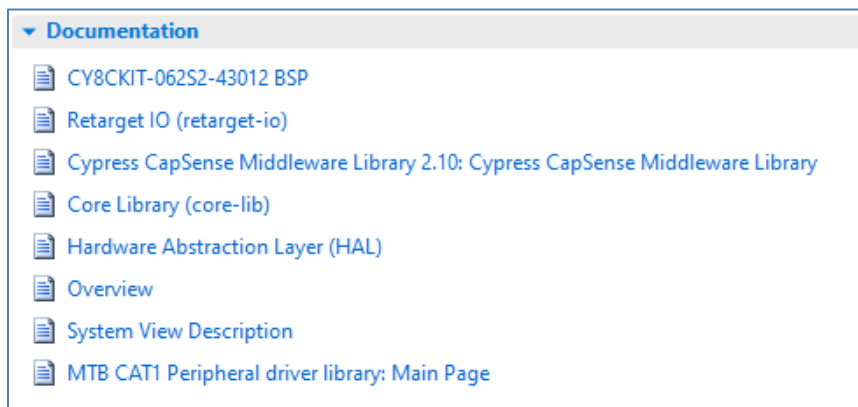
2.6.7 Eclipse IDE tips & tricks

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

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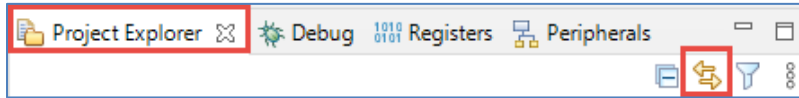
Refer also to our [Eclipse Survival Guide](#) for more tips and tricks.

2.7   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 is placed into 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 2.12). 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 2.12.

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 the files 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/cypressemiconductorco/retarget-io#latest-
v1.X#$$ASSET_REPO$$/retarget-io/latest-v1.X
```

The variable $ASSET_REPO$ points to the root of the shared location, specified in the application's *Makefile*. Use $LOCAL$ in place of $ASSET_REPO$ in the *mtb* file before downloading the libraries to make a library local to the app instead of shared. 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/cypressemiconductorco/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. For example, *mtb_shared* is used for shared libraries and *libs* is used for local libraries. Lastly, a file called *mtb.mk* is created in the application's *libs* directory. The build system uses the *mtb.mk* file to locate 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. Instead 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 both get pulled from GitHub when you run `make getlibs`.

The *mtb* files for indirect references and the *mtb.mk* file are stored in the *libs* directory and do not usually need to be checked in. In fact, the default *.gitignore* file found in our code examples excludes the entire *libs* directory. You should not need to check in any files from that directory and 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 the Infineon libraries. This 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 2.12.

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`, which requires that the library is in a Git repo

For the first four 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 to put manually added libraries somewhere else.

If you choose to 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. You should use a fixed version in the *mtb* file unless you want the dynamic update behavior.

Adding dependencies can be easily achieved 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. Therefore, 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 and 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 **Error! Reference source not found.**.

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, such as:

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. Enter through `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, using manifest files, which will be discussed in section 2.12. Therefore, *mtb* files included manually in your application that are not in a manifest file will *not* show up in the Library Manager.

Note: *The Library Manager never changes anything on disk until the **Update** button is clicked.*

Meanwhile, the Library Manager reads/creates/modifies *mtb* files, creates/modifies the *mtb.mk* file, and then runs `make getlibs`. When you change a library version and click **Update** for example, 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:
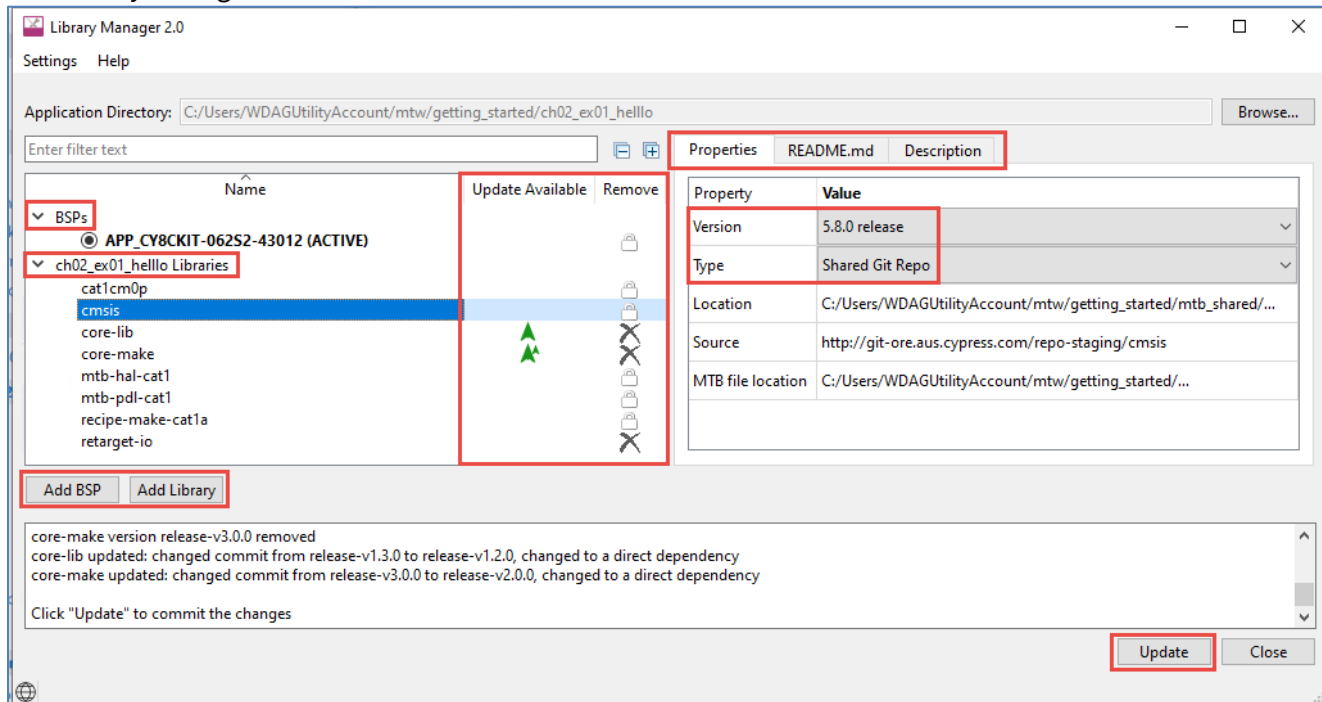
As you can see, there is a section for *BSPs* and a section for *Libraries*. If you are working with a multi-core application, there will be a section with libraries for each project.

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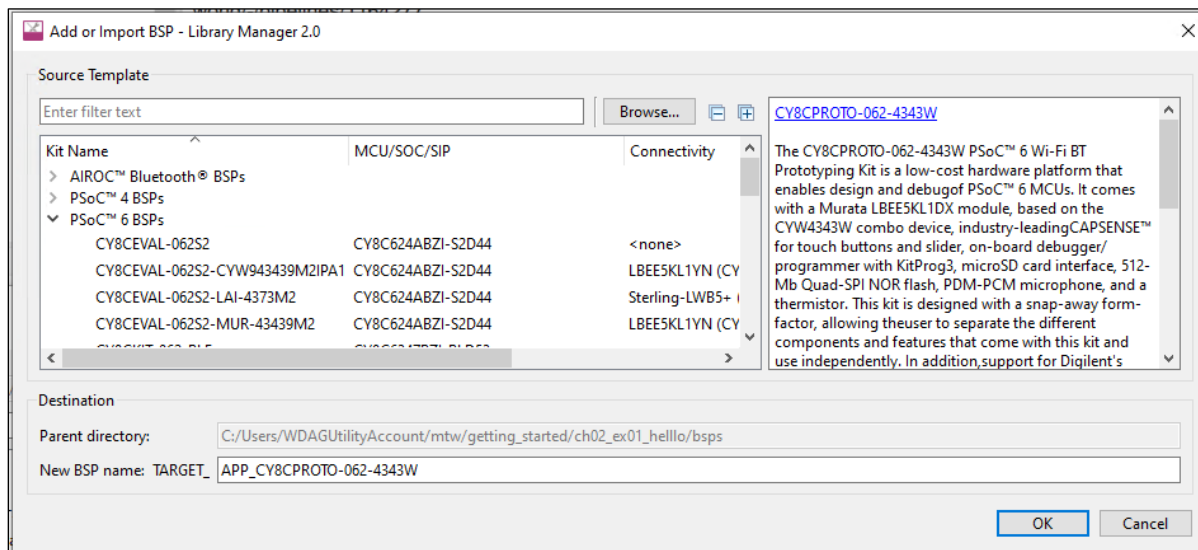
Version: You can select a specific fixed version of a library for your application or choose a dynamic tag that points to a major version but allows `make getlibs` to fetch the latest minor version (e.g. Latest 1.X). The drop-down will list all available versions of the library.

Type: Libraries can be a "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:
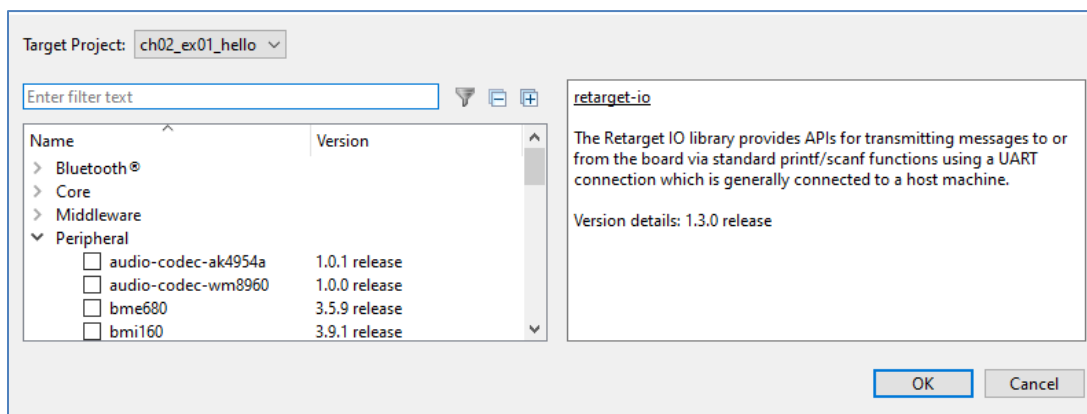
You can use the **+** and **-** buttons at the top to expand/collapse all categories and the text box to filter out specific libraries.

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 labeled "Generate Launches for <app-name>" used to fix the files.

In the case of VS Code, you must manually update the appropriate launch configurations after updating the active BSP or manually editing the `TARGET` in the *Makefile*. 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 be impossible to open the directory from inside the Eclipse IDE and it may not work properly for IntelliSense. To fix this issue, 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**.
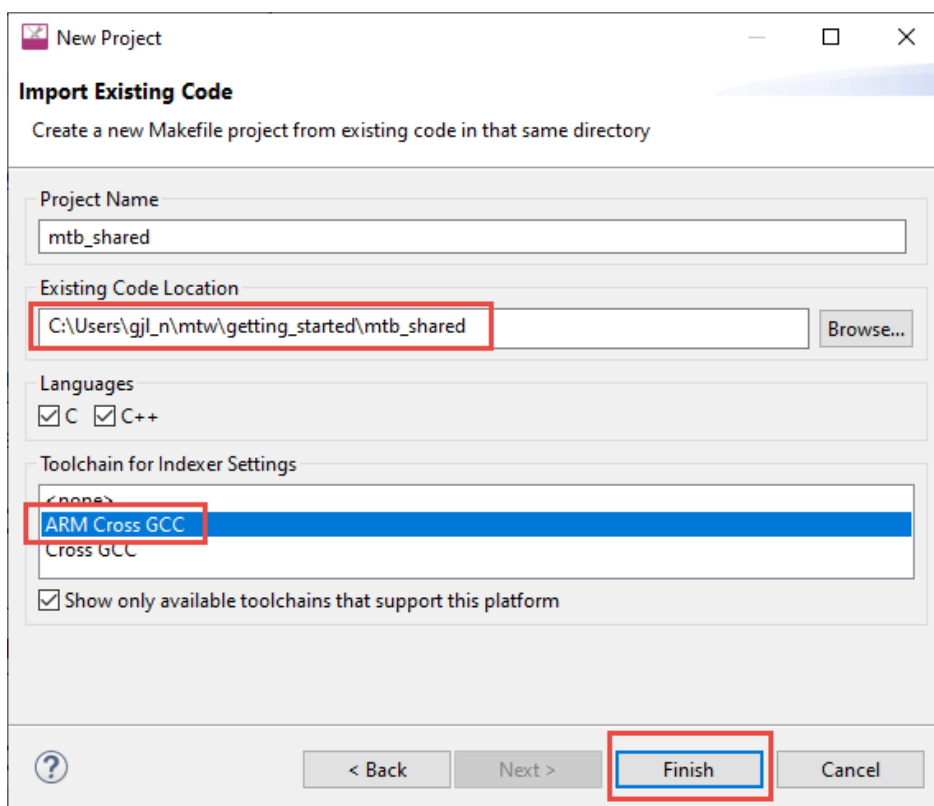
6. To get IntelliSense to work again for an application you must re-build it first.

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
- 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, which will be discussed in further detail later, or from within the application. The configurators are modifying files that are a part of the BSP regardless of which option you choose to launch with.

Note: *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 try to launch the Device Configurator, you will receive 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.
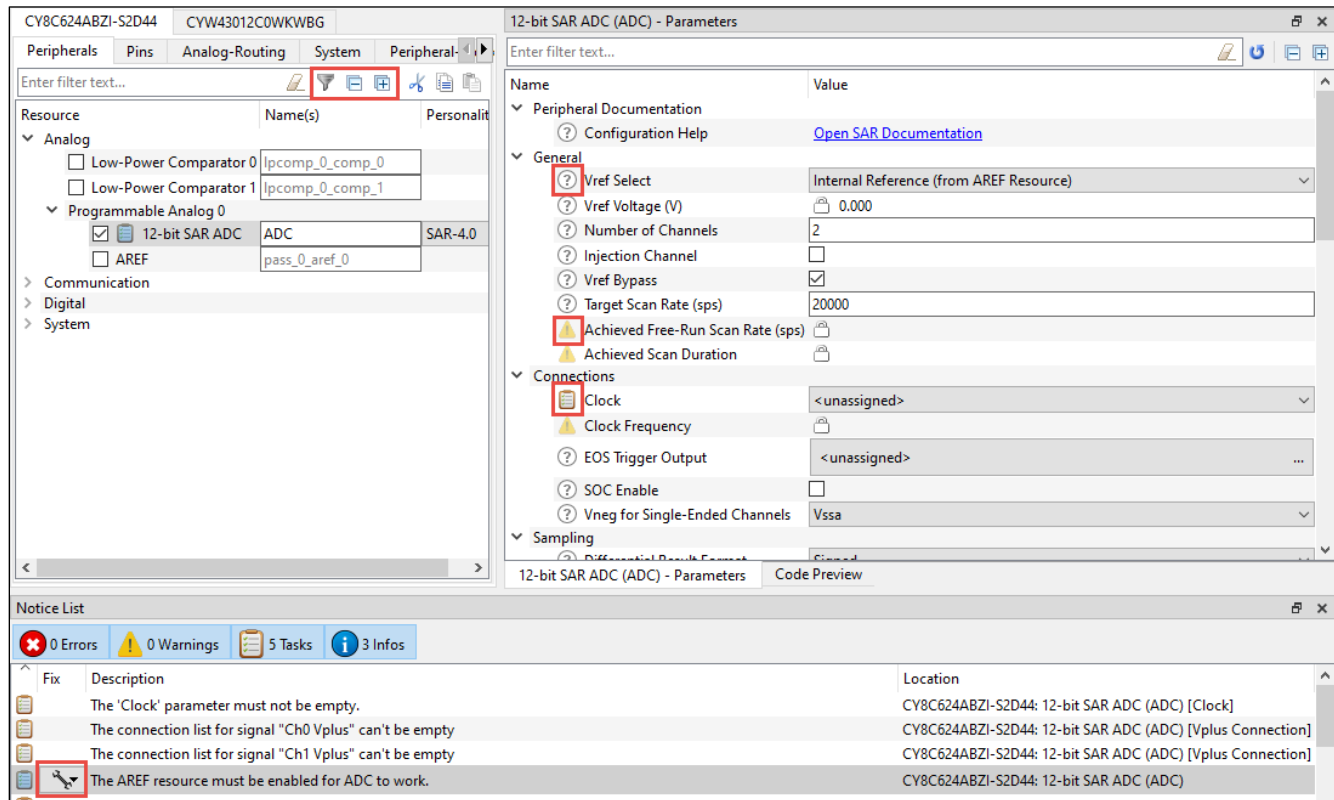
Note: *The **+** and **-** buttons expands/contracts all categories while the filter button shows only items that are selected. This is particularly useful on the **Pins** tab since the list of pins is sometimes very long.*

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 on the item you want to copy to, then select **Paste**. If you use cut/paste, the old item will be disabled and the new item will be enabled.*
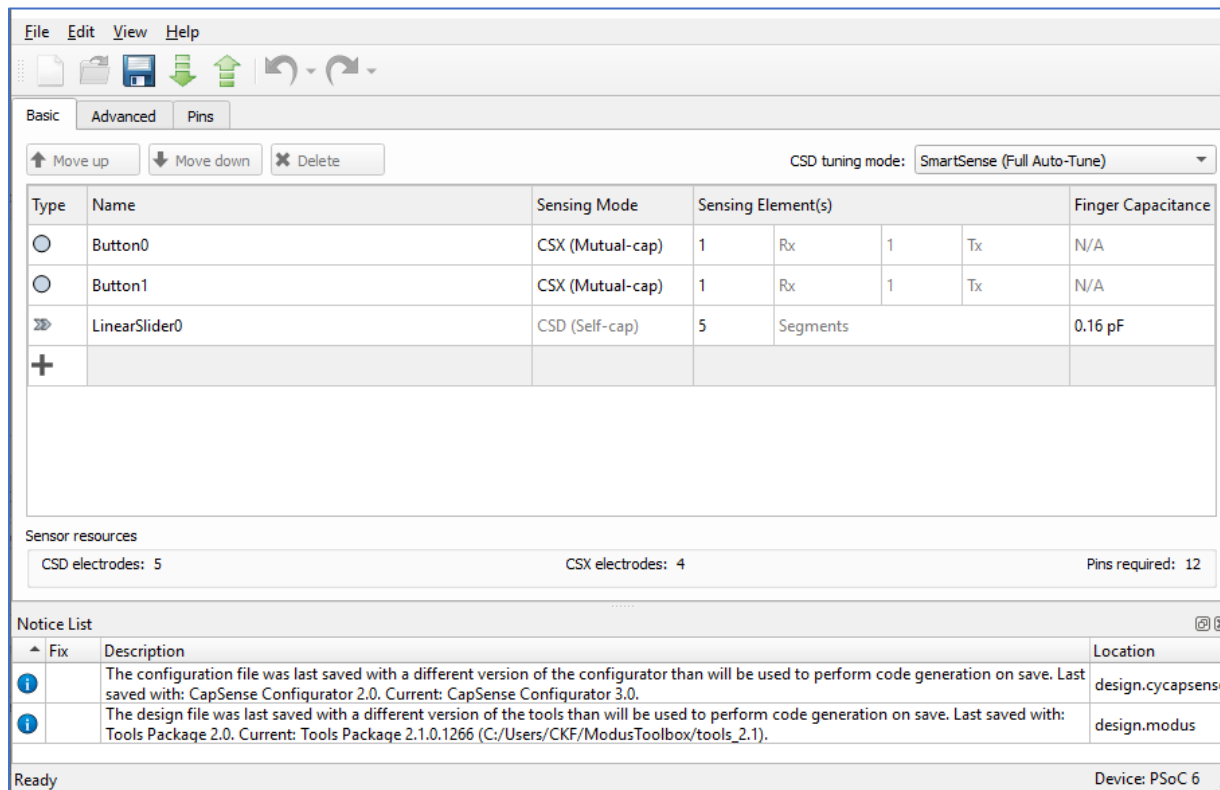
Several helpful icons will show up next to various items in the configurator. A few of interest are:

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.	Calculated parameters (for example, scan rate) cannot be determined if a clock has not been enabled for an ADC.

All other BSP configurators can be launched as 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 its performance.

2.8.1.3 Smart I/O configurator

Configuring the Smart I/O block adds programmable logic to an I/O port. However not all MCUs have Smart I/O. For the MCUs that do, there are usually only a few ports with Smart I/O capability, which makes it important to consider that beforehand. 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.
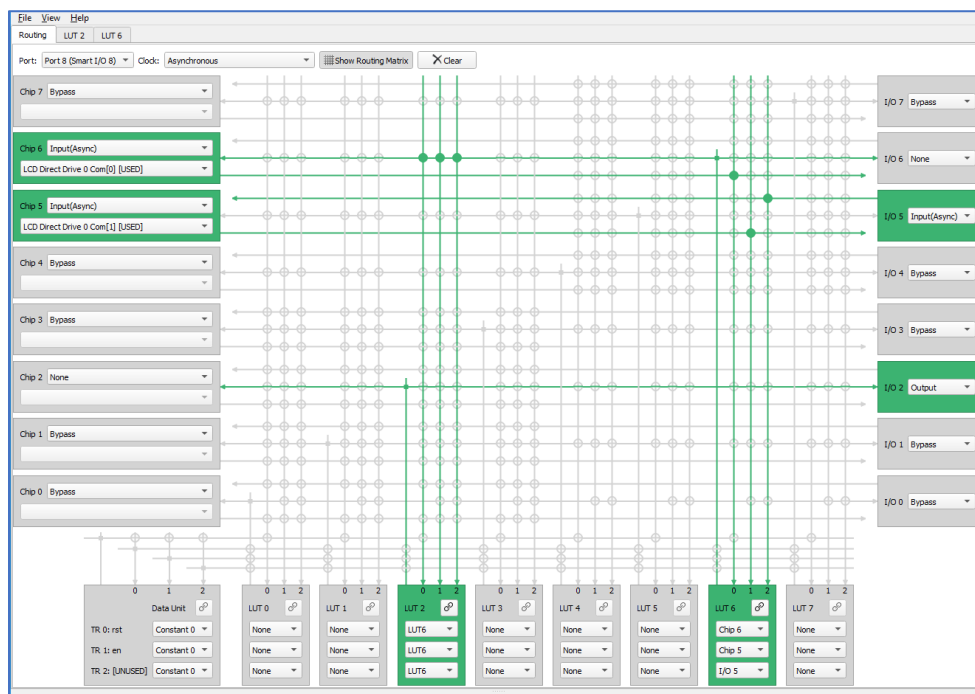
The port is chosen via a drop-down along the top of the window. Also selected at the top of the window is the clocking scheme. It can be asynchronous, or a variety of clock sources can be selected.

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 three 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 in 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).
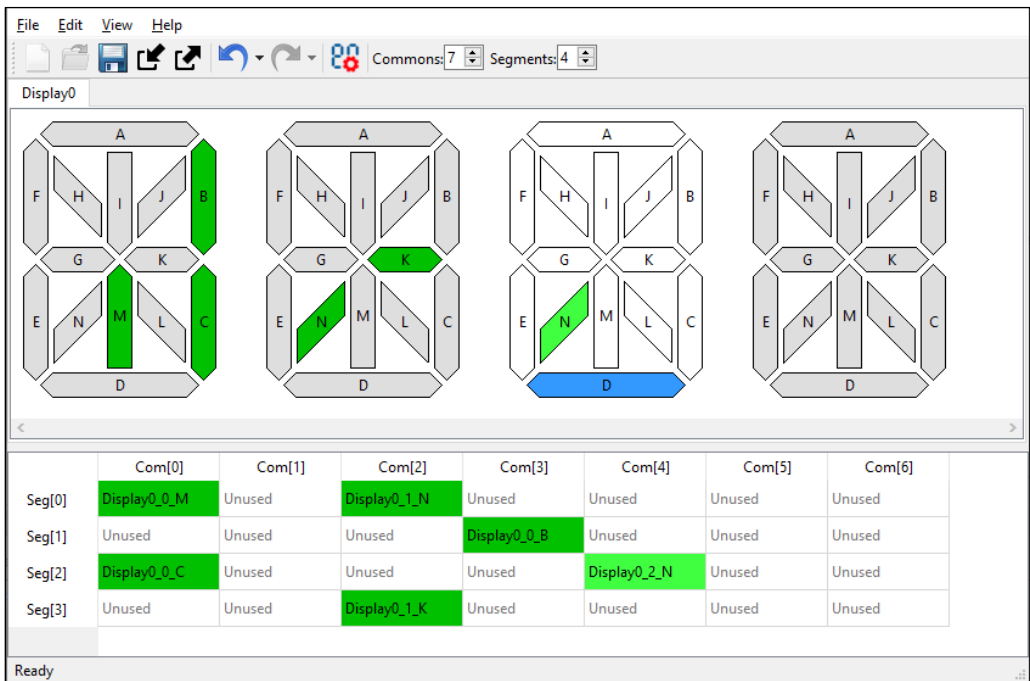
2.8.1.4 QSPI configurator

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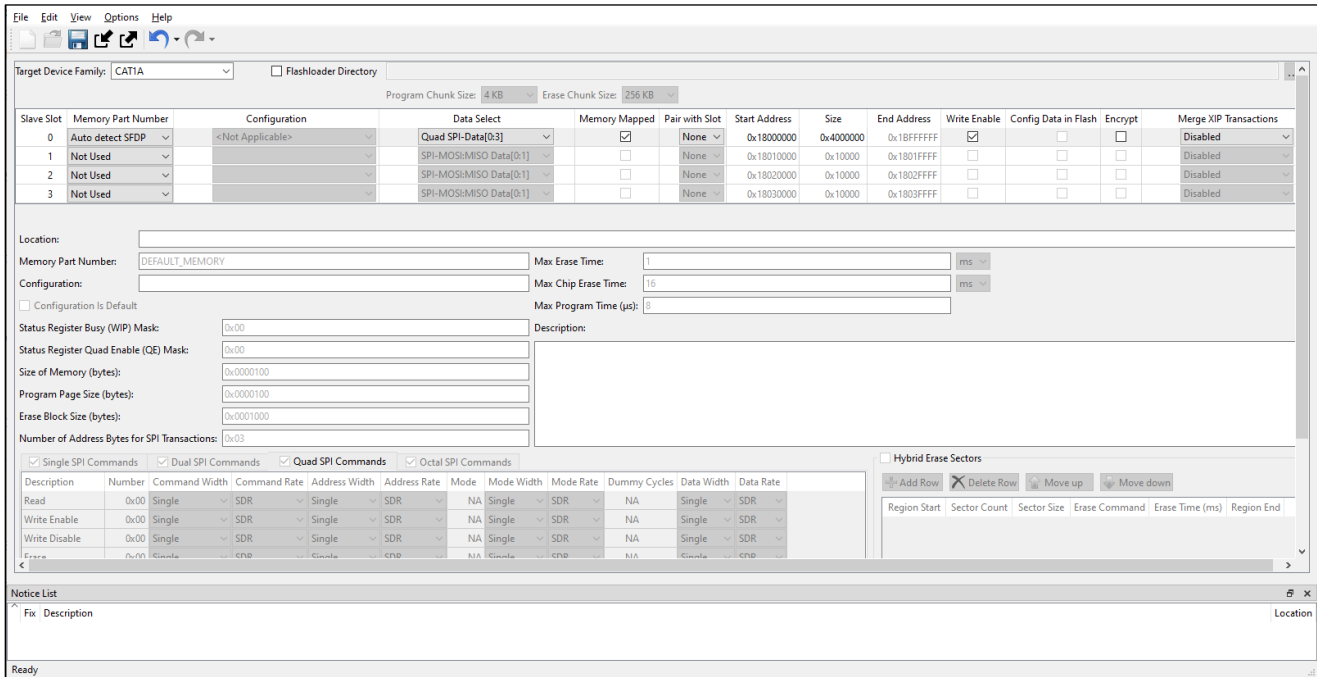
2.8.1.5 SegLCD configurator

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.
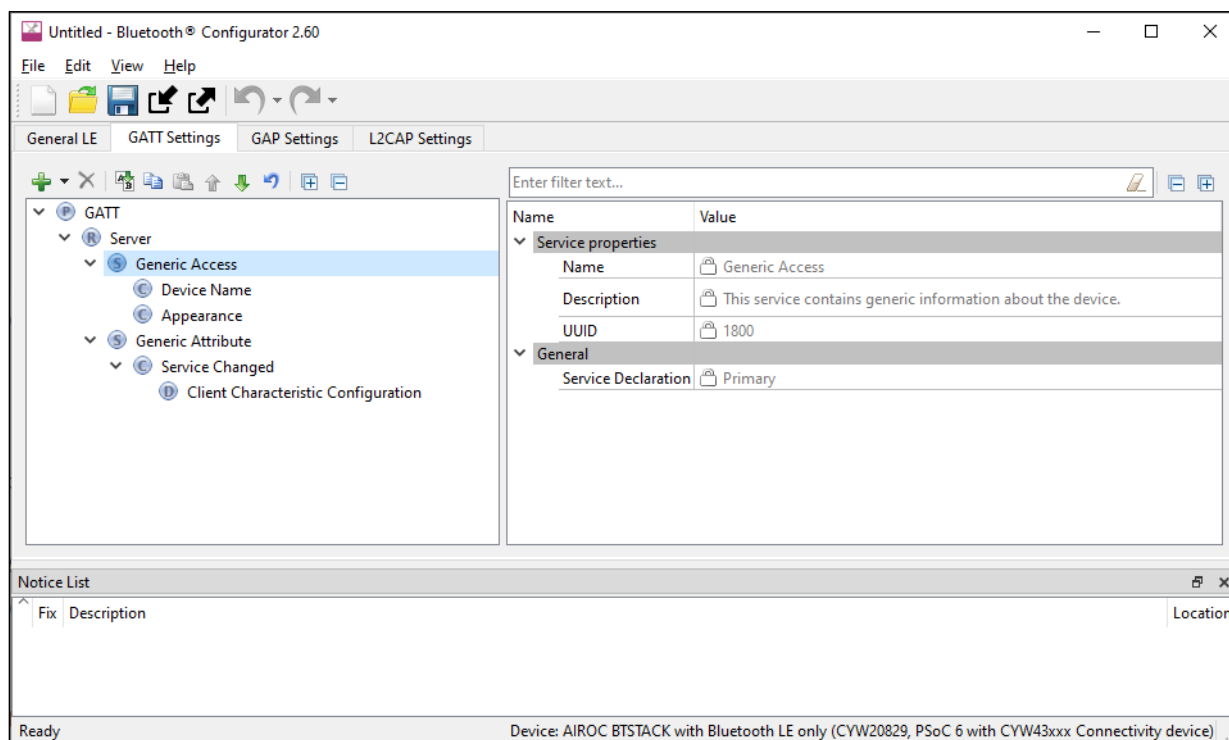
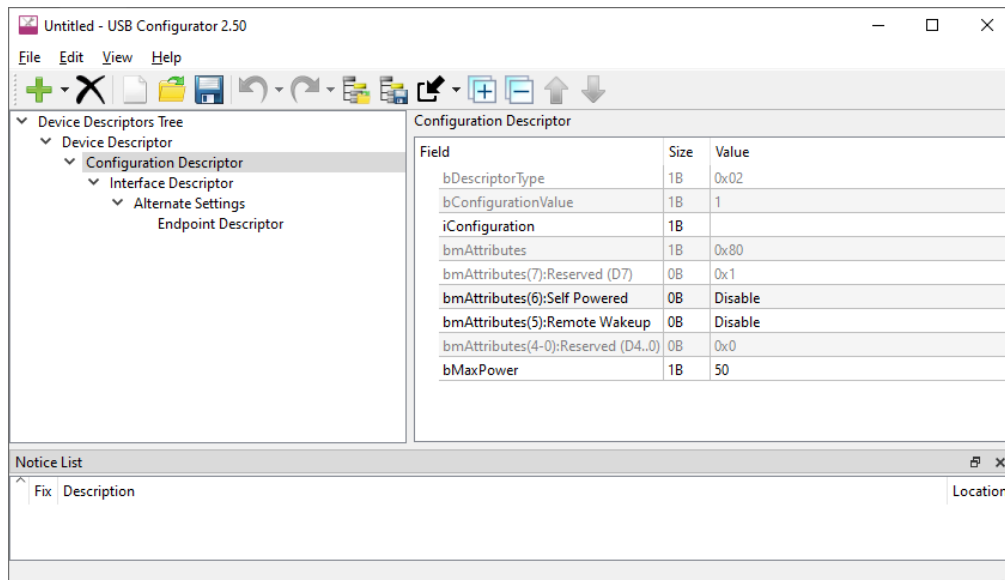## 2.8.2.2 USB configurator

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

2.9 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
- Enter `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.
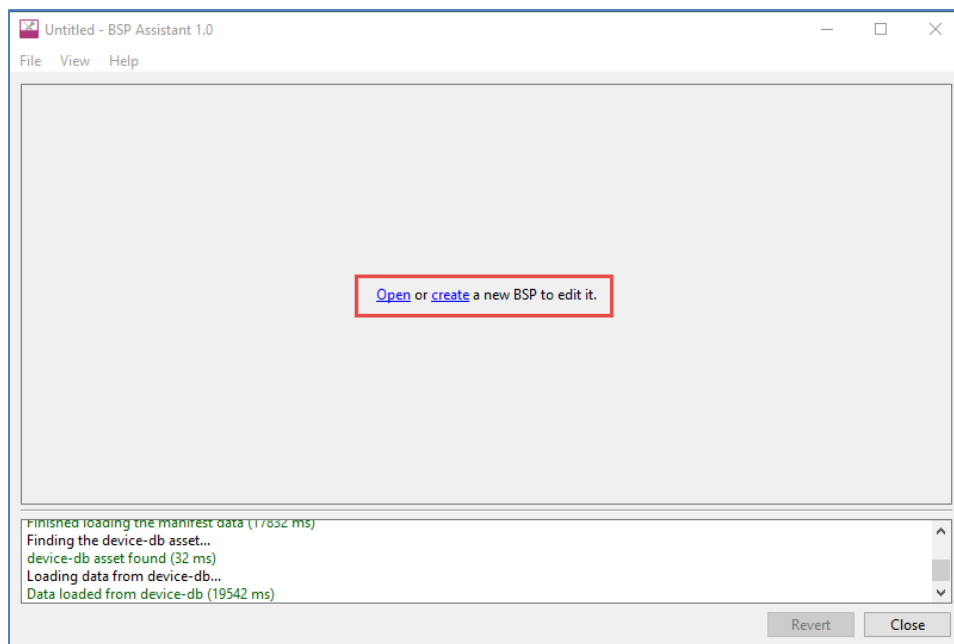
2.9.1.2   Open existing BSP

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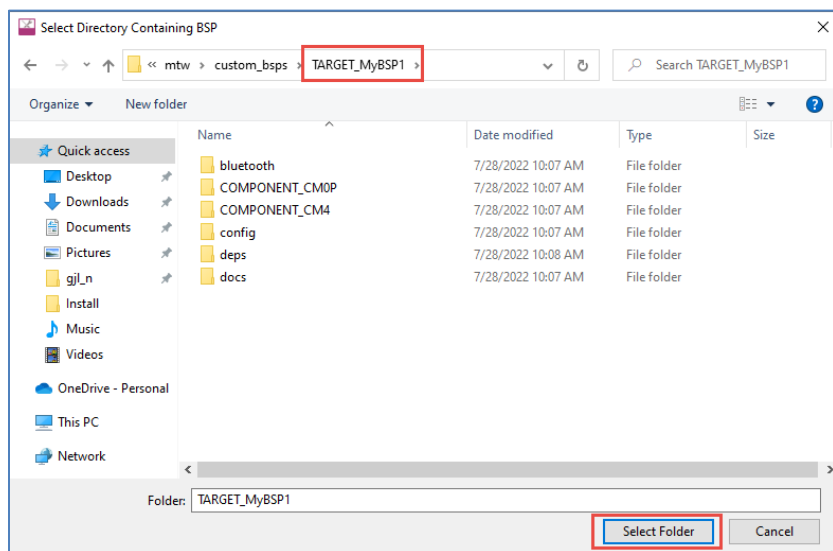
*Note:*   *If the specified BSP has a .git directory that points to an Infineon repository, a warning dialog will indicate that you might want to use the New… dialog to use the BSP as a template to create a new BSP in a new location that is not connected to Infineon's Git repositories before customizing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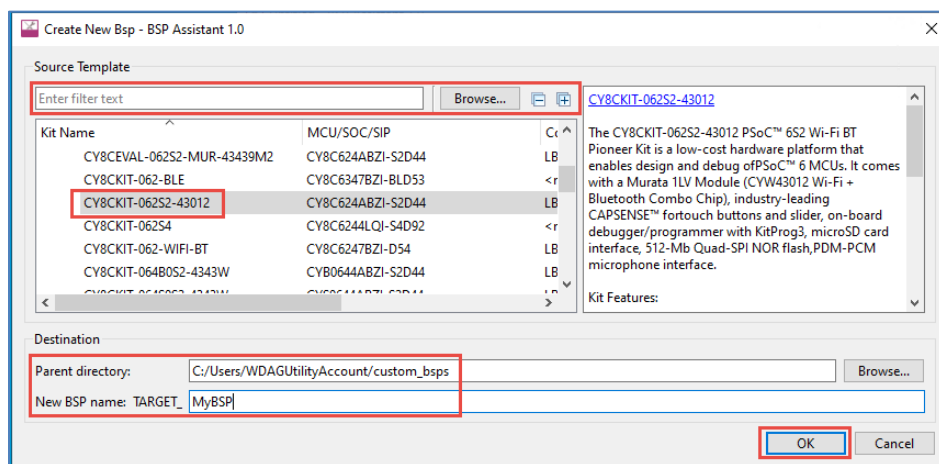

Use the filter box and buttons to expand/collapse all of the categories and make it easier to find what you are looking for.

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, Configuration, Dependencies, and Capabilities. Each of these can be collapsed or expanded (they are all shown collapsed).
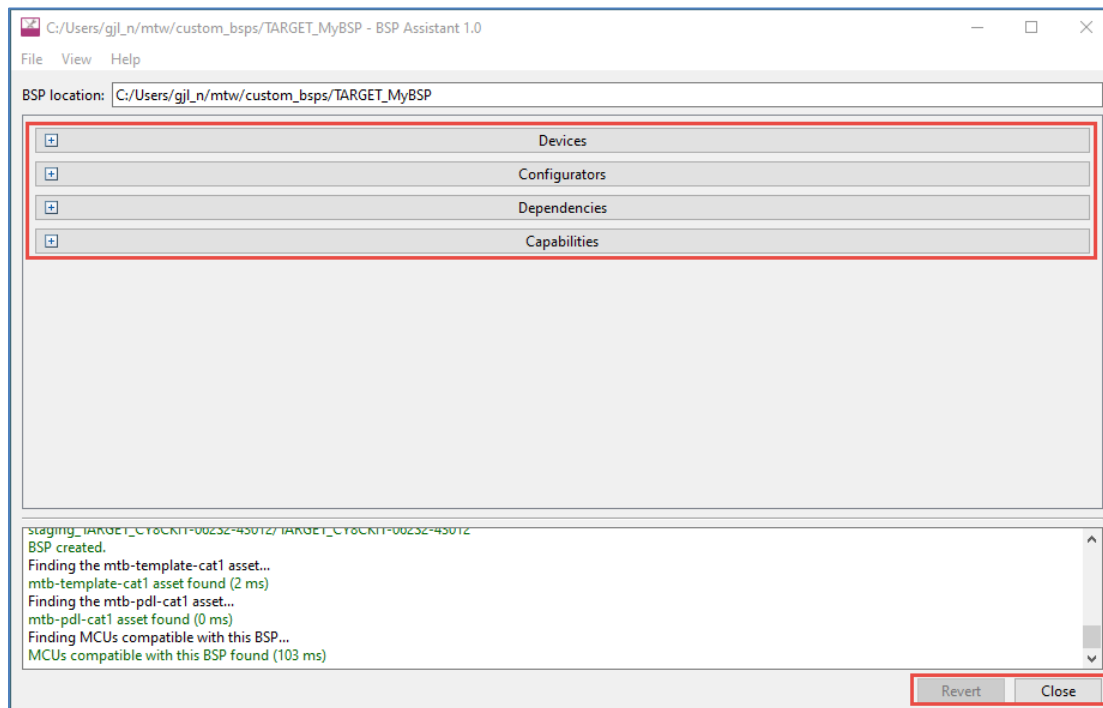
Devices

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 b*sp.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*.
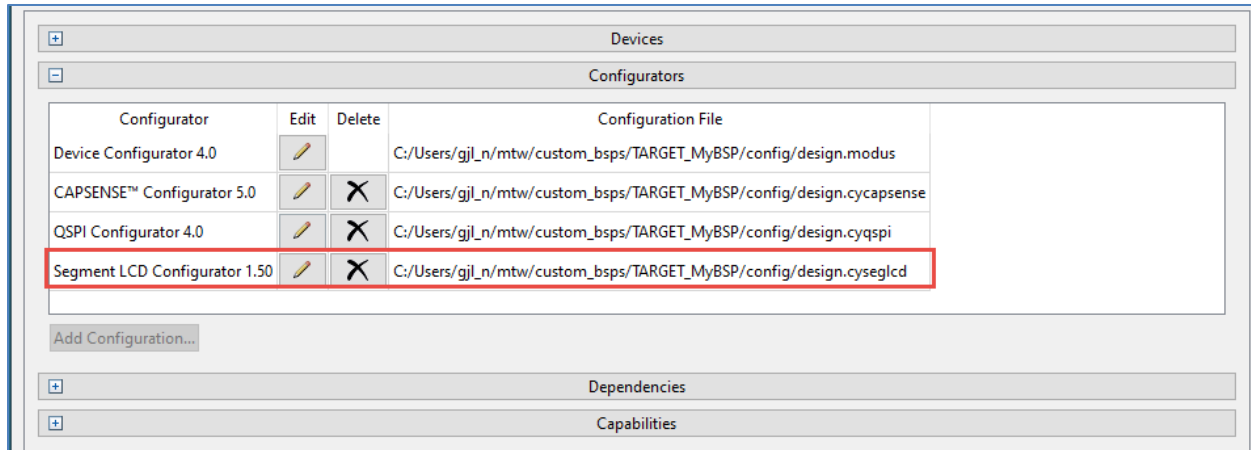
Configurators

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).

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 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 and 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).
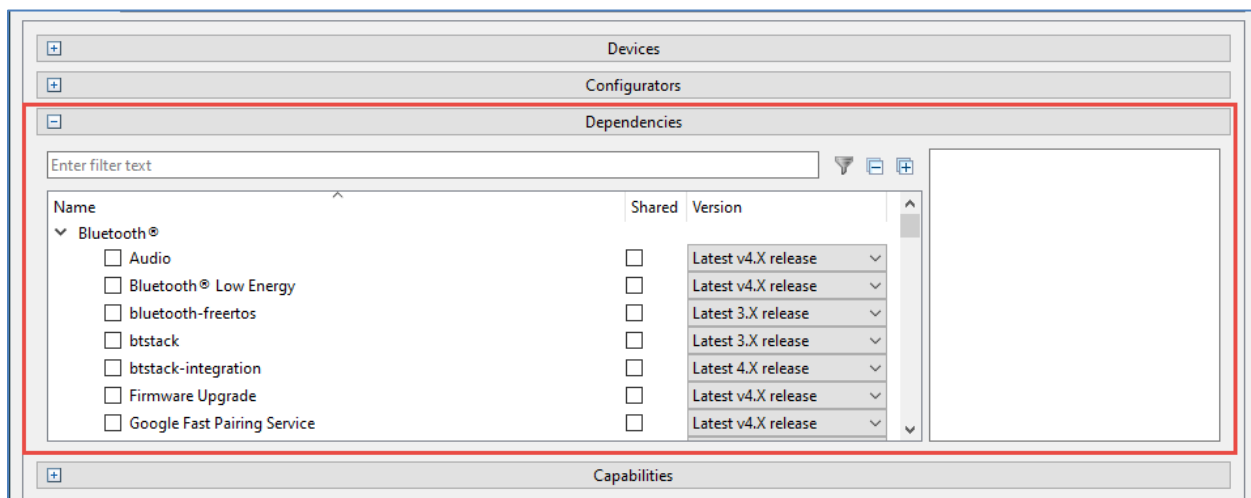
Select the **Filter** button to show only those libraries that are currently listed as dependencies. You can also use the filter box to search for libraries or use the plus and minus buttons to expand or collapse all categories.
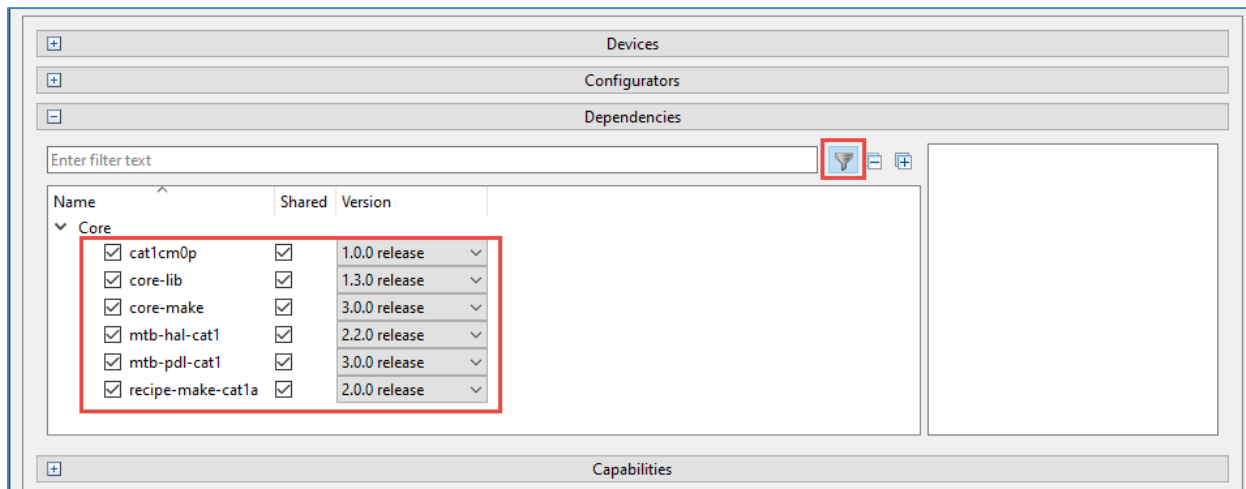
Note: The default versions for the required libraries are the latest release versions at the time the BSP was created. If you want the libraries to be set to the latest versions when the BSP is used to create an application, you can change the versions to "Latest V.X release" where "V" is the major version 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.
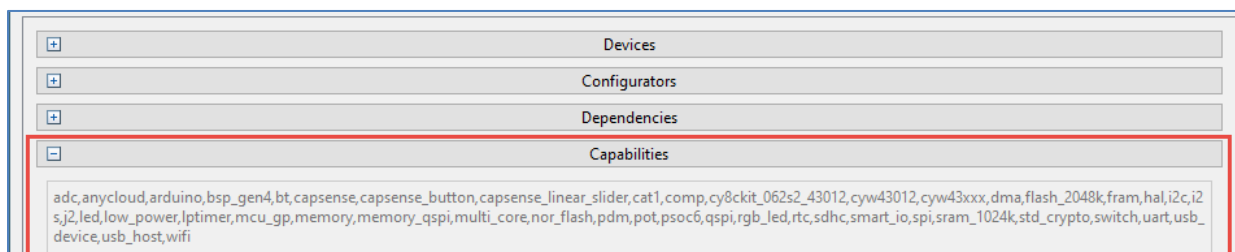
This list is used by the Project Creator to show template applications (i.e. code examples) that are compatible with your BSP. It is also used by the Library Manager to show libraries that are compatible with your BSP.

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 located in the section `opt > props > capabilities`. For example:

```
"opt": {
    "props": {
        "capabilities": [
            "adc",
            "anycloud",
            "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.
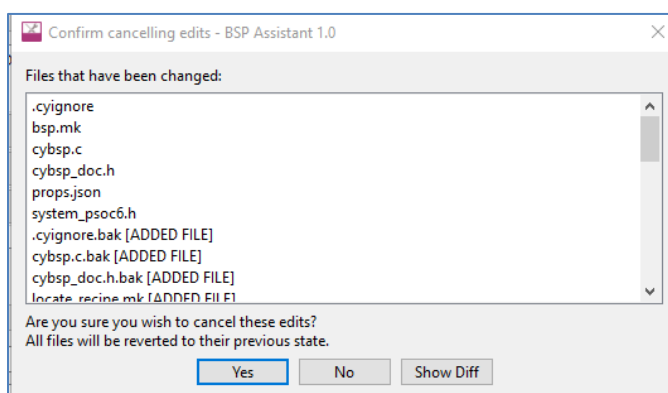
If you click **Yes**, the BSP will be restored to the state when you opened the tool for this session or created the current BSP. You cannot undo a revert operation.

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. Apply 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. You must complete this step 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. The BSP will be copied into the application's *bsps* directory.
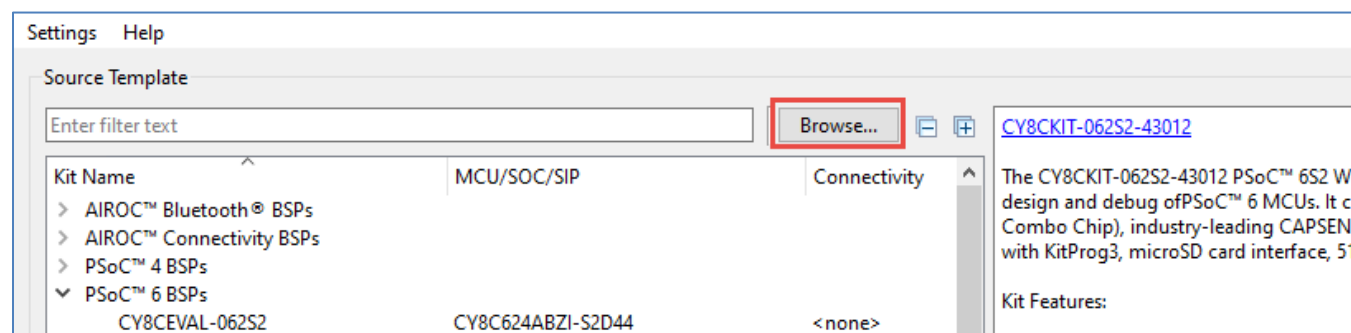
2.9.2.2 Using a custom manifest for the BSP

As stated above, the browse mechanism is 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 2.12 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 create the application with a local BSP by browsing to the custom BSP as described above. Once the application is created, you can delete the local copy of the BSP from the application. 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.*

2.10 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.

2.10.1 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.
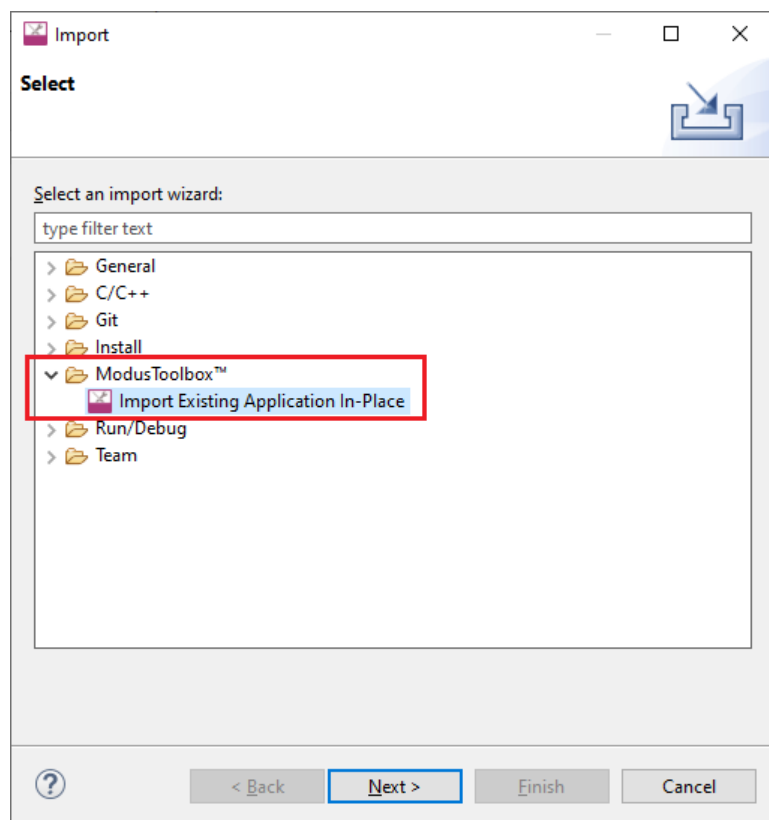
Note: *This feature imports the application so that the Eclipse workspace can view it but it does NOT copy it into your workspace directory. The application must be put in the workspace directory before importing.*

After clicking **Next >**, browse to the location of the application's directory, select it, and click **Finish**.
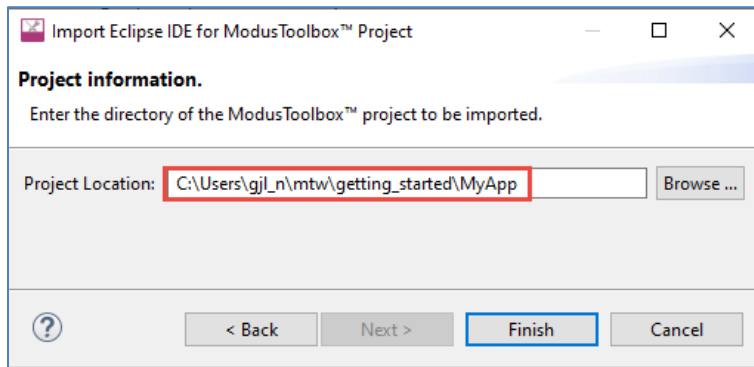
Note: *This import mechanism runs* `make eclipse` *plus additional commands to help the application run smoothly in the Eclipse IDE for ModusToolbox™.*

2.10.1.1   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.
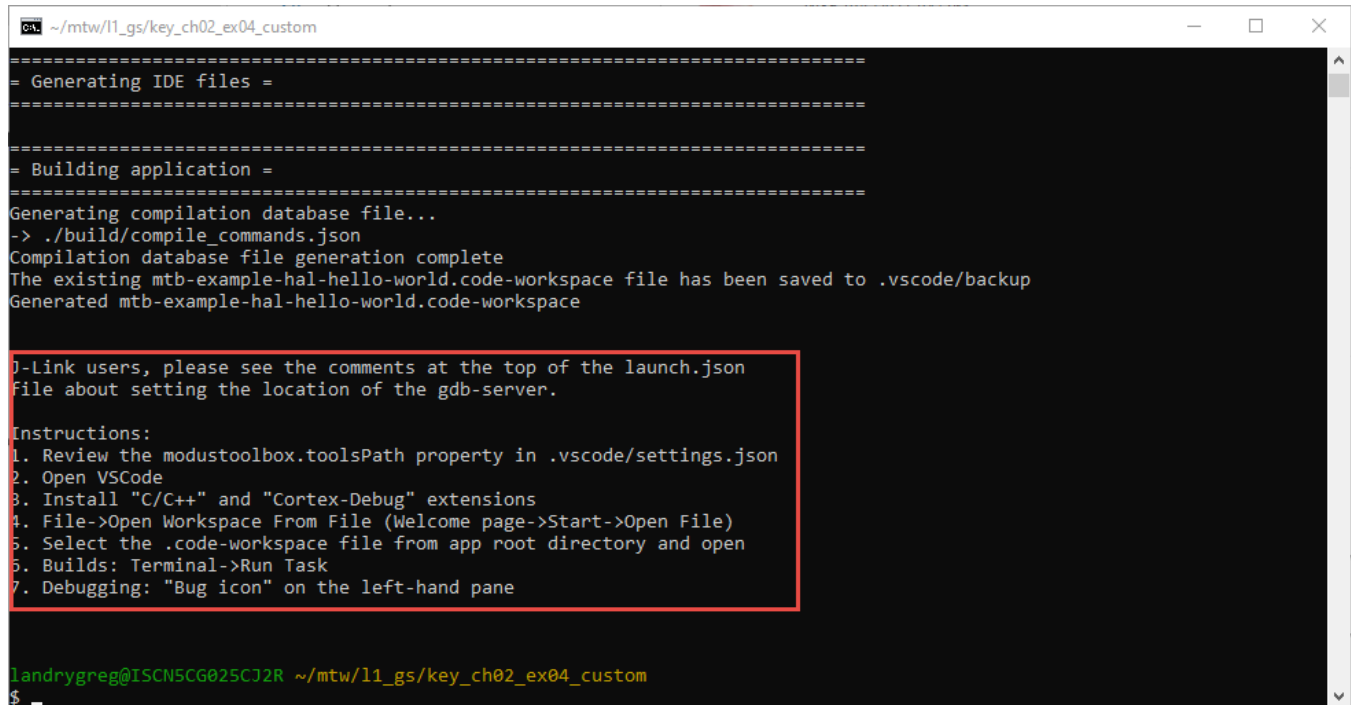
The following list includes some of the scenarios that result in stale or multiple sets of launch configs:

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

2.10.2   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
```


The message near the bottom of the command window indicates the next steps you should 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).
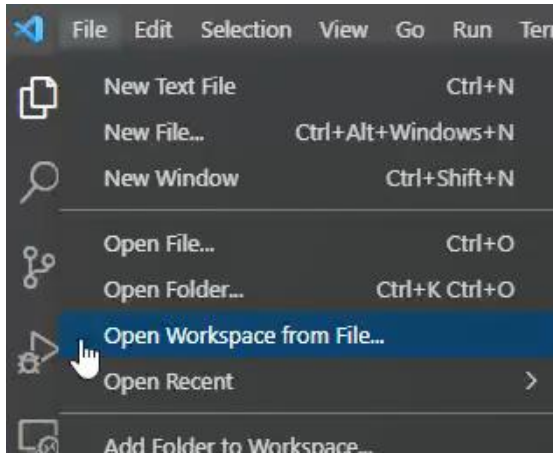
Note: *We'll cover VSCode workspaces in more detail in a minute.*

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.
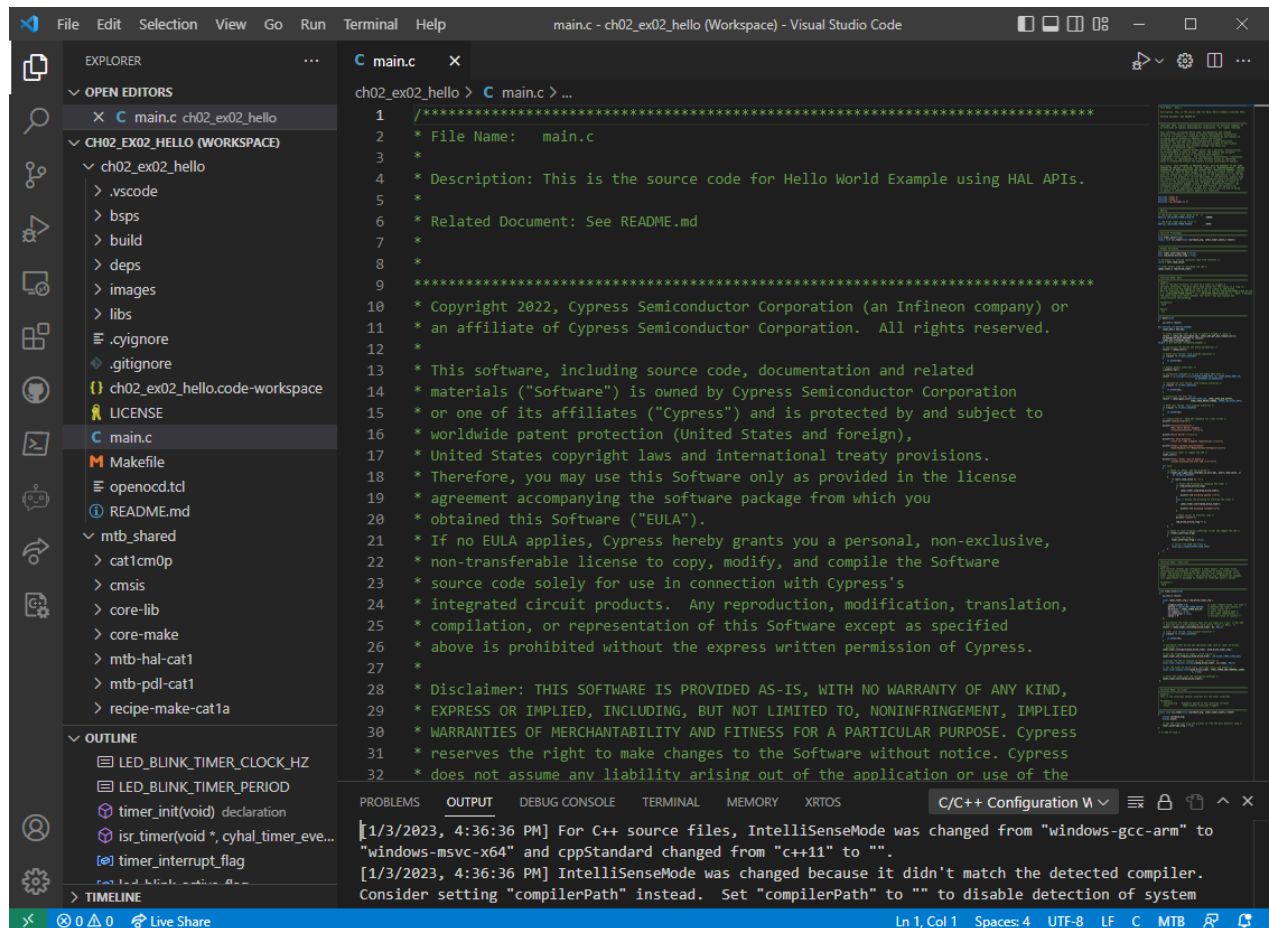
6. In order to build your project, make sure your application is selected and choose the menu item **Terminal > Run Task…**
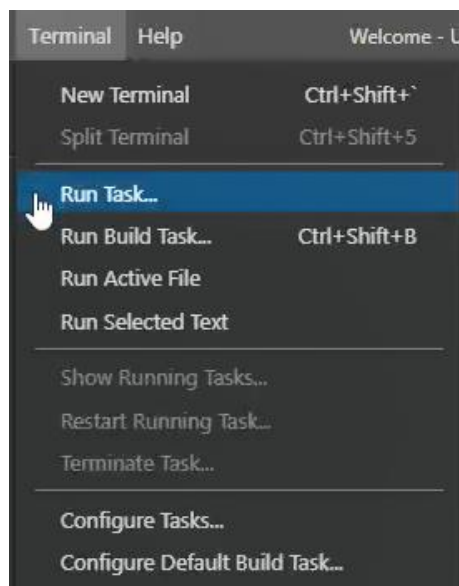
Then, select **Build <appname>**. This will essentially run `make build` at the top level of your project.
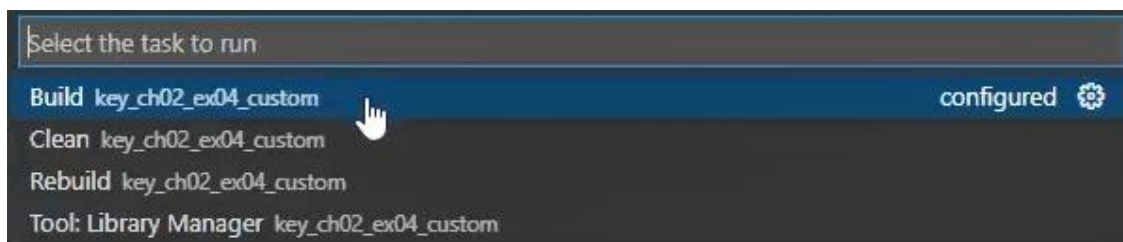
Note: *You can also access the Library Manager from the Task list.*

You should see the normal compiler output in the terminal window at the bottom of VS Code:
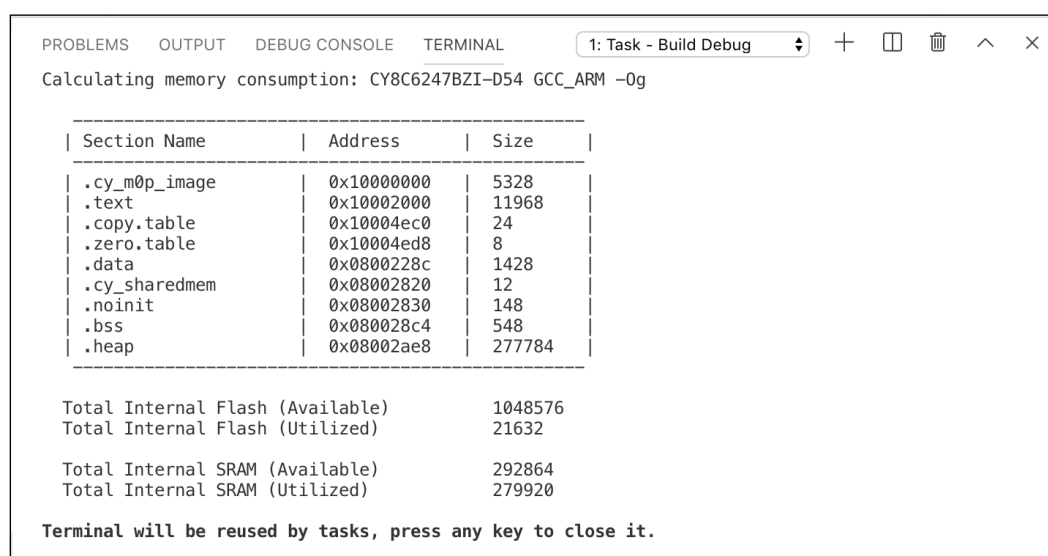
7. To build and then program the kit or run the debugger, first make sure you have installed the "Cortex-Debug" VS Code extension from the marketplace. See the software installation exercise in the introduction chapter if you didn't do this when you installed VS Code.

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 select from the drop-down, click the green **Play** button to start the operation.
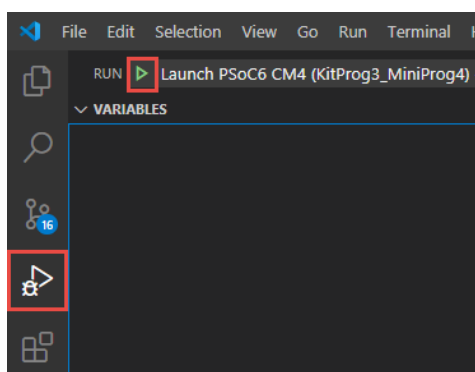
If you chose one of the debugging options, after clicking **Play**, your screen should look something like this. Your application has been programmed into the chip (if you chose the Launch option). The reset of the chip has happened, and the project has run until *main*. Notice the yellow highlighted line. It is waiting on you. You can now add breakpoints or get the program running or single step.
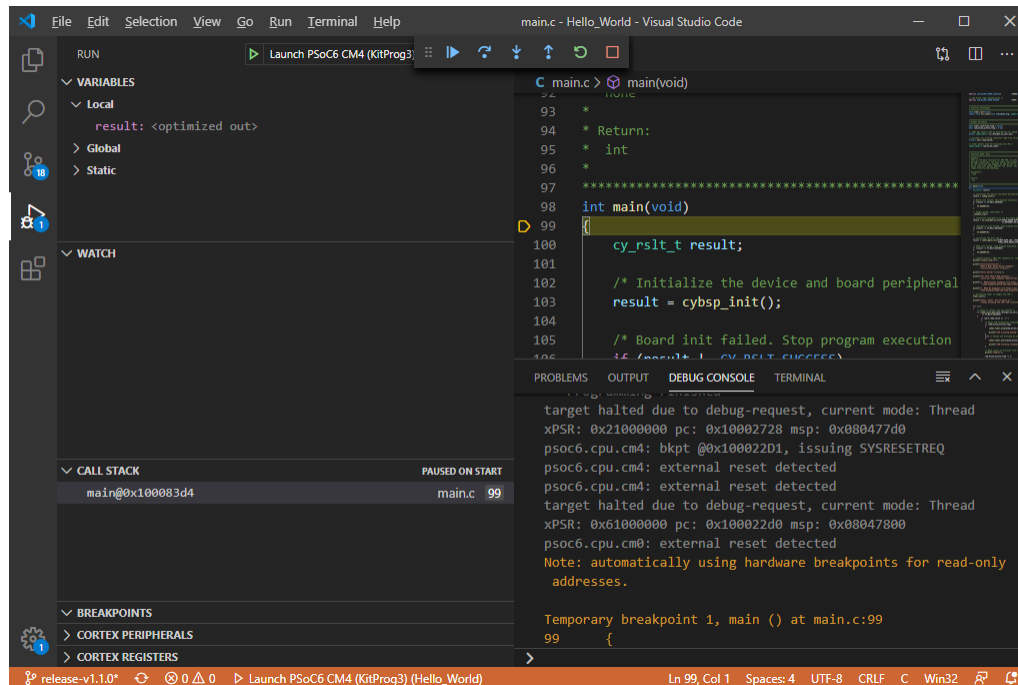
2.10.2.1   Using the ModusToolbox™ Assistant

If you installed the ModusToolbox™ Assistant extension, then you can access other ModusToolbox™ tools and documentation using that extension. Along the left panel, click on the icon for the assistant to open a list of tools that are similar to the Quick Panel in the Eclipse IDE for ModusToolbox™. The VS Code assistant even has some additional information about the application's libraries and variables that is useful.
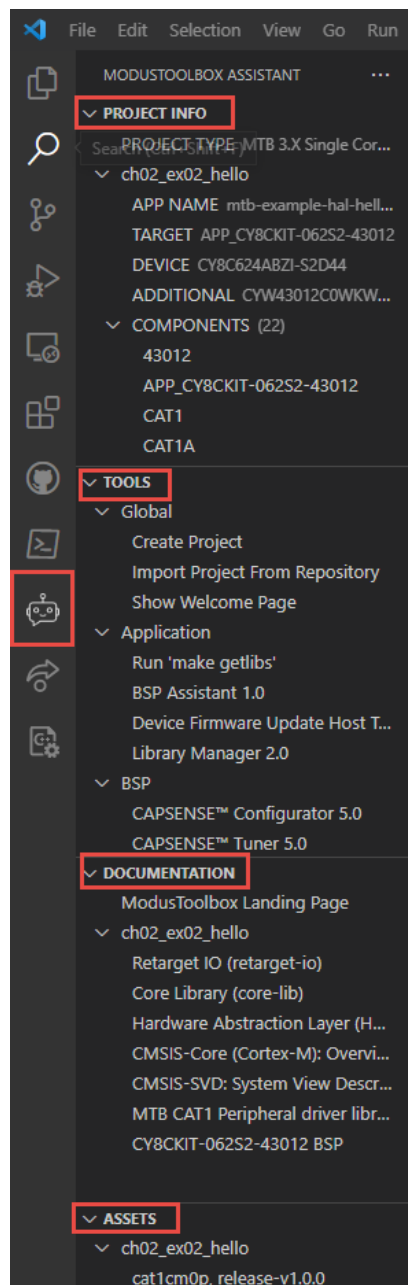
Note: *Each individual panel may have scroll bars so you may need to scroll to see all of the items. For example, the Device Configurator can't be seen in the image above because it is lower down in the Tools section. You can minimize some of the panels or change the sizes if you want to see more items in a panel.*

Note: *The ModusToolbox™ Assistant extension is not Infineon software.*

2.10.2.2   Using Modus-Shell in VS Code

If you are using Windows and you want the command line modus-shell terminal to be available as one of the terminals that you can open from inside the IDE, just add the following to the *settings.json* file that is located in the application's *.vscode* directory:

```
"terminal.integrated.profiles.windows": {
        "ModusToolbox-shell": { // name of profile
            "path": "cmd.exe",
            "args": [
                "/C",    // execute a command
                "D:\\ModusToolbox\\tools_3.0\\modus-shell\\Cygwin.bat"
            ],
            "color": "terminal.ansiMagenta",
            "overrideName": true,
            "icon": "pencil"
        }
    },
```
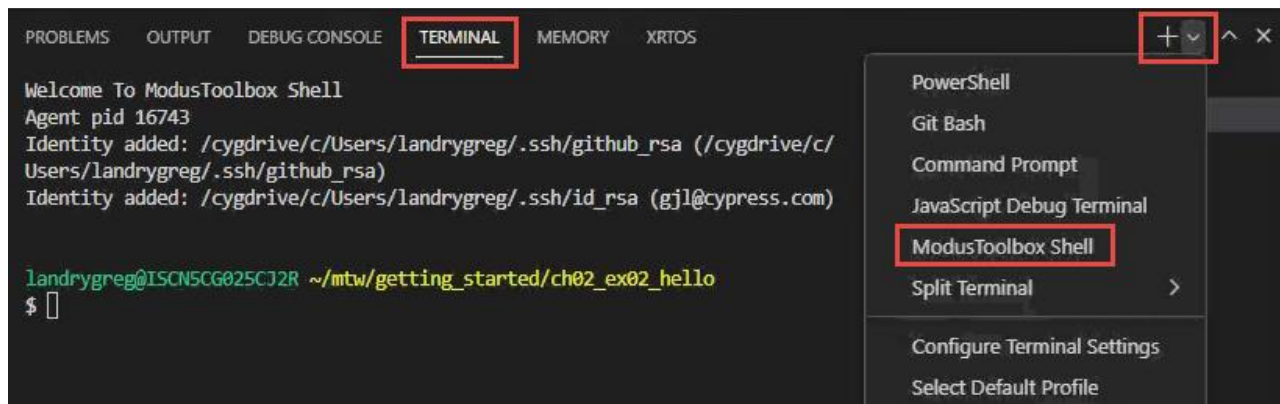
Note: Add the code above just after the initial brace at the top of the settings.json file.

Then, you can open modus-shell by opening the terminal tab, clicking the arrow next to the "+" button, and selecting **ModusToolbox Shell**.

2.10.2.3   Using workspaces in VS Code

Rather than just opening the application's directory in VS Code, you should open a workspace. With this option, the workspace knows that the code in *mtb_shared* is part of the application so it will be shown in the file Explorer window and used for Intellisense operations.

Note: Keep in mind that a workspace in VS Code is not the same as a workspace in Eclipse. 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 file 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:
    o From the command line: `code <workspace_name>`
    o 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 you can use **File > Add Folder to Workspace…**) if you have the workspace open in the GUI.
- If you add a library to the application, it is a recommended to build the application before adding any code for the new library. This will allow Intellisense to find code in the new library.

2.10.3 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. Similarly, 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**.
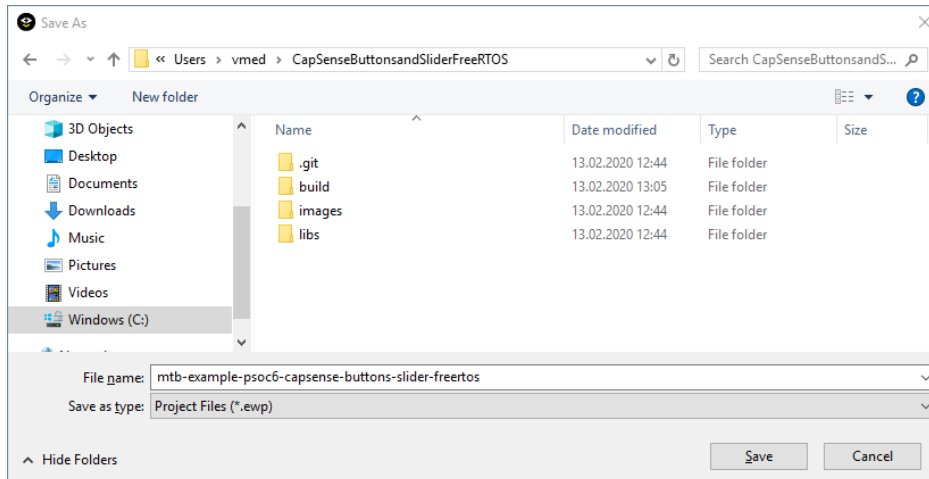
4. After the application has been created, select **File > Save Workspace**, enter an arbitrary workspace name and click **Save**.
5. Select **Project > Add Project Connection** then click **OK**.

6. On the Select IAR Project Connection File dialog, select the *.ipcf* file and click **Open**:
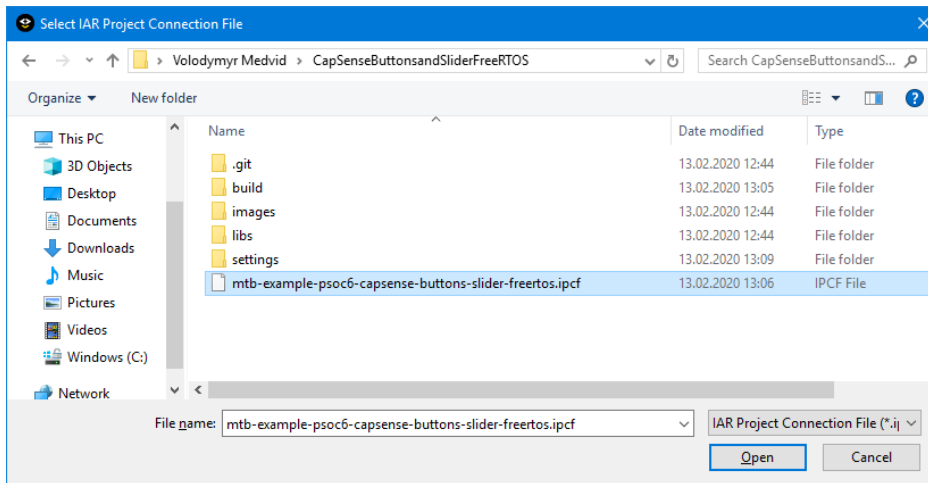
7. On the main menu, Select **Project > Make**.

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.

2.10.4 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. If you want command line builds to use the ARM toolchain, you can edit the Makefile's TOOLCHAIN variable. 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. Likewise, changes in the Makefile will not affect the IDE builds.*

This command 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 complete this action, you may see a dialog similar to this:
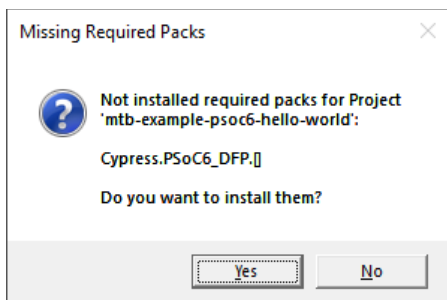
Click **Yes** to install the device pack. You only need to do this once.

2. Follow the steps in the Pack Installer to properly install the device pack.
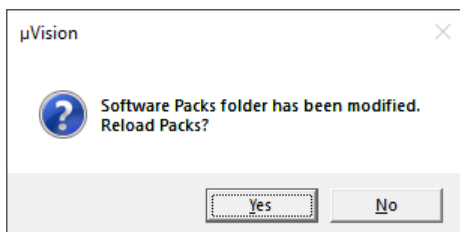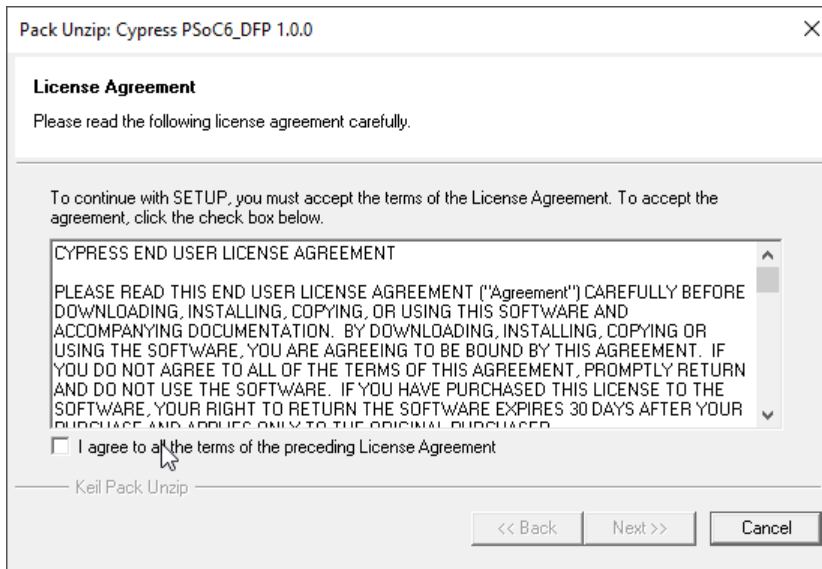
3. When complete, close the Pack Installer and close the Keil µVision IDE.
4. Then double-click the *.cpdsc* file again and the application will be created for you in the IDE.

At this point, the application is open in the Keil µVision IDE. There are several configuration options in µVision that will not be discussed here. For more details about using µVision, refer to section 7.3: "Exporting to supported IDEs" in the ModusToolbox™ tools package user guide.

2.11   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.

2.11.1   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 Manger and clicking the **Update** button. Either option 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 *GeneratedSoruce* files are out of date.

2.11.2   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. Finish with your changes. Then check in your updates after completing 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, but it does NOT create a new copy of the application in your workspace. As such, you should move it to your workspace first before importing it.*

2.11.3   Manual file copy

If you are not using version control software, you can copy a complete application directory from one user to another. You can also choose to exclude the directories listed in 2.11.1. since the libraries can be downloaded by running `make getlibs` or running the Library Manager and clicking **Update**. The other files will regenerate once the application is built.

2.11.4   Using Eclipse archive files

The Eclipse IDE for ModusToolbox™ includes 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**.
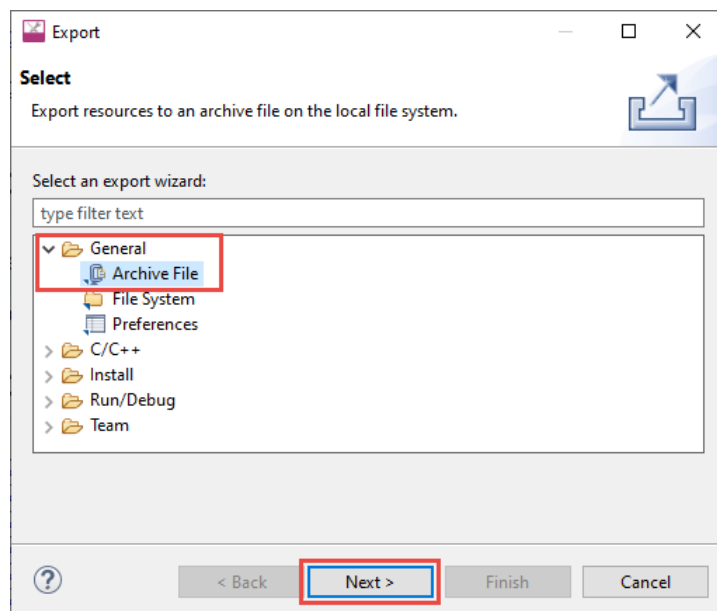
When you get to the next page, you can select the specific application that you want to archive along with the files required.

Note: *You don't need to archive mtb_shared, libs, Generated Source, or build directories.*
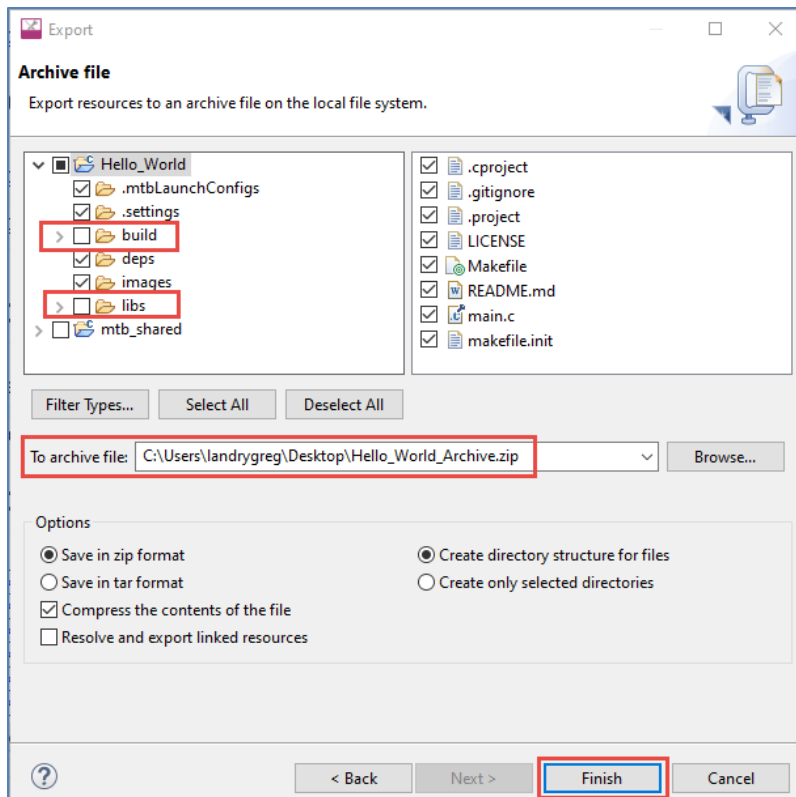
For multi-core applications, the Eclipse IDE export function shows the project folders for each core alongside the main application folder. These project folders are included in that main application folder, so you should exclude them from the export process as well. On the Export dialog, select only 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 select **Finish** to begin the import process. This will take a few moments before the application displays in the Eclipse IDE Project Explorer.

2.12 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.

In order to use your own examples, BSPs, and middleware, you need to create manifest files for your content and a super-manifest, pointing 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 specifying 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. However, 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 syntax examples 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

2.13   Network considerations

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

2.13.1   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. This method also allows you 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, it first looks for a remote manifest file. If that file isn't found, you will not be able to move 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/*.
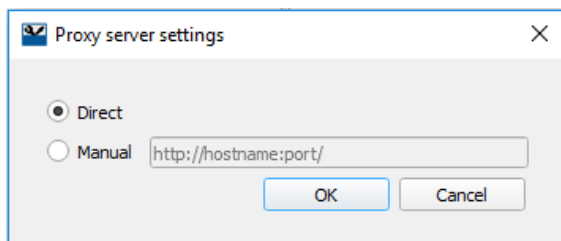
Once you set a proxy server, you can simply enable/disable it by selecting the Manual or Direct options. There is no need to re-enter the proxy server every time you connect to a network requiring that proxy server. The tool will remember your last server name.

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

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

2.13.2   Offloading manifest files

In some locations, `git clone` operations from GitHub may be allowed while raw file access may be restricted. This prevents manifest files from being loaded. In this scenario, 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

2.13.3   Offline content

In the case where network access is impossible, Infineon provides a zipped-up bundle with all of our GitHub repos to allow you to work offline, (i.e. if you are on an airplane or don't otherwise have access to GitHub). To set this up, you must have access to infineon.com in order to download the zip file. After downloading the file, 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:
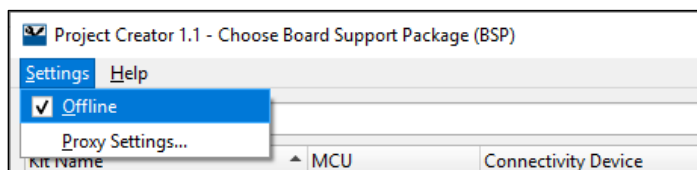
If you try to open these tools without a connection, a message will ask you to switch to Offline Mode.
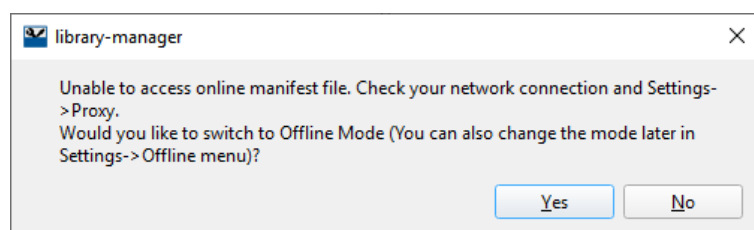
2.14   Compiler optimization

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 it 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.

2.15   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, meaning that it allows you to type into the serial terminal emulator on the PC and send those messages to the PSoC™.*
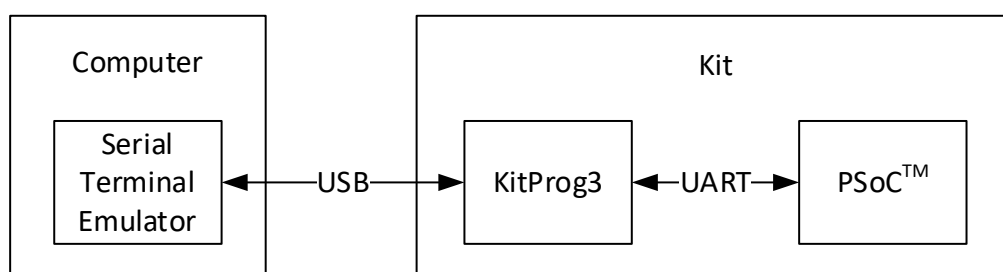
There are various serial terminal emulators available for use. On Windows and Linux, a program called *PuTTY* is typically used. On MacOS, you can use a program called *screen*. If you are familiar with any other serial terminal emulator, feel free to use that instead.

Note: *You can even run a terminal window right from the IDE, if you are using the Eclipse IDE for ModusToolbox™.*

2.15.1   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.*
```

2.15.2   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 step ensures 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.*
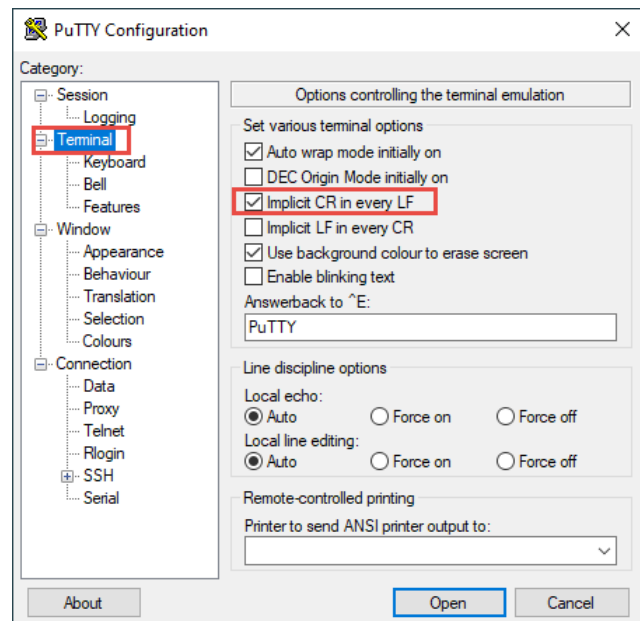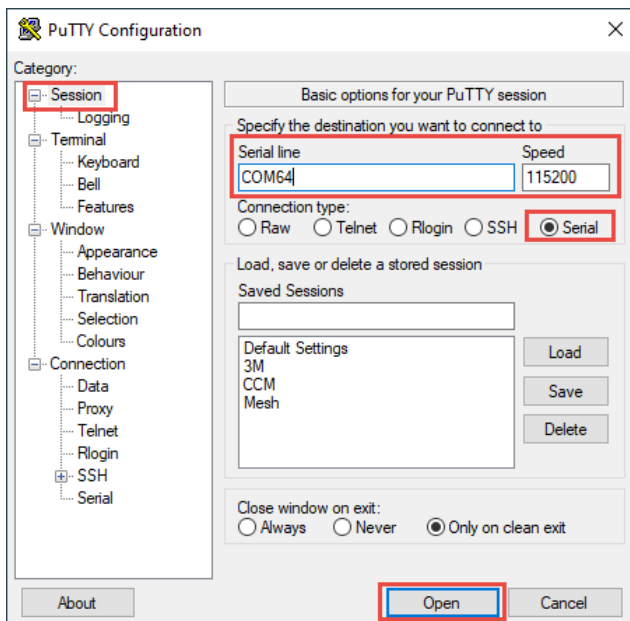
Once it is running, you can right-click on the top banner to show a list of useful commands. You can change settings, clear the screen, and even restart the session of the kit if it was disconnected.
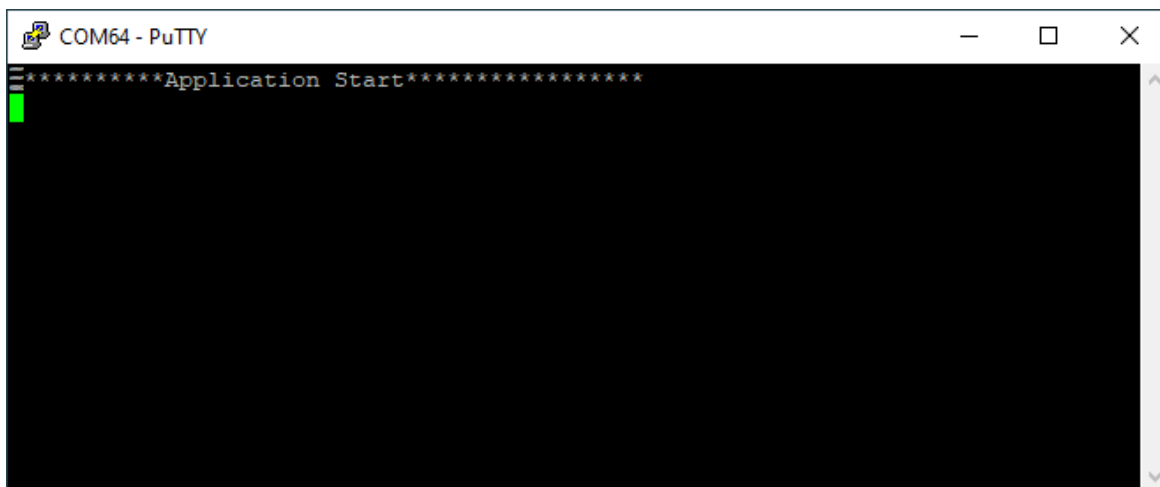
2.15.3 Screen (MacOS)

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
```

2.15.4 Eclipse IDE for ModusToolbox™ built-in terminal

If you want to use the serial terminal capability in ModusToolbox™, first navigate to the **Terminal** tab in the bottom panel and click the button to open a new terminal.
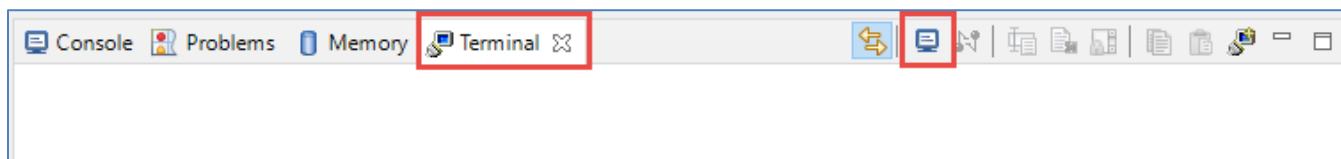
In the window that appears, select **Serial Terminal,** then select the correct port, and other settings. Finally, click **OK**.
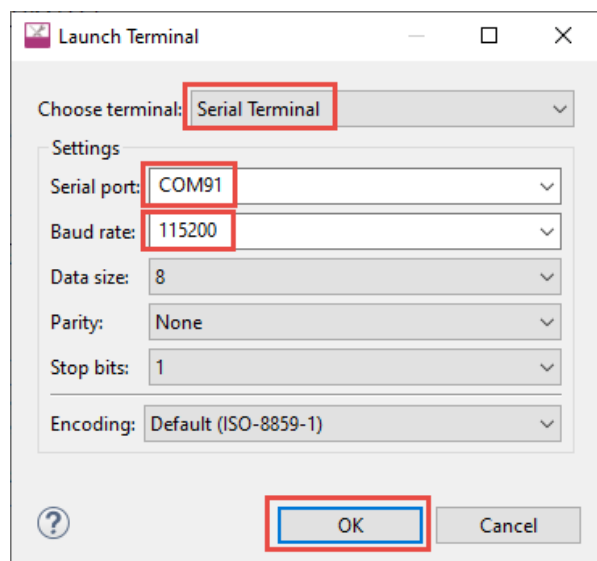
Once you start it, the output will be shown in the Terminal tab in ModusToolbox™. Click the **X** on the tab to close the serial terminal.

Note: *Make sure not to select 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.*

2.16 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 applications you created using ModusToolbox™ versions 2.2 through 2.4 are fully functional within 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 for the application	Specified by a *.mtb* file in the *deps* directory	Specified by the `TARGET` variable in the *Makefile*. There is no *.mtb* file in the *deps* directory.
Default BSP type	Git repo, to make changes without dirtying the repo requires custom BSP	Application-owned, can be directly modified
Local BSP location	Under the *libs* directory	Under the *bsps* directory
BSP Makefile	Makefile name is *<BSP_Name>.mk*	Makefile name is *bsp.mk*
BSP capability	BSP has a capability of `bsp_gen3` or earlier.	BSP has a capability of `bsp_gen4` or later.
BSP properties	BSP does not contain a *props.json* file. The version is contained in *version.xml*.	BSP contains *props.json* with information such as capabilities, dependencies, and version. The BSP does not contain a *version.xml* file.
CAPSENSE library	CAPSENSE library is included as a dependency of the BSP	CAPSENSE library is NOT included as a dependency of the BSP. It is included by all apps that use CAPSENSE.
design.modus file location	*libs/COMPONENT_BSP_DESIGN_MODUS* subdirectory	*bsps/config* subdirectory
Makefile `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 *COMPONENT_CUSTOM_DESIGN_MODUS* and are compiled into the application via *Makefile* COMPONENT settings.	Custom files are in the application directory *templates* and are copied into the BSP during application creation.

This training focuses on the 3.x BSP/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 by following the steps outlines in 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.*

2.17   Exercises

For the following exercises, the CY8CKIT-062S2-43012 kit is used, but you can choose 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 is shown in the following exercises, 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 theEclipse IDE 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*.
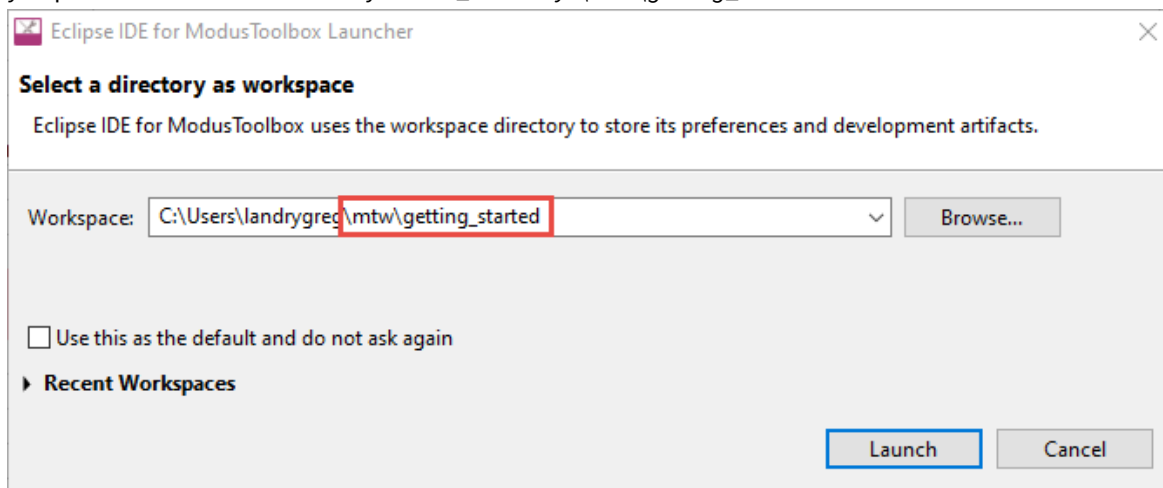
   You can change workspaces at any time and you can have as many workspaces as you want.

2. On the Quick Panel, click the **New Application** link or select either **File > New > ModusToolbox™ Application**. Either option launches the Project Creator tool.
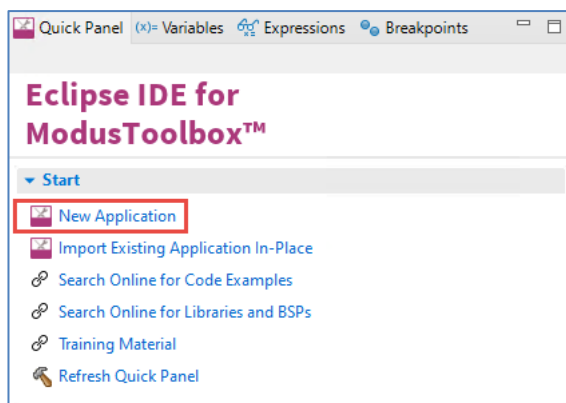
3. Choose the correct development kit and click **Next >**. I will use the CY8CKIT-062S2-43012, but you can use whichever kit you prefer.
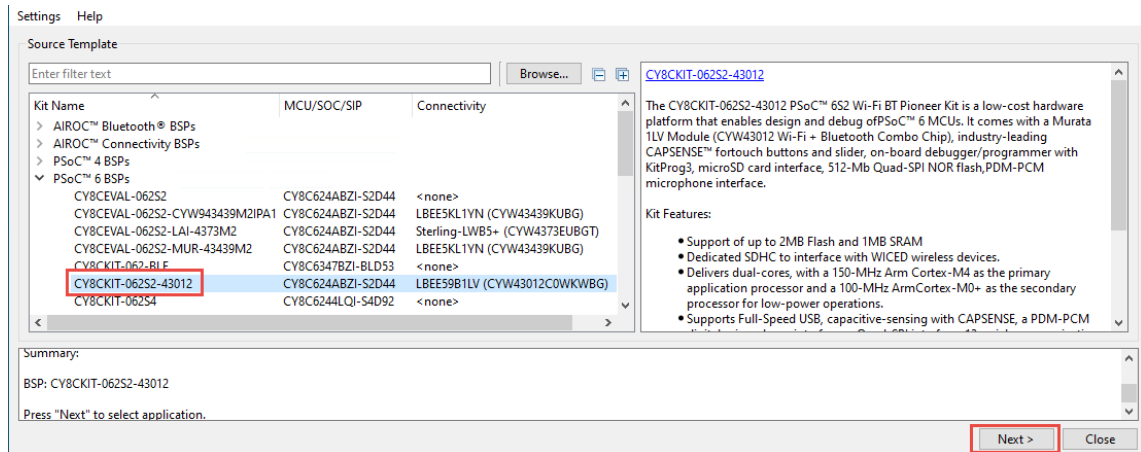
4. Check the box next to *Hello World* from the *Getting Started* category and give your project a name (in this case I'll use **ch02_ex01_hello**). Then, click **Create**.
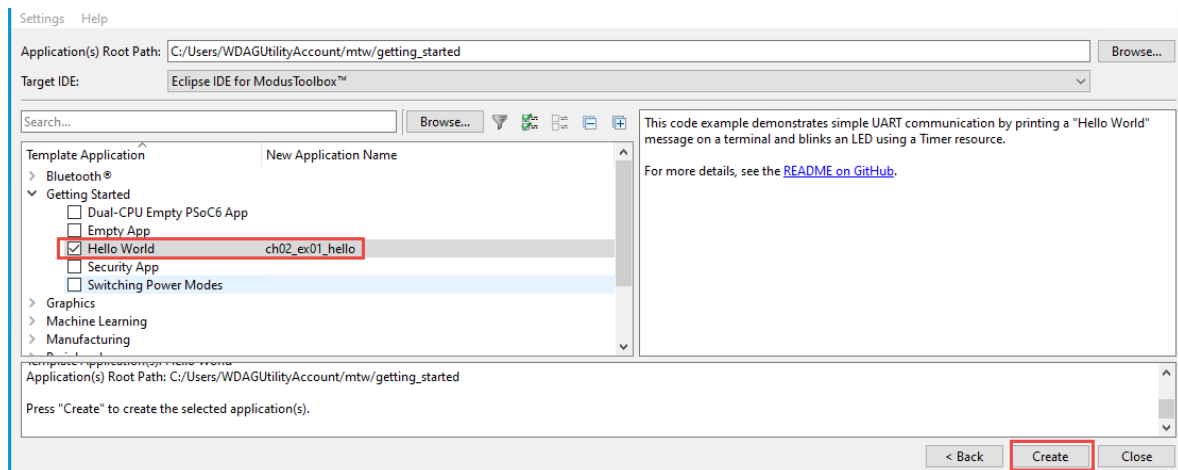
You can check the box for more than one application at a time if you want to create multiple applications in a single operation.

The Project Creator section will go into more detail on application searching and filtering capabilities.

Once the application has been created, the new project creation window will close (unless there are warnings or errors) and the new application will be imported into the Eclipse IDE.
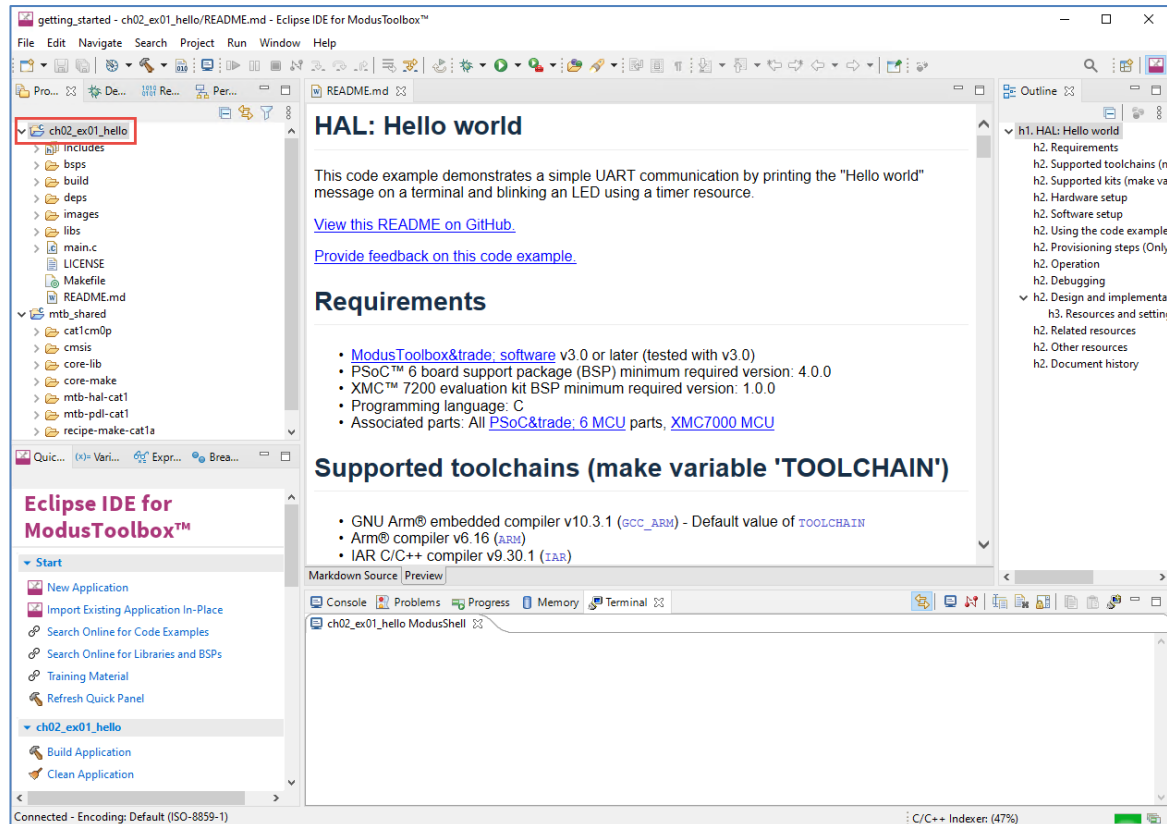
5. The *README.md* file will be open in the main Eclipse window. Read it to see how the example functions.

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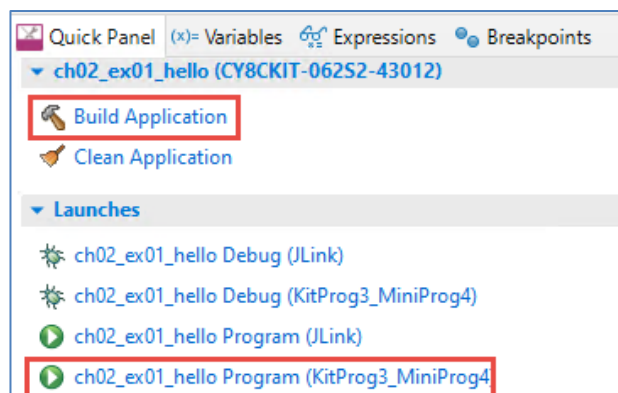
The kit I'm using has a KitProg3 on it so that's the one I will choose.

You should now see a blinking LED.

Note: Some kits have multiple USB connectors. Be sure that your kit is connected to the computer using the connector with the label KITPROG.

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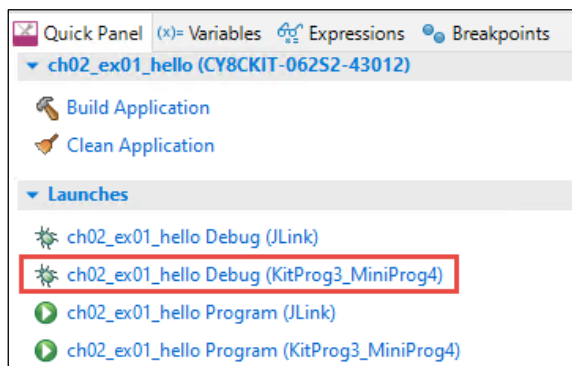
☐ 10. Start/pause execution, single step through the code, and add a breakpoint. See the Eclipse IDE for the ModusToolbox™ User Guide for additional information on debugging.

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 use VS Code instead of Eclipse.

Note: *The instructions ask you to 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.*
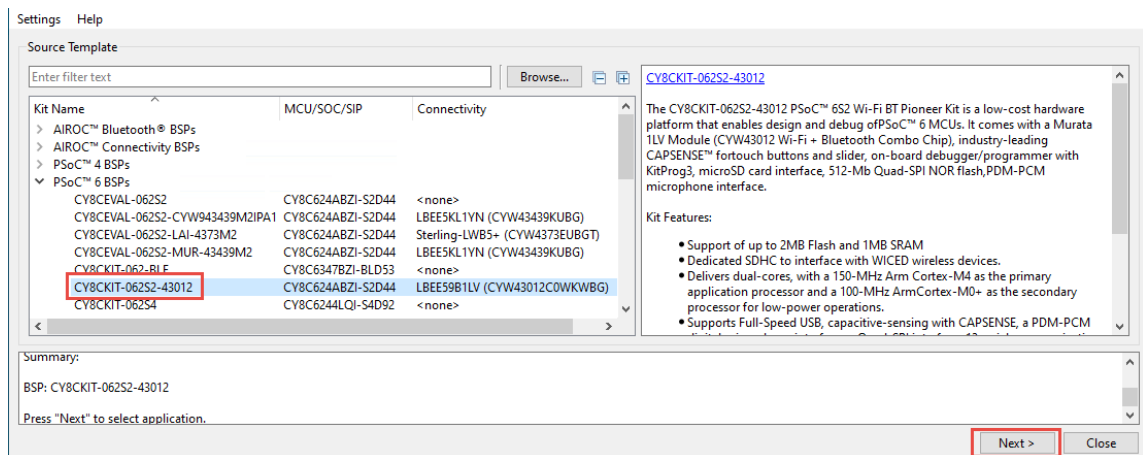
1. Run the Project Creator tool.

   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 existing Application root path so that it will go in the same workspace as the previous exercise, but you can change that if you wish.
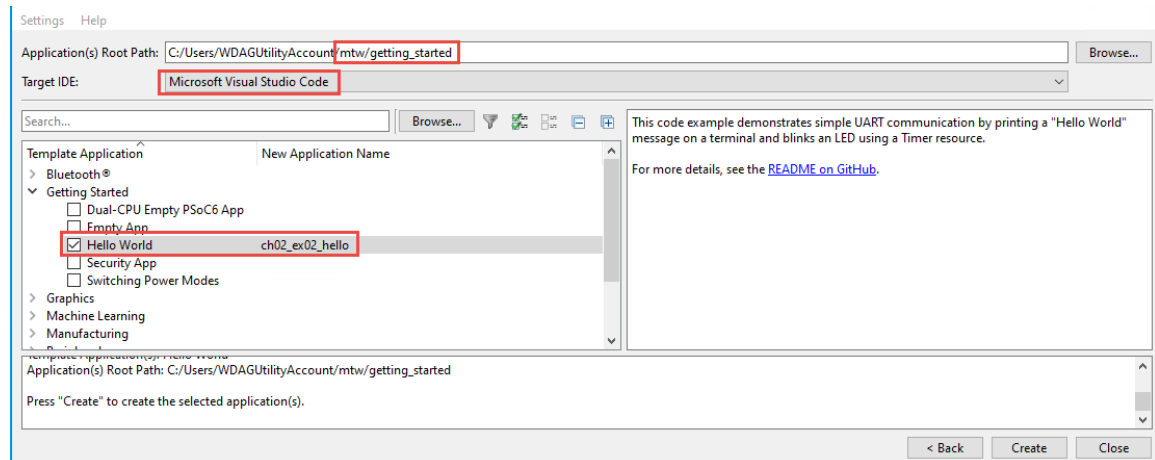

Note: *If you forget to select the **Target IDE** when you create the application, remember that you can run* `make vscode` *from the command line once application creation is complete.*

4. Once the application has been created, click the **Close** button.
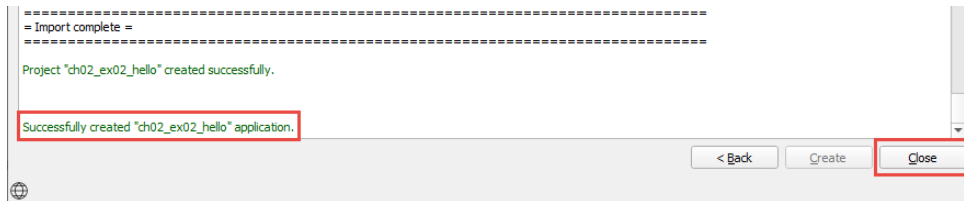
5. Open the application in VS Code and open the workspace. If you open it correctly, you will see the application directory and the *mtb_shared* directory.

Note: If you go to the application's directory from modus-shell, you can run the command code *ch02_ex02_hello.code-workspace*. Alternately, you can run VS Code from the Windows menu, use File > Open Workspace from File and then select the file *ch02_ex02_hello.code-workspace*.
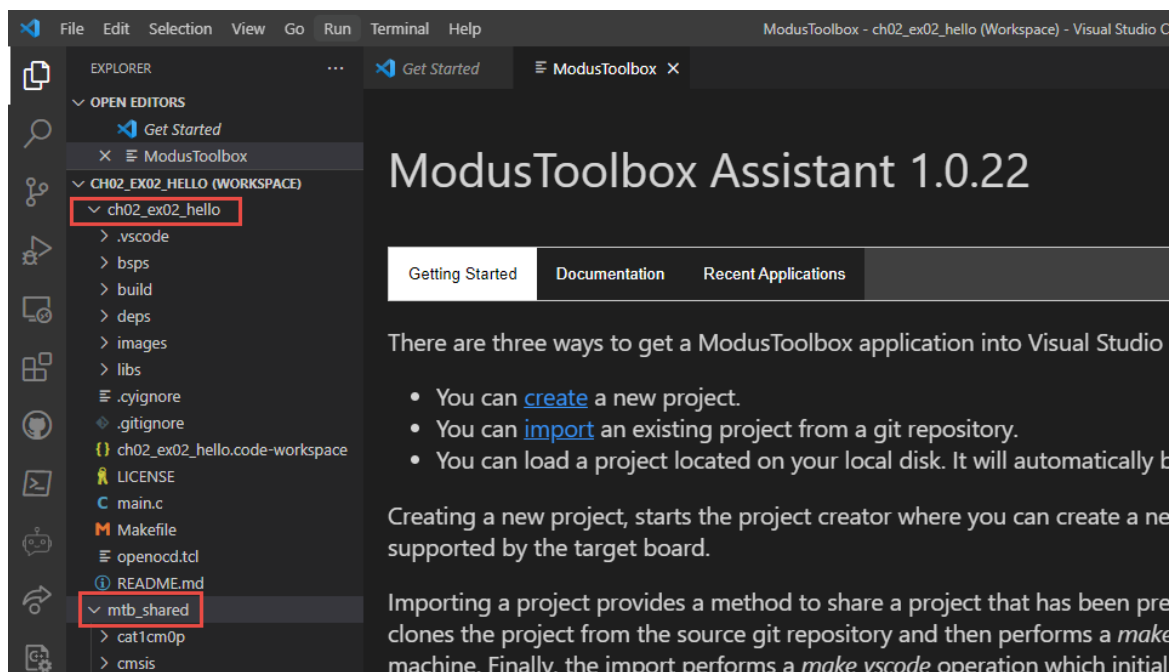
11. Right-click on the *README.md* file and select **Open Preview** to see the formatted Markdown file.

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. At the bottom of the screen, you will see build and program messages in the terminal window.
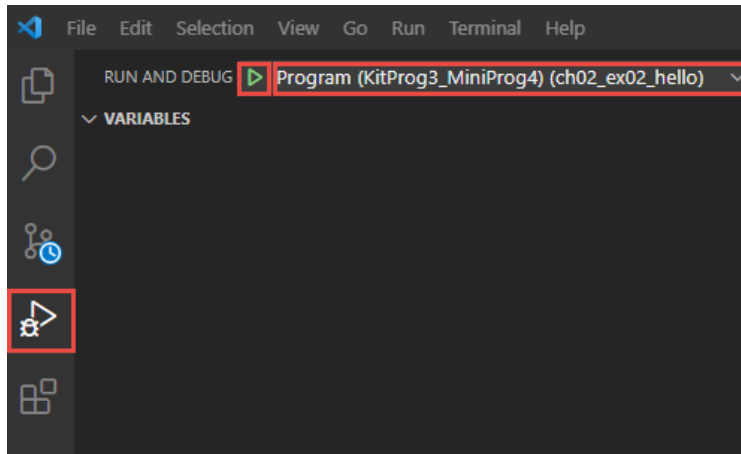
You should now see the faster blinking LED. You can also open a serial terminal window to see messages. Refer to the *README.md* file for the code example instructions.

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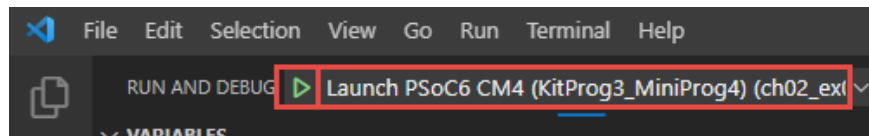
☐   15. Start/pause execution, single step through the code, and add a breakpoint. See the VS Code user documentation for additional information on debugging. Click the stop button when done.

☐   16. To go back to the project explorer window view, click the icon on the left side.
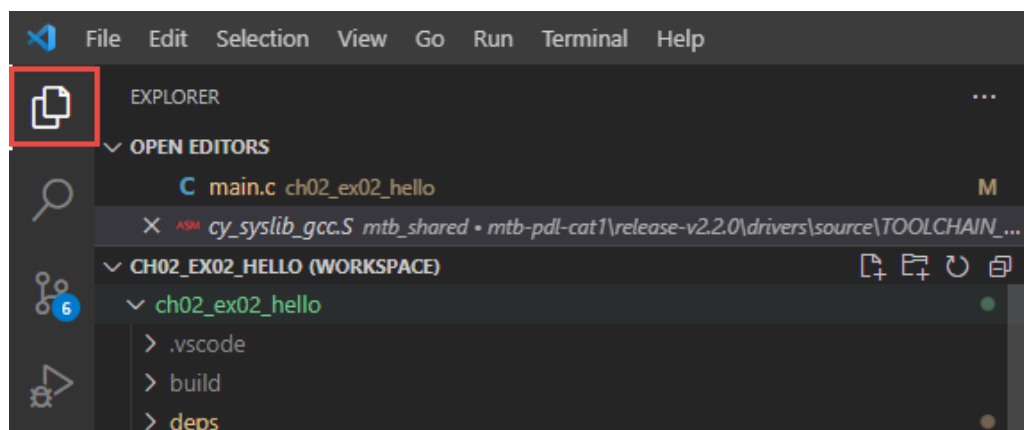
☐   17. Click on the icon for the ModusToolbox™ Assistant extension. Here you can explore all of the different tools, documenation and information available to you.

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.   Switch to the directory for the exercise 1 application.

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]`.
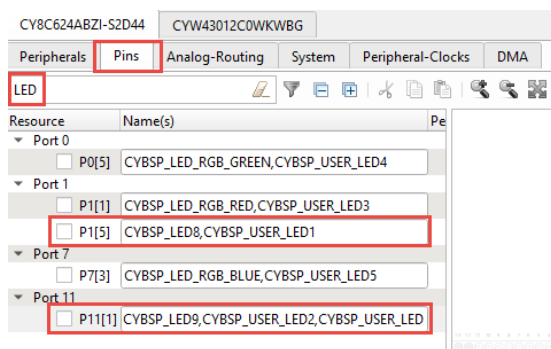
Note: *Don't forget the comma and do NOT add in any spaces as they will be converted to underscores.*

Since the code in *main.c* uses the name `CYBSP_USER_LED`, this change will cause a different LED to blink.

Note: *If you are using the CY8CPROTO-062-4343W kit, the user LED is on pin P13[7]. This kit only has one user LED so you will not be able to move it to a different LED. Instead, you can move it to an unused pin such as P10[0]. No LED will intially blink with this change, but if you want to see it blink you can connect a discrete LED between P10[0] and VTARG. Make sure the anode is connected to VTARG and the cathode is connected to P10[0]. You can also choose to use an oscilloscope to monitor P10[0].*

4. Save the change and exit the Device Configurator.

5. Build and program. Observe the blinking LED is now the other user LED, appearing 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. During application creation, that file is copied into the BSP.*

Exercise 5: Use the BSP Assistant

In this exercise, you will create a custom BSP with the BSP Assistant and then 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.

Note: *Select the kit that you used for the earlier exercises as the BSP Source Template so that the starting BSP will work correctly on your kit. For example, if you used the CY8CKIT-062S2-43012, select that kit as the Source Template.*

☐   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]`.

Note: *If you are using the CY8CKPROTO-062-4343W, you will notice that the user LED is on pin P13[7] and there is no RGB LED. For this kit, simply move it to an unused pin such as P10[0]. No LED will blink with this change, but you can connect a discrete LED between P10[0] and VTARG if you want to see the LED blink. Make sure the anode is connected to VTARG and the cathode is connected to P10[0]. Alternately, you can use an oscilloscope to monitor P10[0].*

☐   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. The directory key_ch02_ex05_bsp contains custom BSPs for the CY8CKIT-062S2-43023 (known as MyBSP) and the CY8CPROTO-062-4343W (known as MyBSP_Proto). The directory key_ch02_ex05_application contains the application that uses the custom BSP. If you want to use the solution project, you will need to select 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.