Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

ModusToolbox™

User Guide

Document Number: 002-29893 Rev. *C

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
www.cypress.com

© Cypress Semiconductor Corporation, 2020. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC (“Cypress”). This document, including any software or firmware included or referenced in this document (“Software”), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress’s patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress does not assume any liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage (“Unintended Uses”). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, ModusToolbox, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.

Contents

1 Introduction

1.1 What is ModusToolbox?

The ModusToolbox software environment is a set of reference flows and products that provide an immersive development experience for creating converged MCU and wireless systems. Cypress provides a set of multi-platform development tools and a comprehensive suite of firmware libraries that enable you to design in the connectivity, processing, sensing, and security functionality you need.

The ModusToolbox experience is adaptable to the way you work, eschewing proprietary tools and custom build environments in favor of simplicity and openness. This means you choose your compiler, your IDE, your RTOS, and your ecosystem without compromising usability or access to our industry-leading CapSense®, Bluetooth Low Energy and Mesh, Wi-Fi, security and low-power features.

1.1.1 Reference Flows

Reference flows are Cypress documented, supported, and qualified methodologies to use ModusToolbox products. A flow is a recipe, defining how to create applications, add middleware, configure devices, build, program and debug.

1.1.1.1 Flows Covered in this Guide

This guide covers the PSoC 6 MCU, PSoC 6 with wireless connectivity, and WICED Bluetooth flows, based on the ModusToolbox installer tools and make build system.

1.1.1.2 Flows Not Covered in this Guide

Mbed OS and Amazon FreeRTOS flows are not covered in this guide; however, some of the tools in this guide are relevant to those flows. Refer to https://www.mbed.com/en/ and https://aws.amazon.com/freertos/, respectively, for more details about those flows.

1.1.2 Products

ModusToolbox products include tools and firmware that can be used individually, or as a group, to develop connected applications for Cypress devices. Cypress understands that you want to pick and choose the ModusToolbox products you use, merge them into your own flows, and develop applications in ways we cannot predict. That's why, unlike previous Cypress software offerings, ModusToolbox is not a monolithic, IDE-centric software tool. Each product is individually executable (for tools), buildable (for firmware), testable, portable, and deliverable. Products are distributed through multiple portals (for example mbed.com, github.com, and cypress.com) to enable you to work in your preferred environment.

1.2 High-Level What is Included

ModusToolbox software includes configuration tools, low-level drivers, middleware libraries, and operating system support, as well as other packages that enable you to create MCU and wireless applications. It also provides support for various industry-leading IDEs, including Visual Studio Code, Eclipse, IAR, and µVision. Unless specifically stated otherwise, ModusToolbox resources are compatible with Linux®, macOS®, and Windows®-hosted environments. Some parts of ModusToolbox are included with the installer, while others available online at the Cypress GitHub website.

See the ModusToolbox Software Overview chapter for more details about what is included as part of the ModusToolbox software.

1.2.1 ModusToolbox Installer

The ModusToolbox installer is available from the Cypress website:

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

It provides the basic software to get started, including:

- Compilers (GCC, ARM)
- Build System (make, Cygwin)
- Programming and Debug Tools (OpenOCD, PyOCD)
- Configurators and Tuners
- Eclipse IDE (optional) – This is a modified version of Eclipse. It includes a Cypress-specific set of plugins that provide convenience features for use with ModusToolbox tools.

1.2.2 Online Content

Cypress provides parts of ModusToolbox, such as libraries, drivers, and code examples, at the Cypress GitHub site:

https://github.com/cypresssemiconductorco

Typical resources available on GitHub include:

- BSPs (Board Support Packages)
- CSPs (Chip Support Packages) integrated into the BSP
- Libraries (e.g., RTOS, Drivers, Network Stacks, Graphics, etc.)
- Application Firmware (code examples)

1.3 About this Guide

This guide provides information and instructions for using the ModusToolbox software tools provided by the installer and the make build system. This document contains the following chapters:

- Chapter 2 provides tutorials for getting started using the ModusToolbox tools.
- Chapter 3 includes an overview of all the software considered a part of ModusToolbox.
- Chapter 4 describes the ModusToolbox build system.
- Chapter 5 covers different aspects of the ModusToolbox Board Support Packages (BSPs).
- Chapter 6 explains the ModusToolbox manifest files and how to use them with BSPs, libraries, and code examples.
- Chapter 7 provides instructions for using a ModusToolbox application with various integrated development environments (IDEs).

2 Getting Started

ModusToolbox software provides various graphical user interface (GUI) and command-line interface (CLI) tools to create and configure applications the way you want. You can use the included Eclipse-based IDE, which provides an integrated flow with all the ModusToolbox tools. Or, you can use other IDEs or no IDE at all, and you can switch between GUI and CLI tools in various ways to fit your design flow. Regardless of what tools you use, the basic flow for working with ModusToolbox applications includes these tasks:

- Install and Configure Software
- Get Help
- Create Applications
- Update BSPs and Libraries
- Configure Settings for Devices, Peripherals, and Libraries
- Write Application Code
- Build, Program, and Debug

This chapter helps you get started using various ModusToolbox tools. It covers these tasks, showing both the GUI and CLI options available.

2.1 Install and Configure Software

The ModusToolbox installer is located on the Cypress website:

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

The software can be installed on Windows, Linux, and macOS. Refer to the ModusToolbox Installation Guide for specific instructions.

2.1.1 GUI Set-up Instructions

In general, the IDE and other GUI-based tools included as part of the ModusToolbox installer work out of the box without any changes required. Simply launch the executable for the applicable GUI tool.

2.1.2 CLI Set-up Instructions

Before using the CLI tools, ensure that the environment is set up correctly.

- For **Windows**, the installer provides a command-line utility. Navigate to the *modus-shell* directory and run *Cygwin.bat*. It is located in the following directory:

  <install_path>/ModusToolbox/tools_2.1/modus-shell/

- For **macOS**, the installer will detect if you have the necessary tools. If not, it will prompt you to install them using the appropriate Apple system tools.
- For **Linux**, there is only a ZIP file, and you are expected to understand how to set up various tools for your chosen operating system.

To check your installation, open the appropriate command-line shell.

- Type `which make`. For most environments, it should return `/usr/bin/make`.
- Type `which git`. For most environments, it should return `/usr/bin/git`.

If these commands return the appropriate paths, then you can begin using the CLI. Otherwise, install and configure the GNU make and git packages as appropriate for your environment.

2.2 Get Help

In addition to this user guide, Cypress provides documentation for both GUI and CLI tools. GUI tool documentation is generally available from the tool's **Help** menu. CLI documentation is available using the tool's `-h` option.

2.2.1 GUI Documentation

2.2.1.1 Eclipse IDE

If you choose to use the integrated Eclipse IDE, see the *Eclipse IDE for ModusToolbox Quick Start Guide* for getting started information, and the *Eclipse IDE for ModusToolbox User Guide* for additional details.

2.2.1.2 Configurator and Tool Guides

Each GUI-based configurator and tool includes a user guide that describes different elements, as well as how to use it. See Installation Resources for descriptions of these tools and links to the documentation.

2.2.2 Command Line Documentation

2.2.2.1 make help

The ModusToolbox build system includes a `make help` target that provides help documentation. In order to use the help, you must first run the `make getlibs` command in an application directory (see make getlibs for details). From the appropriate shell in an application directory, type in the following to print the available make targets and variables to the console:

```
make help
```

To view verbose documentation for any of these targets or variables, specify them using the `CY_HELP` variable. For example:

```
make help CY_HELP=TOOLCHAIN
```

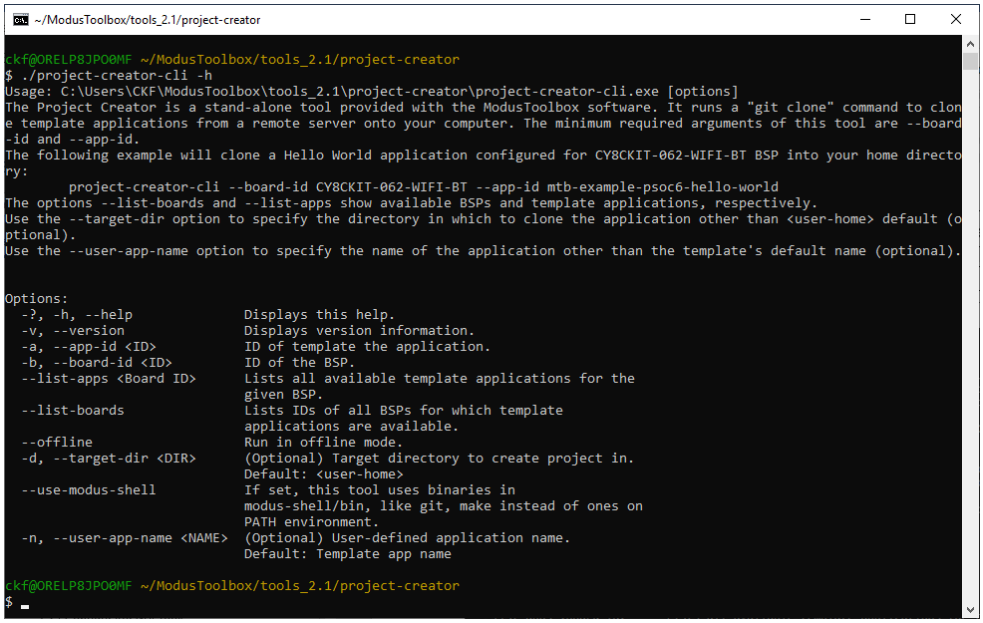
Note This help documentation is part of the base library, and it may also contain additional information specific to a BSP.

To see the various make targets and variables available, see the Available Make Targets and Available Make Variables sections in the ModusToolbox Build System chapter.

2.2.2.2 CLI Tools

Various CLI tools include a `-h` option that prints help information to the screen about that tool. For example, running this command prints output for the Project Creator CLI tool to the screen:

```
./project-creator-cli -h
```



2.3 Create Applications

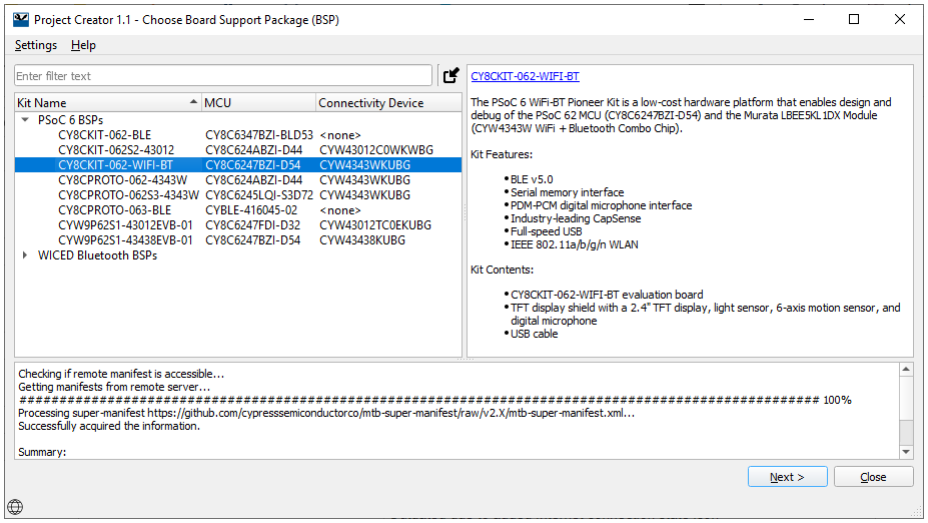
The ModusToolbox software provides the Project Creator as both a GUI tool and a command line tool to easily create a ModusToolbox application, based on available BSPs and code examples. The tool clones the selected code example for the specified BSP and puts everything required for the application into a single directory. You can then use those application files in your preferred IDE or from the command line.

If you prefer not to use the Project Creator tools, you can use the `git clone` command directly.

2.3.1 Project Creator GUI

Open the Project Creator GUI tool by launching the executable file installed in the following directory by default:

<install_path>/ModusToolbox/tools_2.1/project-creator/



Note You can also use the included Eclipse-based IDE tool to launch the Project Creator GUI tool. Using this method seamlessly exports the created application for use in the Eclipse IDE.

The Project Creator GUI tool provides a series of screens to select a BSP and code example, as well as to specify the application name and location. As part of creating the application, the tool adds the appropriate target to the makefile and adds the .lib file to match the selected BSP. It displays various messages during these operations. Refer to the Project Creator Guide for more details.

Note In some cases, the project may depend on one or more other projects. In those cases, the required projects are also created, if they don't already exist in the correct location.

2.3.2 project-creator-cli

Along with the Project Creator GUI tool, ModusToolbox software provides a project-creator-cli tool to create applications from a command-line prompt or from within batch files or shell scripts. The tool is located in the same directory as the GUI version (*<install_path>/ModusToolbox/tools_2.1/project-creator/*).

The following shows an example of running the tool with various options.

```
./project-creator-cli \
    --board-id CY8CKIT-062-WIFI-BT \
    --app-id mtb-example-psoc6-hello-world \
    --user-app-name MyLED \
    --target-dir "C:/cypress_projects"
```

To see all the options available, run the tool with the `-h` option.

In this example, the project-creator-cli tool runs the `git clone` command to clone the HelloWorld code example from the Cypress GitHub server (https://github.com/cypresssemiconductorco). It also updates the `TARGET` variable in the makefile to match the selected BSP `(--board-id)`, creates a *.lib* file for it, and runs the `make getlibs` command to obtain the necessary library files. This example also includes options to specify the name (`--user-app-name`) and location (`--target-dir`) where the application will be stored.

2.3.3 git clone

The Project Creator GUI and command line tools run the `git clone` command as part of the process of creating an application. You can run the `git clone` command directly from the command line. Open the appropriate shell and type in the following command (replace the <URL> with the appropriate URL of the repo you wish to clone):

```
git clone <URL>
```

The clone operation creates an application directory in your current location. Navigate to that directory (`cd <DIR>`), and find the application makefile. This is the top-level file that determines the application build flow. To see the various make targets and variables that you can edit in this file, refer to the Available Make Targets and Available Make Variables sections in the ModusToolbox Build System chapter.

2.4 Update BSPs and Libraries

As part of the application creation process, the Project Creator tools update the application with BSP and library information. If you use the `git clone` command, you may have to update BSP and library information as a separate process using the Library Manager tool or from the command line using the `make getlibs` command. You can also update the BSP and library information at any point in the development cycle using these tools.

2.4.1 Library Manager

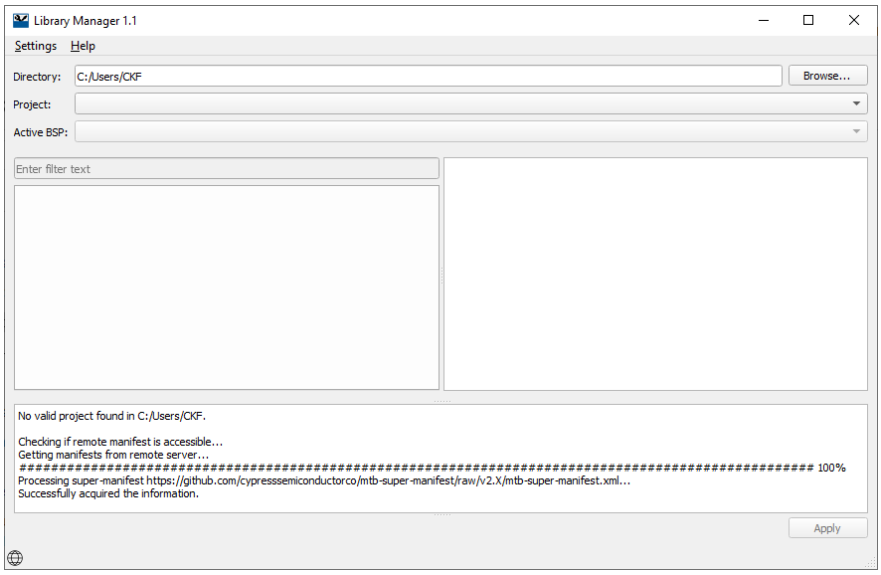
As needed, use the Library Manager tool to add or remove BSPs and libraries for your application, as well as change versions for BSPs and libraries. You can also change the active BSP. Refer to the Library Manager Guide for more details about using this tool.

Note For WICED Bluetooth projects, the Library Manager allows BSP version changes if you open the wiced_btsdk project. However, it does not support adding and removing shared dependencies at this time.

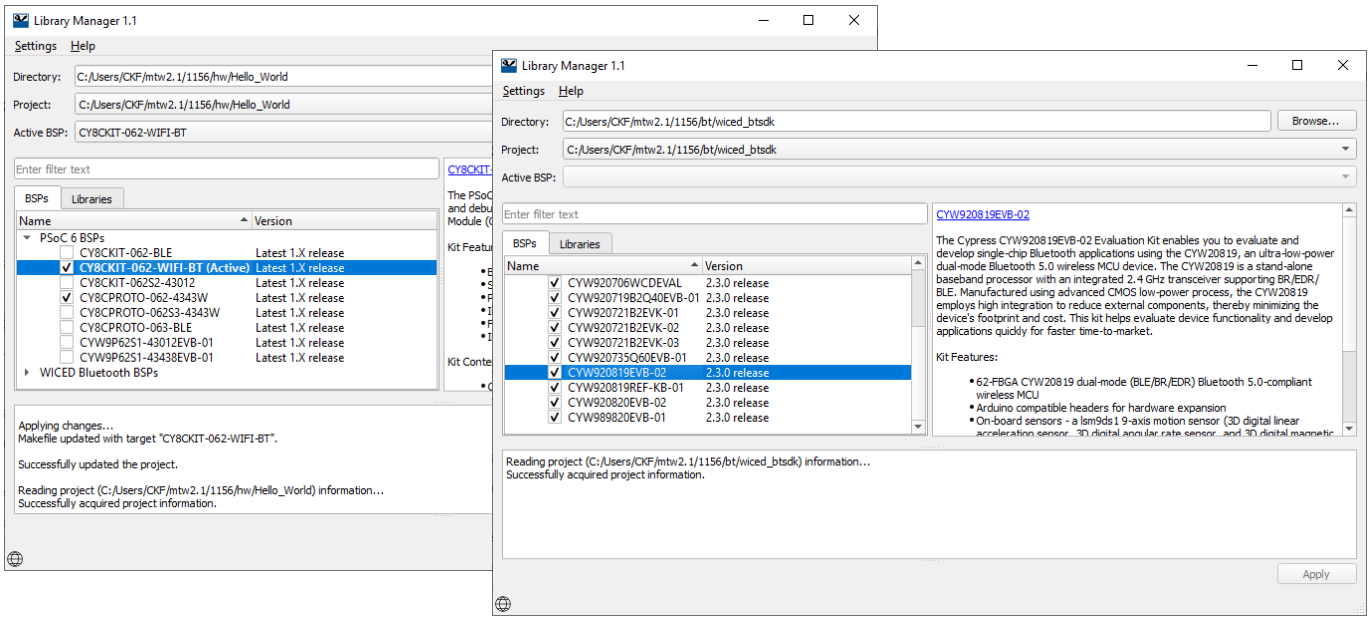
Open the Library Manager tool by launching the executable file installed in the following directory:

<install_path>/ModusToolbox/tools_2.1/library-manager/

If you haven't opened the Library Manager tool before, it opens at your home directory and looks for directories with a makefile. It also scans for available manifest files to acquire BSP/library information.

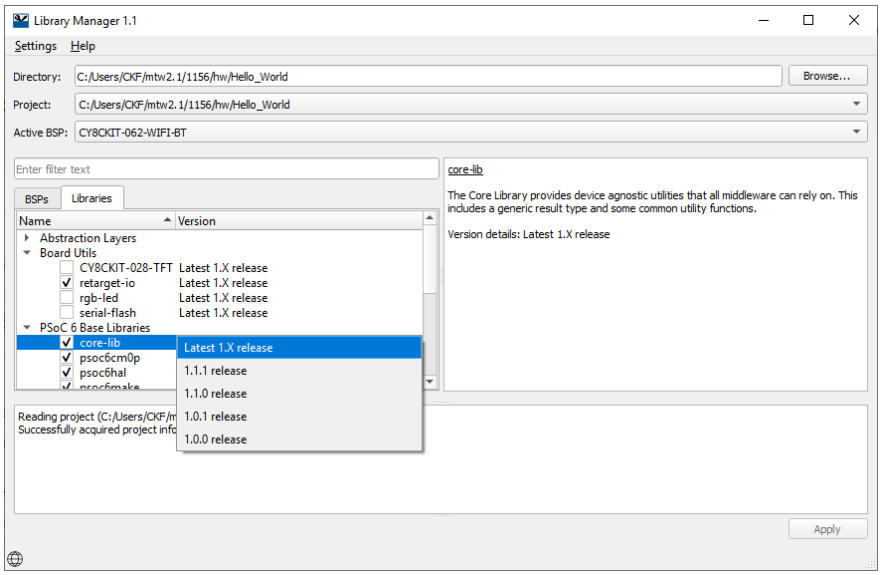


If the tool does not find your application, click the **Browse…** button next to the "Directory" field and navigate to where you created it. Select the directory and click **OK**. The Library Manager then updates to show the selected application and the available BSPs and libraries for it.

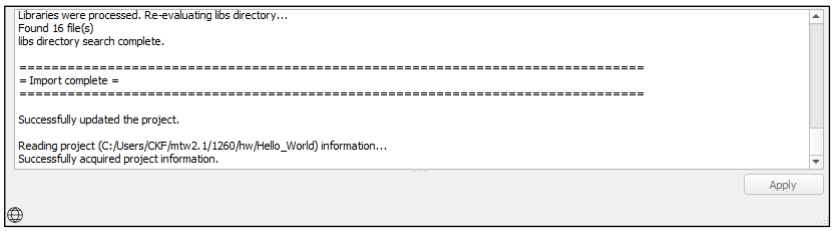


Note The next time you open the Library Manager, it will open with the most recently selected application.

The tool provides a field to select the **Active BSP**. It also includes two tabs to view and select **BSPs** and **Libraries** to add to (or remove from) your application. Select one or more check boxes for the items to add, and choose an appropriate **Version** of each item. Deselect check boxes for items to remove.



Click **Apply** to proceed with the changes. The status box displays various messages while applying changes, and then indicates if the application was updated or not.



2.4.2 make getlibs

The Project Creator tools and the Library Manager tool run the `make getlibs` command to search for all *.lib* files in the application directory. Each *.lib* file contains the library's git URL on which the application depends. These files are parsed, and the libraries are cloned into the application (by default in a directory named "libs").

If you ran the `git clone` command manually and did not use the Library Manager, then your application will only contain default *.lib* files. You will need to run the `make getlibs` command to parse those files and clone the libraries. However, if you want to use to a different BSP than the default provided by the code example, you must first edit the makefile to update the `TARGET` variable to match the desired BSP. Then, you must add a *.lib* file in the */deps* directory that includes a URL to the desired BSP location.

Note Older ModusToolbox version 2.0 applications contain a */libs* directory for *.lib* files, while newer version 2.1 applications have a */deps* directory. The build system will find *.lib* files in either directory, and it will import the necessary library files into the */libs* directory for all applications.

When you are ready to update your application, open the appropriate shell and run the following command in the application directory:

```
make getlibs
```

Note Any *.lib* file that is not a text file (for instance Windows *.lib* archive files) is ignored for this process.

Note The `make getlibs` operation may take a long time to execute as it depends on your internet speed and the size of the libraries that it is cloning. To improve subsequent library cloning operations, a cache directory named ".modustoolbox/cache" exists in the $HOME (Linux, macOS) and $USERPROFILE (Windows) directories.

2.5 Configure Settings for Devices, Peripherals, and Libraries

Depending on your application, you may want to update and generate some of the configuration code. While it is possible to write configuration code from scratch, the effort to do so is considerable. ModusToolbox software provides applications called configurators that make it easier to configure a hardware block or a middleware library.

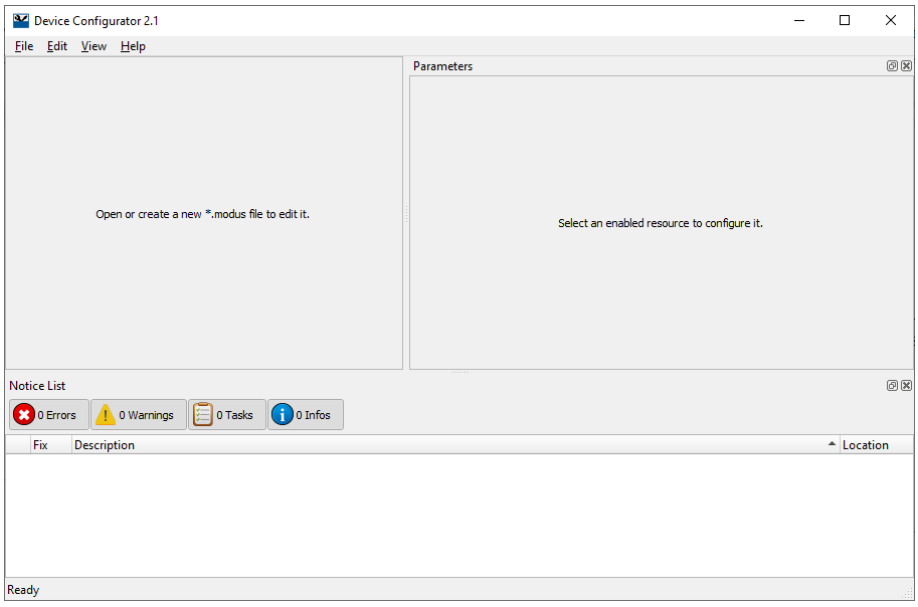
The configurators can be run as GUIs to easily update various parameters and settings. Most can also be run as command line tools to regenerate code as part of a script. For more information about configurators, see the Configurators section in the ModusToolbox Software Overview chapter. Also, each configurator provides a separate document, available from the configurator's **Help** menu, that provides information about how to use the specific configurator.

2.5.1 Configurator GUI Tools

By default, configurators open as GUIs when you launch their executable files, which are located in separate subdirectories the "tools" directory:

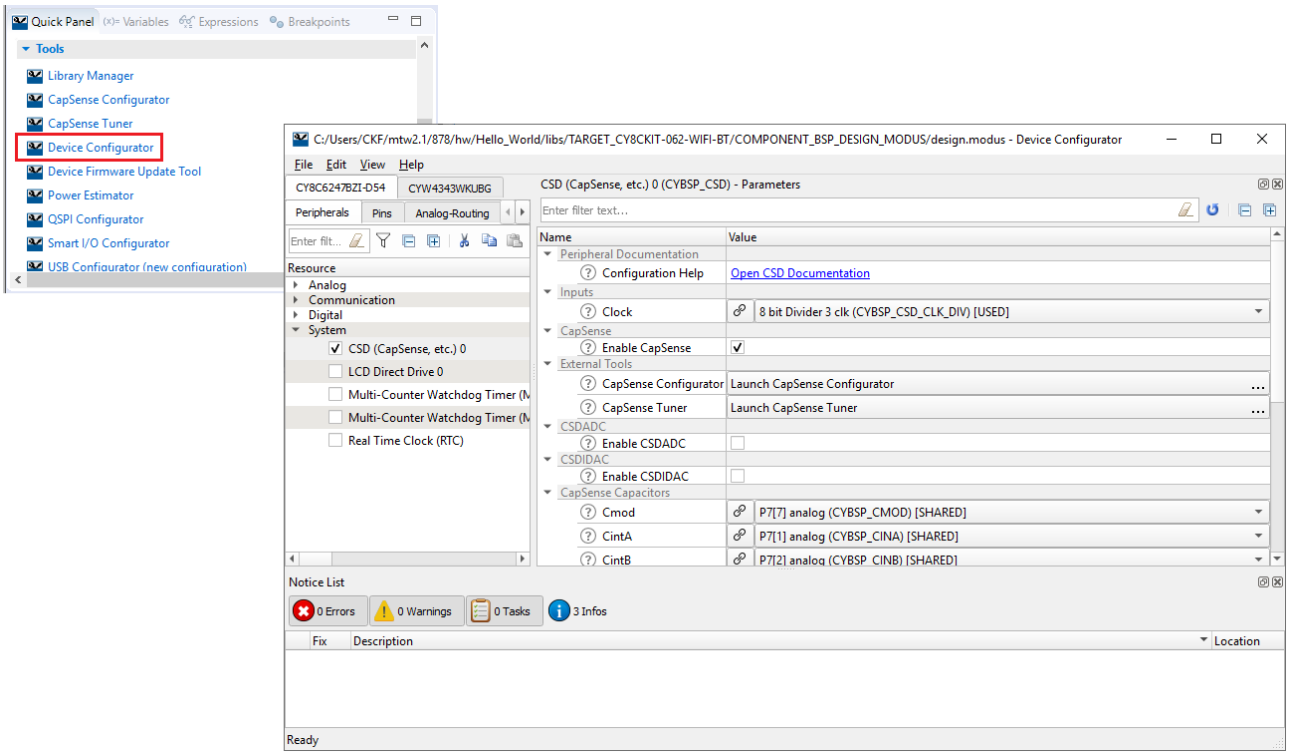
<install_path>/ModusToolbox/tools_2.1/

In the Windows operating system for example, if you double-click the *device-configurator.exe* file, the Device Configurator opens as follows:



In this case, the Device Configurator opens without any application information. So, you'll have to open the application's device configuration (*design.modus*) file, which is located in the application's BSP.

If you use the Eclipse IDE provided with ModusToolbox, configurators are accessible from the IDE, and they will open the appropriate application information. For example, if you select the “Device Configurator” link in the IDE Quick Panel, the tool opens with the application’s *design.modus* file.



From the Device Configurator GUI, you can launch BSP configurators such as CapSense and SegLCD. You can also run the GUIs by launching the executable from the configurator’s install directory.

You can run a few configurators from the command line by using the appropriate make target in the application’s directory. This method also opens the configurator with the appropriate configuration file from the application.

To launch the Device Configurator, run:

```
make config
```

To launch the Bluetooth Configurator, run:

```
make config_bt
```

To launch the USB Configurator, run:

```
make config_usbdev
```

2.5.2 Configurator CLI Tools

Most of the configurator GUIs can also be run from the command line. The primary use case is to re-generate source code based on the latest configuration settings. This would often be part of an overall build script for the entire application. The command-line configurator cannot change configuration settings. For information about command line options, run the configurator using the `-h` option.

2.6 Write Application Code

As in any embedded development application using any set of tools, you are responsible for the design and implementation of the firmware. This includes not just low-level configuration and power mode transitions, but all the unique functionality of your product. As noted, you can use the ModusToolbox Eclipse IDE, your preferred IDE, or a text editor and command line tools. ModusToolbox software is designed to enable your workflow.

Taken together, the multiple resources available to you in ModusToolbox software: BSPs, configurators, driver libraries and middleware, help you focus on your specific application.

2.7 Build, Program, and Debug

After the application has been created, you can export it to an IDE of your choice for building, programming, and debugging. You can also use command line tools. The ModusToolbox build system infrastructure provides several make variables to control the build. So, whether you are using an IDE or command line tools, you edit the makefile variables as appropriate. See the ModusToolbox Build System chapter for detailed documentation on the build system infrastructure.

Variable	Description
`TARGET`	Specifies the target board/kit. For example, CY8CPROTO-062-4343W
`APPNAME`	Specifies the name of the application
`TOOLCHAIN`	Specifies the build tools used to build the application
`CONFIG`	Specifies the configuration option for the build [Debug Release]
`VERBOSE`	Specifies whether the build is silent or verbose [true false]

ModusToolbox software is tested on these tool chains:

Variable value	Tools	Host OS
`GCC_ARM`	GNU Arm Embedded Compiler v7.2.1	Mac OS, Windows, Linux
`ARM`	Arm compiler v6.11	Windows, Linux
`IAR`	Embedded Workbench v8.32	Windows

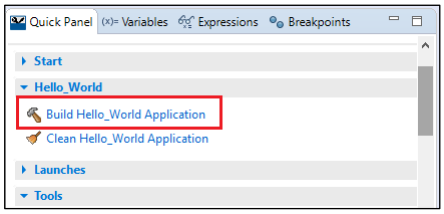
In the makefile, set the `TOOLCHAIN` variable to the build tools of your choice. For example: `TOOLCHAIN=GCC_ARM`.

There are also variables you can use to pass compiler and linker flags to the tool chain.

ModusToolbox software installs the GNU Arm tool chain and uses it by default. If you wish to use another tool chain, you must provide it and specify the path to the tools. For example, `CY_COMPILER_PATH=<yourpath>`. If this path is blank, the build infrastructure looks in the ModusToolbox install directory.

2.7.1 Use Eclipse IDE

When using the provided Eclipse IDE, click the **Build Application** link in the Quick Panel for the selected application.



Because the IDE relies on the build infrastructure, it does not use the standard Eclipse GUI to modify build settings. It uses the build options specified in the makefile. This design ensures that the behavior of the application, its options, and the make process are consistent regardless of the development environment and workflow.

2.7.2 Export to another IDE

If you prefer to use another IDE, see the Exporting to IDEs chapter for more details. When working with a different IDE, you must manage the build using the features and capabilities of that IDE.

2.7.3 Use Command Line

2.7.3.1 make build

When all the libraries are available (after executing `make getlibs`), the application is ready to build. From the appropriate shell, type the following:

```
make build
```

This instructs the build system to find and gather the source files in the application and initiate the build process. In order to improve the build speed, you may parallelize it by giving it a `-j` flag (optionally specifying the number of processes to run). For example:

```
make build -j16
```

2.7.3.2 make program

Connect the target board to the machine and type the following in the shell:

```
make program
```

This performs an application build and then programs the application artifact (usually an *.elf* or *.hex* file) to the board using the recipe-specific programming routine (usually OpenOCD). You may also skip the build step by using `qprogram` instead of `program`. This will program the existing build artifact.

2.7.3.3 make debug/qdebug/attach

The following commands can be used to debug the application, as follows:

- `make debug` – Build and program the board. Then launch the GDB server.
- `make qdebug` – Skip the build and program steps. Just launch the GDB server.
- `make attach` – Starts a GDB client and attaches the debugger to the running target.

3 ModusToolbox Software Overview

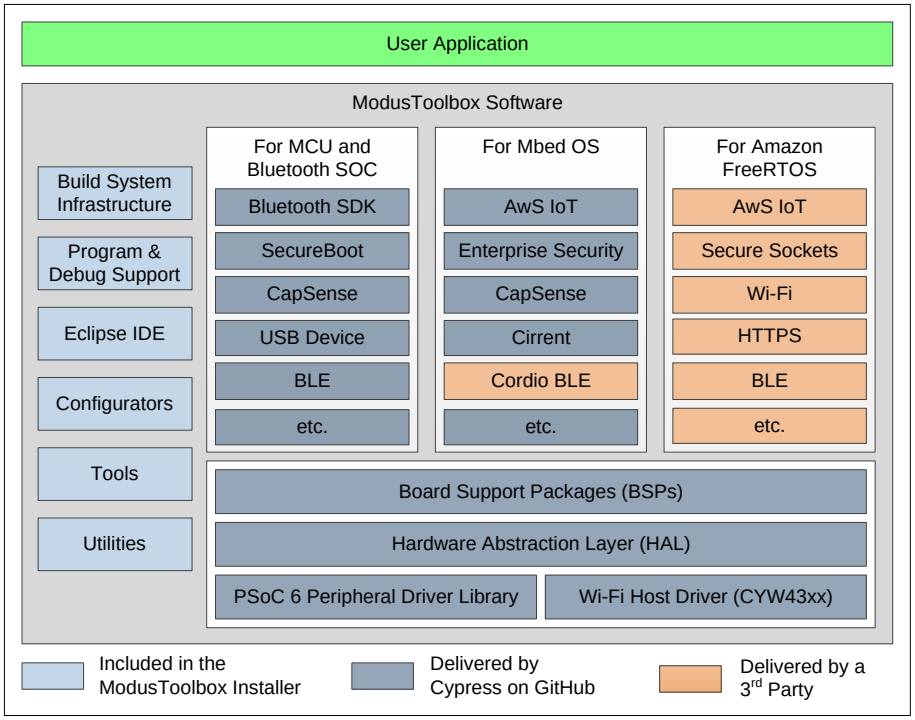
ModusToolbox software includes configuration tools, low-level drivers, middleware libraries, and operating system support, as well as other packages that enable you to create MCU and wireless applications. It also includes an optional Eclipse IDE. Unless specifically stated otherwise, ModusToolbox resources are compatible with Linux®, macOS®, and Windows®-hosted environments.

The ModusToolbox installer provides resources you need to get started, such as configurators that generate code based on your design. In addition, Cypress provides libraries and enablement software at the Cypress Semiconductor GitHub site. Some resources will be used by all developers. Others will be used by developers in particular ecosystems.

Cypress software resources available at GitHub support one or more of the target ecosystems:

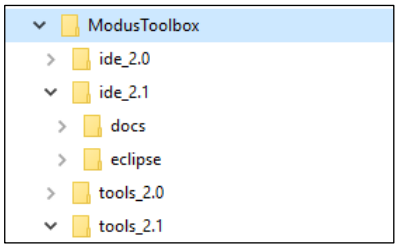
- MCU and Bluetooth SOC ecosystem – a full-featured platform for PSoC 6, Wi-Fi, Bluetooth, and Bluetooth Low Energy application development
- Mbed OS ecosystem – provides an embedded operating system, transport security and cloud services to create connected embedded solutions
- Amazon FreeRTOS ecosystem – extends the FreeRTOS kernel with software libraries that make it easy to securely connect small, low-power devices to AWS cloud services

Some resources support all ecosystems. Others are specific to a particular ecosystem. The following block diagram is not a comprehensive list. However, it conveys the idea that, depending upon your programming domain, multiple resources are available to you. See Installation Resources for available tools and Enablement Software for available libraries and board support packages.



3.1 Directory Structure

Refer to the *ModusToolbox Installation Guide* for information about installing ModusToolbox. Once it is installed, the various ModusToolbox top-level directories are organized as follows:



Note This image shows ModusToolbox versions 2.0 and 2.1 installed. Your installation may only include ModusToolbox version 2.1. Refer to the Product Versioning section for more details.

These directories contain the following files and folders:

- **ide_2.1**
  - **docs –** This is the top-level documentation directory. It contains various top-level documents and an html file with links to documents provided as part of ModusToolbox. See Documentation for more information.
  - **eclipse (or ModusToolbox.app on macOS) –** This contains the IDE. See Eclipse IDE.
- **tools_2.1 –** This contains all the various tools and scripts provided as part of ModusToolbox. See Tools for more information.

3.1.1 Documentation

The */ide_2.1/docs* directory contains top-level documents and an HTML document with links to all the documents included in the installation and on the web.

3.1.1.1 Release Notes

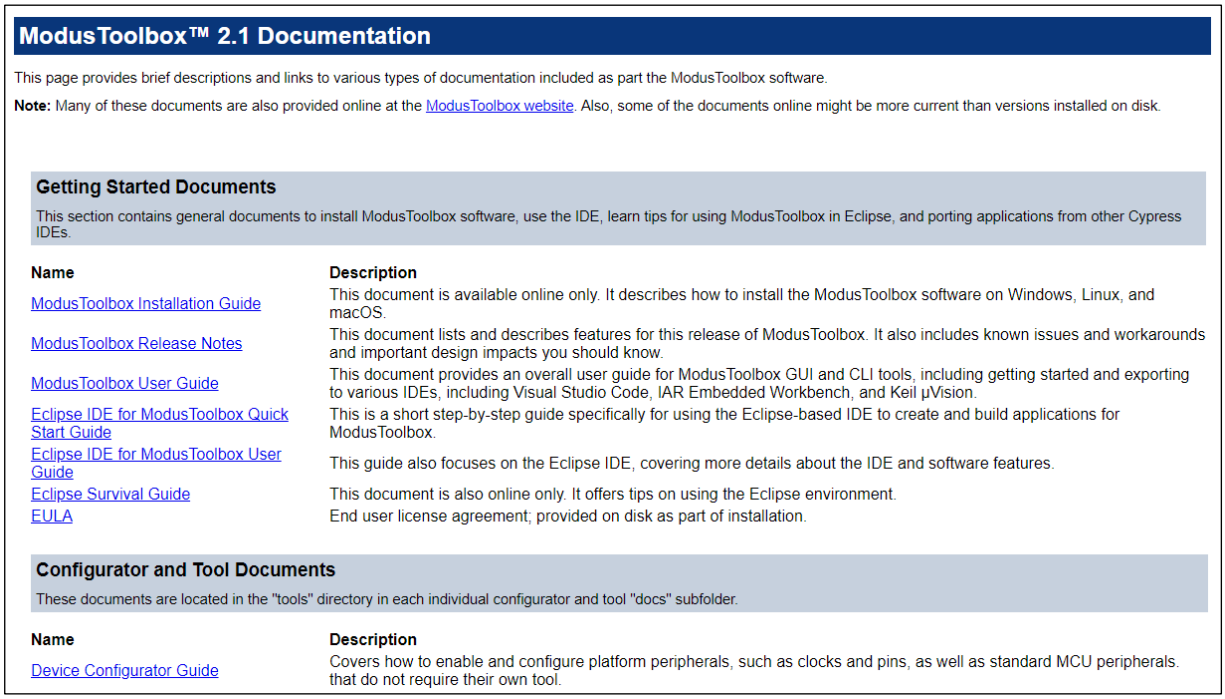
For the 2.1 release, the release notes document is for all of the ModusToolbox software included in the installation.

3.1.1.2 Top-Level Documents

This folder also contains the Eclipse IDE documentation and this user guide. These guides cover different aspects of using the IDE and various ModusToolbox tools.

3.1.1.3 Document Index Page

The *doc_landing.html* file provides links to all the documents included in the installation and on the web. This file is also available from the IDE **Help** menu.



ModusToolbox™ 2.1 Documentation

This page provides brief descriptions and links to various types of documentation included as part the ModusToolbox software.

Note: Many of these documents are also provided online at the ModusToolbox website. Also, some of the documents online might be more current than versions installed on disk.

Getting Started Documents

This section contains general documents to install ModusToolbox software, use the IDE, learn tips for using ModusToolbox in Eclipse, and porting applications from other Cypress IDEs.

Name	Description
ModusToolbox Installation Guide	This document is available online only. It describes how to install the ModusToolbox software on Windows, Linux, and macOS.
ModusToolbox Release Notes	This document lists and describes features for this release of ModusToolbox. It also includes known issues and workarounds and important design impacts you should know.
ModusToolbox User Guide	This document provides an overall user guide for ModusToolbox GUI and CLI tools, including getting started and exporting to various IDEs, including Visual Studio Code, IAR Embedded Workbench, and Keil µVision.
Eclipse IDE for ModusToolbox Quick Start Guide	This is a short step-by-step guide specifically for using the Eclipse-based IDE to create and build applications for ModusToolbox.
Eclipse IDE for ModusToolbox User Guide	This guide also focuses on the Eclipse IDE, covering more details about the IDE and software features.
Eclipse Survival Guide	This document is also online only. It offers tips on using the Eclipse environment.
EULA	End user license agreement; provided on disk as part of installation.

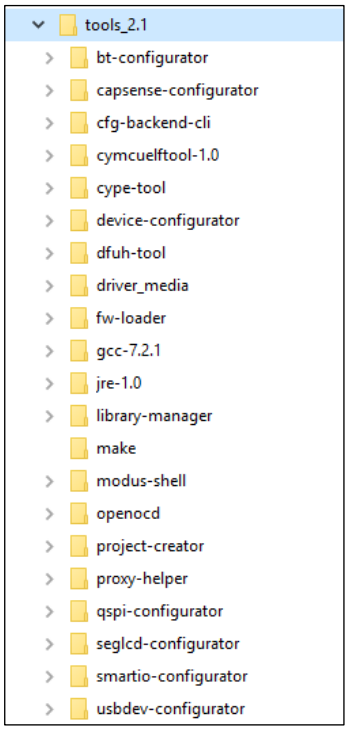
Configurator and Tool Documents

These documents are located in the "tools" directory in each individual configurator and tool "docs" subfolder.

Name	Description
Device Configurator Guide	Covers how to enable and configure platform peripherals, such as clocks and pins, as well as standard MCU peripherals. that do not require their own tool.

3.1.2 Tools

The */tools_2.1* folder contains the following tools:



- Configurators – There are several configurators used to update various settings for different peripherals. See Configurators.
- cfg-backend-cli – This contains backend support files used by the system. You do **not** need to interact with this folder.
- cymcuelftool-1.0 – This tool is used to manipulate Elf files. Refer to the *CyMCUElfTool User Guide* located in the tool's doc folder.
- cype-tool – This is the Low Power Estimator tool.
- dfuh-tool – This tool is used to communicate with and update firmware on a PSoC 6 MCU that has already been programmed with an application that includes device firmware update capability.
- driver_media – This folder contains WICED board drivers.
- fw-loader – This is the Firmware Loader tool used to update firmware on the programmer/debugger device on PSoC 6 MCU kits.
- gcc-7.2.1 – ModusToolbox software includes GCC version 7.2.1 as the preferred toolchain. See https://www.gnu.org/software/gcc/ for information.
- jre-1.0 – This folder contains the Java Runtime Environment version provided as part of the tool. This is used by the IDE and the backend. See https://www.java.com for more information.
- library-manager – This is the Library Manager tool.
- make – This folder contains scripts for the build system.
- modus-shell – This folder contains various helper utilities used by the system. On Windows, this contains a version of Cygwin designed to work with ModusToolbox.
- openocd – This contains the version of the Open On-Chip Debugger used by ModusToolbox to program various boards. For more information, refer to the *Cypress Programmer User Guide*.
- project-creator – Tool used to create projects. You invoke this tool when creating projects in the IDE. You can also run this as a stand-alone tool.
- proxy-helper – This is a command-line tool for managing ModusToolbox proxy server settings.

3.2 Product Versioning

As stated previously, ModusToolbox products include tools and firmware that can be used individually, or as a group, to develop connected applications for Cypress devices. Cypress understands that you want to pick and choose the ModusToolbox products you use, merge them into your own flows, and develop applications in ways we cannot predict. However, it is important to understand that each product may have more than one version. This section describes how ModusToolbox products are versioned.

3.2.1 General Philosophy

ModusToolbox is not a single monolithic entity that is tested and distributed all together. Libraries and tools in the context of ModusToolbox are effectively "mini-products" with their own release schedules, upstream dependencies, and downstream dependent assets and applications.

The delivery method chosen for the libraries is GitHub. The delivery method for tools is within the ModusToolbox installation package.

All ModusToolbox products developed by Cypress follow the standard versioning scheme:

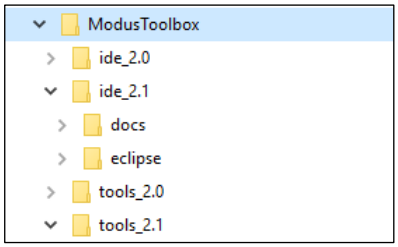
- If there are known backward compatibility breaks, the major version is incremented.
- Minor version changes may introduce new features and functionality, but are "drop-in" compatible.
- Patch version changes address defects. They are very low-risk (fix the essential defect without unnecessary cruft or complexity).

Cypress recommends you lock down the tools versions used in a project makefile, along with switching to "release-X.Y.Z" versions of the libraries as soon as you are done with initial development for your project. You may update later.

3.2.2 Install Package Versioning

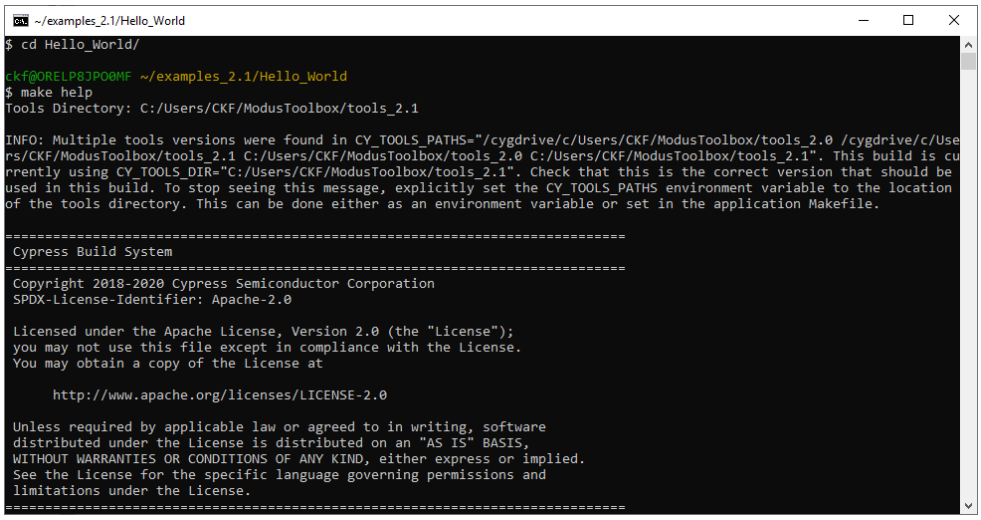
The ModusToolbox installation package is versioned as MAJOR.MINOR.PATCH. The file located at *<install_path>/ModusToolbox/tools_2.1/version-2.1.0.xml* also indicates the build number.

Every MAJOR.MINOR version of ModusToolbox products are installed by default into *<install_path>/ModusToolbox*. So, if you have multiple versions of ModusToolbox installed, they are all installed in parallel in the same *ModusToolbox* directory, as follows:



3.2.3 Multiple Tools Versions Installed

When you run make commands from the command line, a message displays if you have multiple versions of the “tools” directory installed and if you have not specified a version to use.

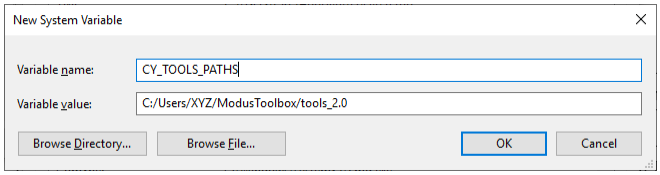


3.2.4 Specifying Alternate Tools Version

By default, the ModusToolbox software uses the most current version of the tools directory installed. That is, if you have ModusToolbox version 2.1 and 2.0 installed, and if you launch the Eclipse IDE from the ModusToolbox 2.0 installation, the IDE will use the tools from the “tools_2.1” directory to launch configurators and build an application. This section describes how to specify the path to the desired version.

3.2.4.1 System Variable

The overall way to specify a path other than the default “tools” directory, is to use a system variable named `CY_TOOLS_PATHS`. On Windows, open the Environment Variables dialog, and create a new System Variable:



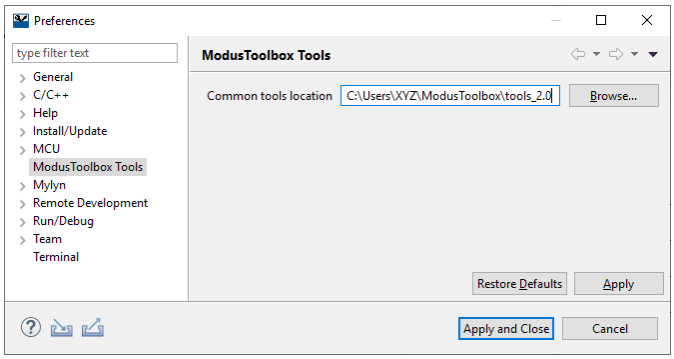
Note: Use a Windows style path, (that is, not like /cygdrive/c/). Also, use forward slashes. For example:

C:/Users/XYZ/ModusToolbox/tools_2.0

Use the appropriate method for setting variables in macOS and Linux for your system.

3.2.4.2 Eclipse IDE Workspace Setting

The Eclipse IDE provided with ModusToolbox includes a setting to specify the tools path that applies only to a specific workspace. Select **Windows > Preferences > ModusToolbox Tools**.



Then, in the Common tools location field, click the **Browse…** button and navigate to the appropriate “tools” directory to use.

3.2.4.3 Specific Project Makefile

To preserve a specific “tools” path for the specific project, edit that project’s makefile, as follows:

```
# If you install the IDE in a custom location, add the path to its
# "tools_X.Y" folder (where X and Y are the version number of the tools
# folder).
CY_TOOLS_PATHS+=C:/Users/XYZ/ModusToolbox/tools_2.0
```

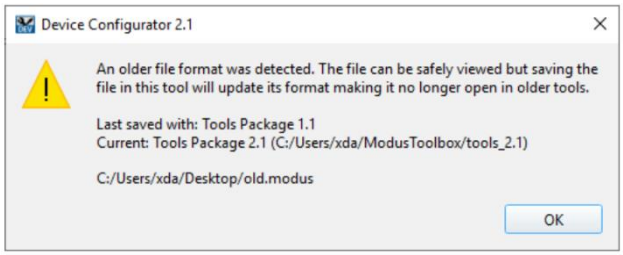
Note: If you are using the Eclipse IDE, you must still specify the Eclipse IDE Workspace Setting.

3.2.5 Tools and Configurators Versioning

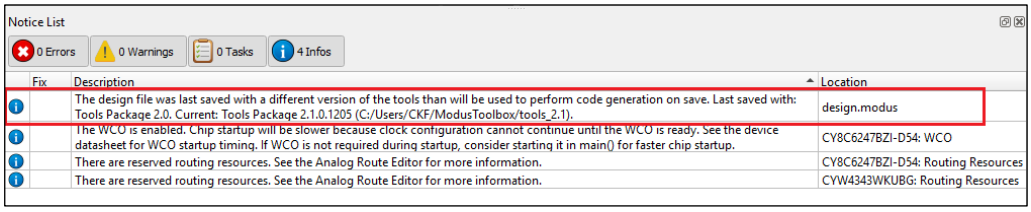
Every tool and configurator follow the standard versioning scheme and include a *version.xml* file that also contains a build number.

3.2.5.1 Configurator Messages

Configurators indicate if you are about to modify the configuration file (for example, *design.modus*) with a newer version of the configurator, as well as if there is a risk that you will no longer be able to open it with the previous version of the configurator:



Configurators will also indicate if you are trying to open the existing configuration with a different, backward and forward compatible version of the Configurator.



Note: If using the command line, the build system will notify you with the same message.

3.2.6 GitHub Libraries Versioning

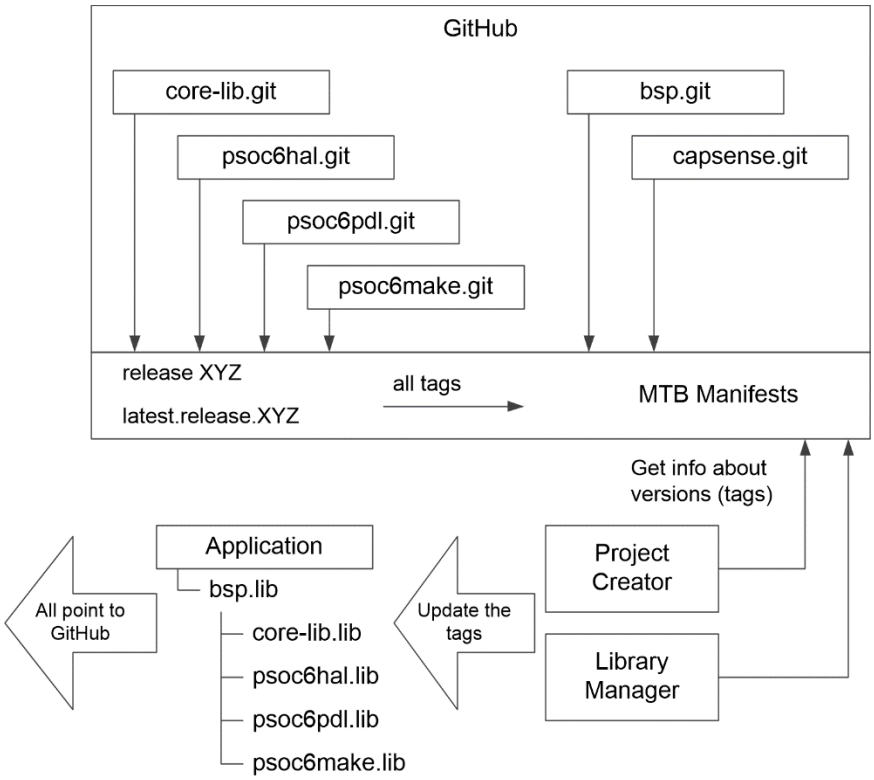
GitHub Libraries follow the same versioning scheme: MAJOR.MINOR.PATCH. The GiHub libraries, besides the code itself, also provide two files in MD format: README and RELEASE. The latter includes the version and the change history.

The versioning for GitHub libraries is implemented using GitHub Tags. These tags are captured in the manifest files (see the Manifest Files chapter for more details). The Project Creator and Library Manager tools parse these manifests and allow you to see and select between various tags of these libraries. When selecting a particular library of a particular version, the *.lib* file gets created in your project. These *.lib* files are a link to the specific tag. Refer to the Library Manager User Guide for more details about tags.

Once complete with initial development for your project, Cypress recommends you switch to specific “release” tags. Otherwise, running the `make getlibs` command will update the libraries referenced by the *.lib* files, and will deliver the latest code changes for the major version.

3.2.7 Dependencies Between Libraries

The following diagram shows the dependencies between libraries.



There are dependencies between the libraries. There are two types of dependencies:

3.2.7.1 Dependencies via.lib files

One library includes the other library – implemented with the .lib files included in the parent library. This way the BSP references the core and make libraries for example. See the Library manager guide to understand more about the direct and indirect libraries and how they are stored based upon the CY_GETLIBS_DEPS_PATH project makefile variable.

3.2.7.2 Regular C Dependencies via #include

Cypress Libraries only call the documented public interface of other Libraries. Every library declares its version in the header. The consumer of the library including the header checks if the version is supported, and will notify via #pragma if the newer version is required. Examples of the dependencies:

- The Device Support library (PDL) driver is used by the Middleware.
- The configuration generated by the Configurator depends on the versions of the device support library (PDL) or on the Middleware headers.

Similarly, if the configuration generated by the Configurator of the newer version than you have installed, the notification via the build system will trigger asking you to install the newer version of the ModusToolbox. ModusToolbox has a fragmented distribution model. Users are allowed and empowered to update Libraries individually.

3.3 Installation Resources

The ModusToolbox installer provides required and optional core resources for any application. This section provides an overview of the available resources:

- Build System Infrastructure
- Program and Debug Support
- Eclipse IDE
- Configurators
- Tools
- Utilities

The installer does not include enablement software such as driver libraries or middleware.

3.3.1 Build System Infrastructure

The build system infrastructure is the fundamental resource in ModusToolbox software. It serves three primary purposes:

- create an application
- create an executable
- provide debug capabilities

A makefile defines everything required for your application, including:

- the target hardware (board/board support package to use)
- the source code and libraries to use for the application
- the build tools to use
- compiler/assembler/linker flags to control the build

The build system automatically discovers all .c, .h, .cpp, .s, .a, .o files in the application directory and subdirectories, and uses them in the application. The makefile can also discover files outside the application directory. You can add another directory using the `CY_SHAREDLIB_PATH` variable. You can also explicitly list files in the `SOURCES` and `INCLUDES` make variables.

Each library used in the application is identified by a *.lib* file. This file contains the URL to a git repository, and a commit tag. Cypress git repositories are on GitHub. For example, a capsense.lib file might contain the following line:

```
https://github.com/cypresssemiconductorco/capsense/#release-v2.0.0
```

The build system implements the `make getlibs` command. This command finds each *.lib* file, clones the specified repository, checks out the specified commit, and collects all the files in a single *libs* directory in the application directory. Typically the `make getlibs` command is invoked transparently when you create an application, although you can invoke the command directly from a command line interface. See ModusToolbox Build System for detailed documentation on the build system infrastructure.

3.3.2 Program and Debug Support

ModusToolbox software supports the Open On-Chip Debugger (OpenOCD) using a GDB server, and supports the J-Link debug probe. For the Mbed OS ecosystem, ModusToolbox supports Arm Mbed DAPLink.

The Eclipse IDE can program devices and establish a debug session. For programming, Cypress Programmer is available separately. It is a cross-platform application for programming Cypress PSoC 6 devices. It can program, erase, verify, and read the flash of the target device.

Cypress Programmer and the Eclipse IDE use KitProg3 low-level communication firmware. The firmware loader (fw-loader) is a software tool you can use to easily switch back and forth between KitProg2 and KitProg3, if you need to do so. The fw-loader tool is installed with the ModusToolbox software. It is also available separately in a GitHub repository.

Tool	Description	Documentation
Cypress Programmer	Cypress Programmer functionality is built into ModusToolbox Software. Cypress Programmer is also available as a stand-alone tool.	Programming Tools page, go to the documentation tab
fw-loader	A simple command line tool to identify which version of KitProg is on a Cypress kit, and easily switch back and forth between legacy KitProg2 and current KitProg3.	*readme.txt* file in the tool directory
KitProg3	This tool is managed by fw-loader, it is not available separately. KitProg3 is Cypress' low-level communication/debug firmware that supports CMSIS-DAP and DAPLink (for Mbed OS). Use fw-loader to upgrade your kit to KitProg3, if it has KitProg2 installed.	User Guide
OpenOCD	A Cypress-specific implementation of OpenOCD is installed with ModusToolbox software.	Developer's Guide
DAPLink	Support is implemented through KitProg3	DAPLink Handbook

3.3.3 Eclipse IDE

The Eclipse IDE included with ModusToolbox is a full-featured, cross-platform IDE. It includes application management, code authoring and editing, build tools, and debug capabilities. The IDE supports the C and C++ programming languages. It includes the GCC Arm build tools. It supports debugging via OpenOCD or J-Link. You can use the IDE to develop applications using ModusToolbox software. The IDE is optional. See Eclipse IDE for ModusToolbox User Guide for more details.

3.3.4 Configurators

Depending on your application, you may want to update and generate some of the configuration code. While it is possible to write configuration code from scratch, the effort to do so is considerable. ModusToolbox software provides graphical applications called configurators that make it easier to configure a hardware block or a middleware library. 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.

Each configurator is a cross-platform tool that allows you to set configuration options for the corresponding hardware peripheral or library. When you save a configuration, the tool generates the C code or configuration file used to initialize the hardware or library with the desired configuration.

Configurators are independent of each other, but they can be used together to provide flexible configuration options. They can be used stand alone, in conjunction with other configurators, or as part of a complete application. All of them are installed during the ModusToolbox installation. Each configurator provides a separate guide, available from the configurator's **Help** menu.

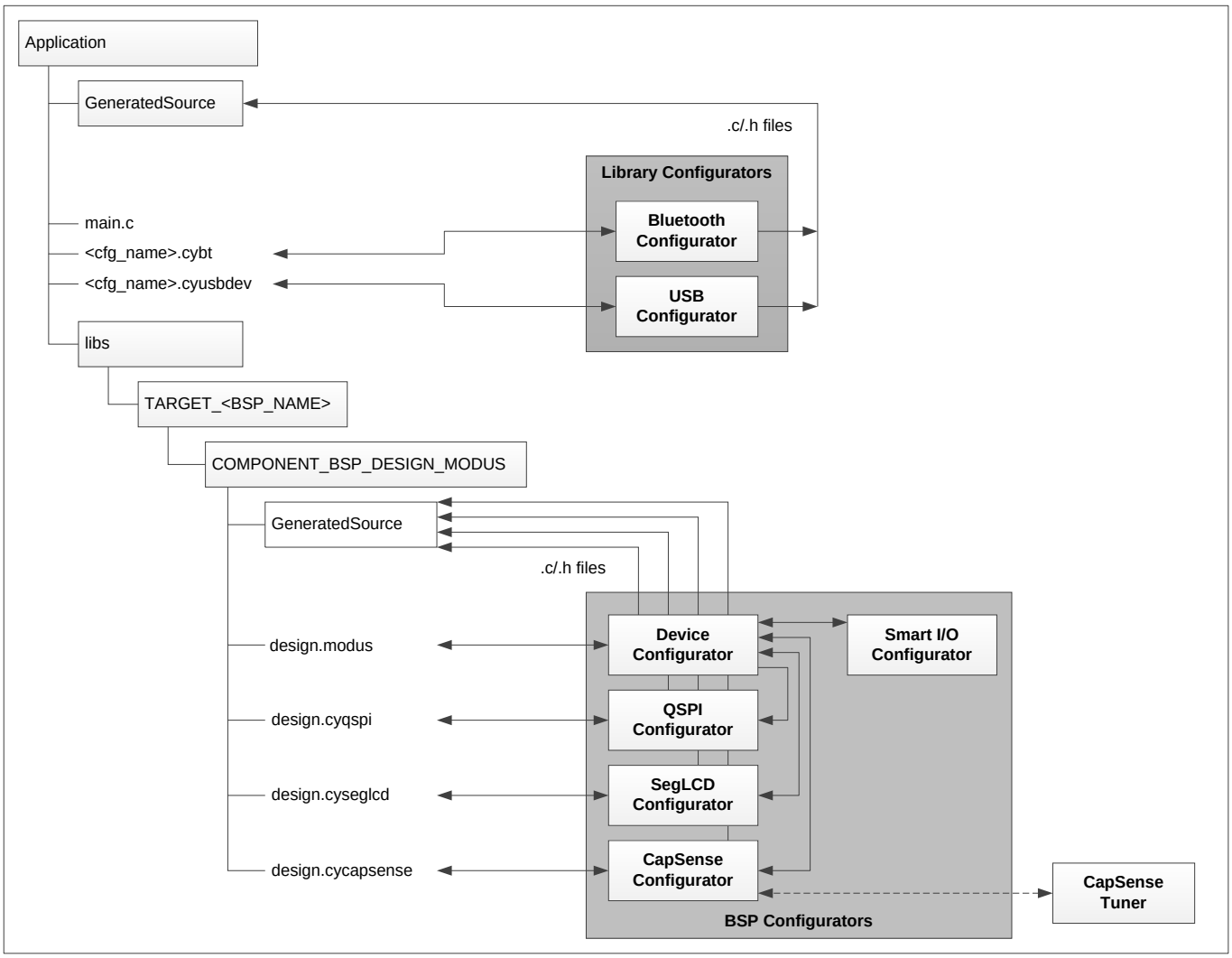
Configurators perform tasks such as:

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

Note Some configurators may not be useful for your application.

Configurators store configuration data in an XML data file that provides the desired configuration. Each configurator has a "command line" mode that can regenerate source based on the XML data file. Configurators are divided into two types: BSP Configurators and Library Configurators.

The following diagram shows a high-level view of the configurators in a typical application.



BSP configurators configure the hardware on a specific device. This can be a board provided by Cypress, a Cypress partner, or a board that you create that is specific to your application. Some of these configurators interact with the *design.modus* file to store and communicate configuration settings between different configurators. Code generated by a BSP Configurator is stored in a directory named *GeneratedSource*, which is in the same directory as the *design.modus* file. This is generally located in the BSP for a given target board. BSP configurators include:

- Device Configurator: Set up the system (platform) functions such as pins, interrupts, clocks, and DMA, as well as the basic peripherals, including UART, Timer, etc. See Device Configurator Guide for more details.
- CapSense Configurator: Configure CapSense hardware, and generate the required firmware. This includes tasks such as mapping pins to sensors and how the sensors are scanned. See CapSense Configurator Guide for more details.
- There is also a CapSense Tuner to adjust performance and sensitivity of CapSense widgets. See CapSense Tuner Guide for more details.
- QSPI Configurator: Configure external memory and generate the required firmware. This includes defining and configuring what external memories are being communicated with. See QSPI Configurator Guide for more details.
- Smart I/O™ Configurator: Configure the Smart I/O. This includes defining and configuring what external memories are being communicated with. See Smart I/O Configurator Guide for more details.
- 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. See SegLCD Configurator Guide for more details.

Library configurators support configuring application middleware. Library configurators do not read nor depend on the *design.modus* file. They generally create data structures to be consumed by software libraries. These data structures are specific to the software library and independent of the hardware. Configuration data is stored in a configurator-specific XML file (for example, *.cybt, *.cyusbdev). Any source code generated by the configurator is stored in a *GeneratedSource* directory in the same directory as the XML file. Library configurators include:

- Bluetooth Configurator: Configure Bluetooth settings. This includes options for specifying what services and profiles to use and what features to offer by creating SDP and/or GATT databases in generated code. This configurator supports both PSoC MCU and WICED Bluetooth applications. See Bluetooth Configurator Guide for more details.
- USB Configurator: Configure USB settings and generate the required firmware. This includes options for defining the 'Device' Descriptor and Settings. See USB Configurator Guide for more details.

3.3.5 Tools

ModusToolbox software includes other tools that provide support for application creation, device firmware updates, and so on. All tools are installed by the ModusToobox Installer. With rare exception each tool has a user guide located in the *docs* directory beside the tool itself. Most user guides are also available online.

Other Tools	Details	Documentation
project-creator	Create a new application. This tool is a stand-alone tool, available as a GUI and a command-line tool (CLI).	User Guide
library-manager	Add, remove, or update libraries and BSP used in an application; edits the makefile	User Guide
fw-loader	Update KitProg communication firmware on a kit. Also available separately on GitHub	*readme.txt* file in the tool directory
cymcuelftool	Merges CM0+ and CM4 application images into a single executable. Typically launched from a post-build script. This tool is not used by most applications.	User Guide is in the tool's *docs* directory
dfuh-tool	Use the Device Firmware Update Host tool to communicate with a PSoC® 6 MCU that has already been programmed with an application that includes device firmware update capability. Provided as a GUI and a command-line tool. Depending on the ecosystem you target, there may be other over-the-air firmware update tools available.	User Guide
cype-tool	The power estimator tool provides an estimate of the power consumed by a target device.	User Guide

3.3.6 Utilities

ModusToolbox software includes some additional utilities that are often necessary for application development. In general you use these utilities transparently.

Utility	Description
GCC	Supported toolchain installed by ModusToolbox.
GDB	The GNU Project Debugger is installed as part of GCC.
OpenOCD	The Open On-Chip Debugger provides a debugging and programming interface for embedded systems.
JRE	Java Runtime Environment; required by various applications and backend processes.

3.4 Enablement Software

This section organizes enablement software in these broad resource categories:

- Code Examples
- Board Support Packages and Kits
- Middleware
- Low-Level Resources

ModusToolbox software targets three primary software development flows:

- PSoC 6 MCU and Bluetooth SoC ecosystem development
- Mbed OS ecosystem development
- Amazon FreeRTOS ecosystem development

As discussed in Build System Infrastructure, to include a resource a starter application specifies a *.lib* file, which provides the URL and commit for the required code. The build system copies the files into your application directory. This means that if you use the Project Creator, all required files appear automatically. However, each software enablement resource from Cypress is available separately in a GitHub repository that typically includes both source code and documentation.

BSPs typically include one or more resources automatically. For example, any BSP that targets a PSoC 6 device includes the PDL driver library automatically. If the board supports CapSense, the BSP includes the CapSense library.

At a higher level, Cypress starter applications (code examples) include the BSP for the supported kit, along with any other required middleware. As a result, the libraries that the example depends upon are brought into the application automatically.

Because the libraries are freely available on GitHub, you may download any library to create a local copy. You can then refer to the library from multiple applications in your development environment, should you prefer.

Some significant resources are available as part of a supported ecosystem and not provided by Cypress. For example, the Mbed Transport Layer Security (TLS) library or the Cordio Bluetooth stack are part of the Mbed ecosystem. You include ecosystem-specific resources using whatever mechanism is defined in that ecosystem.

3.4.1 Code Examples

All current ModusToolbox examples can be found through the GitHub code example page. There you will find links to examples for the Bluetooth SDK, AWS IoT, Mbed OS, and PSoC 6 MCU, among others.

ModusToolbox code examples are example applications. In the ModusToolbox build infrastructure any example application that requires a library downloads that library automatically. Follow the directions in the code example repository to instantiate the example. Instructions vary based on the nature of the application and the targeted ecosystem.

3.4.2 Board Support Packages and Kits

BSPs are aligned with Cypress kits; they provide files for basic device functionality. A BSP typically has a *design.modus* file that configures clocks and other board-specific capabilities. That file is used by the ModusToolbox configurators. A BSP also includes the required device support code for the device on the board. Users can modify the configuration to suit their application. A BSP uses low-level resources to add functionality. For example, a BSP typically adds the following libraries, as appropriate for the kit/device:

- core-lib – to implement return types and generic functionality useful for any kit
- psoc6hal – to implement the ModusToolbox hardware abstraction layer
- psoc6cm0p – to add a predefined CM0+ application
- psoc6make – to implement the build system infrastructure
- capsense – if appropriate for the kit

Application-specific functionality is added by the starter application makefile. For example, if the example uses the RGB LED on a kit, it will include the rgb-led library in its makefile.

Cypress releases BSPs independently of ModusToolbox software as a whole. This search link finds all currently available BSPs on the Cypress GitHub site.

The search results include links to each repository, named TARGET_<kit number>. For example, you will find links to repositories like TARGET_CY8CPROTO-062-4343W. Each repository provides links to relevant documentation. The following links use this BSP as an example. Each BSP has its own documentation.

The information provided varies, but typically includes one or more of:

- an API reference for the BSP
- a link to the associated kit page with kit-specific documentation

A BSP is specific to a board and the device on that board. For custom development, you can create or modify a BSP for your device. See the Boards Support Packages chapter for how BSPs work and how to create your own for a custom board.

3.4.3 Middleware

This category includes any library that implements an API for a particular domain, for example capacitive sensing or an http server. A middleware library may be created by Cypress or come from a third party. Cypress-created middleware may use the Cypress HAL or an LLD directly. In that case, you need the corresponding driver library or BSP for the middleware to work. Any example application that requires a library downloads that library automatically.

The Amazon FreeRTOS ecosystem provides a collection of libraries that provide significant connectivity and other capabilities beyond the FreeRTOS kernel and its internal libraries. This includes interaction with AWS IoT services. Those libraries are not listed here.

Connectivity Middleware	Description	Docs	MCU & BT SOC	Mbed OS	Amazon FreeRTOS
btsdk-audio	The SDK features a dual-mode Bluetooth stack with stack- and profile-level APIs for embedded BT application development. It supports GAP, GATT, SMP, RFCOMM, SDP, AVDT/AVCT and BLE Mesh protocols, as well as over-the-air upgrade. The Bluetooth SDK is factored into a collection of smaller libraries, so that you can download and use those parts of the SDK necessary for your application. Follow instructions in the Project Creator tool to create a local copy of the entire Bluetooth SDK.	btsdk-docs	✓		
btsdk-ble					
btsdk_drivers					
btsdk-hid					
btsdk-include					
btsdk-mesh					
btsdk-ota					
btsdk-rfcomm					
btsdk-tools					
btsdk-utils					
btsdk-host-apps-bt-ble					
btsdk-host-apps-mesh					
btsdk-peer-apps-ble					
btsdk-peer-apps-mesh					
btsdk-peer-apps-ota					
aws-iot	Provides secure, bi-directional communication between Internet-connected devices such as sensors, actuators, embedded micro-controllers, or smart appliances and the AWS Cloud. Supports MQTT and HTTP protocols.	Developer Guide		✓	Available as part of Amazon FreeRTOS
enterprise-security	This library implements a collection of the most commonly used Extensible Authentication Protocols (EAP) used in enterprise WiFi networks	API Reference		✓	
http-server	Provides communication functions for an HTTP server.	GitHub readme		✓	
connectivity-utilities	General purpose middleware connectivity utilities, for instance a linked_list or a json_parser	See the code for each		✓	

Connectivity Middleware	Description	Docs	MCU & BT SOC	Mbed OS	Amazon FreeRTOS
bless	The Bluetooth Low Energy Subsystem (bless) library contains a comprehensive API to configure the BLE Stack and the underlying chip hardware. It incorporates a Bluetooth Core Specification v5.0 compliant protocol stack. You may access the GAP, GATT and L2CAP layers of the stack using the API.	API Reference	✓		
Arm Mbed Cordio	An open source Bluetooth Low Energy (BLE) solution offering both host and controller subsystems, with abstraction interfaces for both RTOS and hardware. It is part of the Mbed OS ecosystem and not provided by Cypress.	Mbed OS documentation		✓	

PSoC 6 Middleware	Description	Docs	MCU	Mbed OS	Amazon FreeRTOS
capsense	Cypress capacitive sensing solution. Capacitive sensing can be used in a variety of applications and products where conventional mechanical buttons can be replaced with sleek human interfaces to transform the way users interact with electronic systems.	API Reference	✓	✓	
csdadc	Enables the ADC or IDAC functionality of the CapSense Sigma-Delta hardware block. Useful for devices that do not include other ADC/IDAC options. The CSD HW block enables multiple sensing capabilities on PSoC devices including self-cap and mutual-cap capacitive touch sensing solutions, a 10-bit ADC, IDAC, and Comparator.	API Reference	✓	✓	
csdidac		API Reference	✓	✓	
usbdev	The USB Device library provides a full-speed USB 2.0 Chapter 9 specification compliant device framework. It uses the USBFS driver from PDL. The middleware supports Audio, CDC, and HID, and other classes. Use the USB Configurator tool to construct the USB Device descriptor	API Reference	✓	✓	
dfu	The Device Firmware Update (DFU) library provides an API for updating firmware images. You can create an application loader to receive and switch to the new application, and a loadable application to be transferred and programmed.	API Reference	✓	✓	
Secure Boot Package	This package includes all required libraries, tools, and sample code to provision and develop applications for PSoC 64 MCUs.	User Guide	✓	✓	
emeeprom	The Emulated EEPROM library provides an API to manage an emulated EEPROM in flash. It has support for wear leveling and restoring corrupted data from a redundant copy.	API Reference	✓	✓	

Other Middleware	Description	Docs	MCU	Mbed OS	Amazon FreeRTOS
freertos	FreeRTOS kernel, distributed as standard C source files with configuration header file, for use with the PSoC 6 MCU.	FreeRTOS web page	✓	✓	
emwin	Segger embedded graphic library and graphical user interface (GUI) framework designed to provide processor- and display controller-independent GUI for any application that needs a graphical display.	Overview	✓	✓	
Arm Mbed TLS	A library to include cryptographic and SSL/TLS capabilities in an embedded application. It is part of the Mbed OS ecosystem and not provided by Cypress.	API Reference		✓	

3.4.4 Low-Level Resources

Low-level resources are related to specific device features. For example, a low-level driver (LLD) contains the API and source code to configure and use a feature or peripheral on a device. ModusToolbox provides the Peripheral Driver Library (PDL) for PSoC 6 devices, or the Wi-Fi Host Driver (WHD) for CYW43xx connectivity devices. Device-specific source code and header files are included in the LLD.

In addition, Cypress provides a hardware abstraction layer (HAL). You can use the HAL for most hardware configuration. It abstracts some of the complexities of using a low level driver directly. For example, a BSP typically uses HAL functions to configure the hardware. In cases where your application requires driver features not supported in the HAL, you can use driver library function calls directly. The HAL and the driver libraries are compatible.

ModusToolbox configurators also generate code (particularly configuration structures) that you can use to configure hardware. In general, the configurators work with the PSoC 6 PDL directly and do not use the HAL. If you use a configurator to configure a peripheral, the HAL will not modify that configuration.

This design means that you can mix and match HAL function calls, direct driver library function calls, and configurator generated source.

The abstraction-rtos library provides a common API that retargets your call to the appropriate RTOS-specific function. This currently supports the FreeRTOS and RTX kernels. Should you wish to use the abstraction-rtos library with a different RTOS, you can examine the API and redirect calls to your RTOS.

Note that ModusToolbox configurators generate PDL-specific configuration structures and function calls. That code requires the PDL to be part of the application. BSPs always include the PDL when necessary.

Item	Details	Docs	MCU & BT SOC	Mbed OS	Amazon FreeRTOS
psoc6hal	The Hardware Abstraction Layer (HAL) provides a high-level interface to configure and use hardware blocks on Cypress MCUs. It is a generic interface that can be used across multiple product families. The focus on ease-of-use and portability means the HAL does not expose all of the low-level peripheral functionality	API Reference	✓	✓	✓
abstraction-rtos	A common API that allows code or middleware to use RTOS features without knowing what the RTOS is	API Reference	✓	✓	✓
core-lib	Provides header files that declare basic types and utilities (such as result types or ASSERT) that can be used by multiple BSPs	API Reference	✓	✓	✓
retarget-io	Provides a board-independent API to retarget text input/output to a serial UART on a kit	API Reference	✓	✓	✓
rgb-led	Provides a board-independent API to use the RGB LED on a kit	API Reference	✓	✓	✓
serial-flash	Provides a board-independent API to use the serial flash on a kit	API Reference	✓	✓	✓
psoc6pdl	Peripheral driver library for PSoC 6 devices. The library is device-independent, so can be precompiled and used for any PSoC 6 MCU device or application. Included automatically by any BSP targeting a PSoC 6 device	API Reference	✓	✓	✓
wifi-host-driver	Driver library for Cypress WLAN devices (CYW43xxx) that can be easily ported to popular RTOSs such as Amazon FreeRTOS and Mbed OS. The wifi-host-driver is included automatically by board support packages that require this driver (those with CYW43xx devices)	API Reference	✓	✓	✓
psoc6cm0p	Prebuilt application images for the Cortex M0+ CPU of the dual-CPU PSoC 6 devices. The images are provided as C arrays ready to be compiled as part of the Cortex M4 application. The Cortex M0+ application code is placed to internal flash by the Cortex M4 linker script.	See the readme file in the repository	✓	✓	✓
psoc6make	This repository provides the build recipe makefiles and scripts for building and programming PSoC 6 applications. You can build an application either through a command-line interface (CLI), the Eclipse IDE, or a third-party IDE.	See the readme file in the repository	✓	✓	✓

Item	Details	Docs	MCU & BT SOC	Mbed OS	Amazon FreeRTOS
Mbed OS HAL	Mbed OS hardware abstraction layer. Support for Cypress targets is available from the Mbed OS archive. This HAL is part of the Mbed ecosystem and not provided directly by Cypress.	Mbed API documentation		✓	

4 ModusToolbox Build System

This chapter covers various aspects of the ModusToolbox build system. Refer to Using the Command Line for getting started information about using the command line tools. This chapter is organized as follows:

- Overview
- Application Types
- BSPs
- make getlibs
- Adding source files
- Pre-builds and post-builds
- Program and debug
- Available make targets
- Available make variables

4.1 Overview

The ModusToolbox build system is based on GNU make. It performs application builds and provides the logic required to launch tools and run utilities. It consists of a light and accessible set of makefiles deployed as part of every application. This structure allows each application to own the build process, and it allows environment-specific or application-specific changes to be made with relative ease. The system runs on any environment that has the make and git utilities. For a "how to" document about the ModusToolbox makefile system, refer to https://community.cypress.com/docs/DOC-18994. Also, as described in the Getting Started chapter, you can run the `make help` command to get details on the various targets and variables available.

The ModusToolbox Command Line Interface (CLI) and supported IDEs all use the same build system. Hence, switching between them is fully supported. Program/Debug and other tools can be used in either the command line or an IDE environment. In all cases, the build system relies on the presence of ModusToolbox tools included with the ModusToolbox installer.

The tools contain a *start.mk* file that serves as a reference point for setting up the environment before executing the recipe-specific build in the base library. The file also provides a `getlibs` make target that brings libraries into an application. Every application must then specify a target board on which the application will run. These are provided by the *<BSP>.mk* files deployed as a part of a board support package (BSP) library.

The majority of the makefiles are deployed as git repositories (called "repos"), in the same way that libraries are deployed in the ModusToolbox software. The library that contains the recipe makefiles is referred to as the "base library." This is the minimum required library to enable an application build. Together, these makefiles form the build system.

4.2 Application Types

The build system supports the following application types:

- Normal app – The application consists of one application makefile. The build process creates one artifact. All prebuilt libraries are brought in at link time. A normal application is constructed by defining the APPNAME variable in the application makefile.
- Library app – The application consists of one application makefile. The sources are built into a library. These libraries may be linked in as part of a Normal app build. A library application is constructed by defining the LIBNAME variable in the application makefile.

The library apps are usually placed as companions to normal apps. These normal apps specify their dependency on library apps by including them in the `SEARCH_LIBS_AND_INCLUDES` make variable. They also drive the build process of the library apps by defining a `shared_libs` make target. For example:

```
SEARCH_LIBS_AND_INCLUDES=../bspLib
shared_libs:
    make -C ../bspLib build -j
```

4.3 BSPs

An application must specify a target BSP through the `TARGET` variable in the makefile. Cypress provides reference BSPs for its development kits. Use these as a reference to construct your own BSP. For more information about BSPs, refer to the Board Support Packages chapter.

Use the Library Manager to add, update, or remove a BSP from an application. You can also add a *.lib* file that contains the URL and a version tag of interest in the application.

4.4 make getlibs

When you run the `make getlibs` command, the build system finds all the *.lib* files in the application directory and performs `git clone` operations on them. A *.lib* file contains a git URL to a library repo, and a specific tag for a version of the code.

The `getlibs` target finds and processes all *.lib* files and uses the `git` command to clone or pull the code as appropriate. Then, it checks out the specific tag listed in the *.lib* file. The Project Creator and Library Manager invoke this process automatically.

- The `getlibs` target must be invoked separately from any other make target (for example, the command `make getlibs build` is not allowed and the makefiles will generate an error; however, a command such as `make clean build` is allowed).
- The `getlibs` target performs a `git fetch` on existing libraries but will always checkout the tag pointed to by the overseeing *.lib* file.
- The `getlibs` target detects if users have modified the Cypress code and will not overwrite their work. This allows you to perform some action (for example commit code or revert changes, as appropriate) instead of overwriting the changes.

The build system also has a `printlibs` target that can be used to print the status of the cloned libraries.

4.4.1 repos

The cloned libraries are located in their individual git repos in the directory pointed to by the `CY_GETLIBS_PATH` variable (for example, */deps*). These all point to the “cypress” remote origin. You can point your repo by editing the *.git/config* file or by running the `git remote` command.

If the repos are modified, add the changes to your source control (git branch is recommended). When `make getlibs` is run (to either add new libraries or update libraries), it requires the repos to be clean. You may also use the *.gitignore* file for adding untracked files when running `make getlibs`.

4.5 Adding source files

Source and header files placed in the application directory hierarchy are automatically added by the auto-discovery mechanism. Similarly, library archives and object files are automatically added to the application. Any object file not referenced by the application is discarded by the linker.

The application makefile can also include specific source files (`SOURCES`), header file locations (`INCLUDES`) and prebuilt libraries (`LDLIBS`). This is useful when the files are located outside of the application directory hierarchy or when specific sources need to be included from the filtered directories.

4.5.1 Auto-Discovery

The build system implements auto-discovery of Cypress library files, source files, header files, object files, and pre-built libraries. If these files follow the specified rules, they are guaranteed to be brought into the application build automatically. Auto-discovery searches for all supported file types in the application directory hierarchy and performs filtering based on a directory naming convention and specified directories, as well as files to ignore. If files external to the application directory hierarchy need to be added, they can be specified using the `SOURCES`, `INCLUDES`, and `LIBS` make variables.

Auto-discovery of source code (source and headers) has no depth limit (it uses the “find” tool). Auto-discovery of other types of files do have a depth limit, including:

- .lib file depth
- .mk file of the selected TARGET
- device support library discovery
- configurator file discovery

The default depth limit for these files is five directories deep. They can be changed to up to nine directories deep by setting the following options in the makefile:

```
CY_UTILS_SEARCH_DEPTH=9
CY_LIBS_SEARCH_DEPTH=9
```

To control which files are included/excluded, the build system implements a filtering mechanism based on directory names and *.cyignore* files.

4.5.1.1 .cyignore

Prior to applying auto-discovery and filtering, the build system will first search for *.cyignore* files and construct a set of directories and files to exclude. It contains a set of directories and files to exclude, relative to the location of the *.cyignore* file. The `CY_IGNORE` variable can also be used in the makefile to define directories and files to exclude.

Note The `CY_IGNORE` variable should contain paths that are relative to the application root. For example, *./src1*.

4.5.1.2 TOOLCHAIN_<NAME>

Any directory that has the prefix “TOOLCHAIN_” is interpreted as a directory that is toolchain specific. The “NAME” corresponds to the value stored in the `TOOLCHAIN` make variable. For example, an IAR-specific set of files is located under a directory named *TOOLCHAIN_IAR*. Auto-discovery only includes the *TOOLCHAIN_<NAME>* directories for the specified `TOOLCHAIN`. All others are ignored.

4.5.1.3 TARGET_<NAME>

Any directory that has the prefix “TARGET_” is interpreted as a directory that is target specific. The “NAME” corresponds to the value stored in the `TARGET` make variable. For example, a build with `TARGET=CY8CPROTO-062-4343W` ignores all *TARGET_* directories except *TARGET_CY8CPROTO-062-4343W*.

Note The TARGET_ directory is often associated with the BSP, but it can be used in a generic sense. E.g. if application sources need to be included only for a certain TARGET, this mechanism can be used to achieve that.

Note The output directory structure includes the *TARGET* name in the path, so you can build for target A and B and both artifact files will exist on disk.

4.5.1.4 CONFIG_<NAME>

Any directory that has the prefix “CONFIG_” is interpreted as a directory that is configuration (Debug/Release) specific. The “NAME” corresponds to the value stored in the `CONFIG` make variable. For example, a build with `CONFIG=CustomBuild` ignores all *CONFIG_* directories, except *CONFIG_CustomBuild*.

Note The output directory structure includes the *CONFIG* name in the path, so you can build for config A and B and both artifact files will exist on disk.

4.5.1.5 COMPONENT_<NAME>

Any directory that has the prefix “COMPONENT_” is interpreted as a directory that is component specific. The “NAME” corresponds to the value stored in the `COMPONENT` make variable. For example, consider an application that sets `COMPONENTS+=comp1`. Also assume that there are two directories containing component-specific sources:

```
COMPONENT_comp1/src.c
COMPONENT_comp2/src.c
```

Auto-discovery will only include *COMPONENT_comp1/src.c* and ignore *COMPONENT_comp2/src.c*. If a specific component needs to be removed, either delete it from the `COMPONENTS` variable or add it to the `DISABLE_COMPONENTS` variable.

4.5.1.6 BSP Makefile

Auto-discovery will also search for a <*TARGET*>*.mk* file (aka BSP makefile). If no matching *TARGET* makefile is found, it will fail to build. This makefile can also be manually specified by setting it in the `CY_EXTRA_INCLUDES` variable.

4.6 Pre-builds and Post-builds

A pre-build or post-build operation is typically a script file invoked by the build system. Such operations are possible at several stages in the build process. They can be specified at the application, BSP, and recipe levels.

You can pre-build and post-build arguments in the application makefile. For example:

```
PREBUILD=command -arg1 -arg2
```

If you want to run more than one command, separate them with a semicolon (;). For example:

```
PREBUILD=command1 -arg1; command2 -arg1 -arg2
```

The sequence of execution in a build is as follows:

1. BSP pre-build – Defined using `CY_BSP_PREBUILD` variable.
2. Application pre-build – Defined using `PREBUILD` variable.
3. Source generation – Defined using `CY_RECIPE_GENSRC` variable.
4. Recipe pre-build – Defined using `CY_RECIPE_PREBUILD` variable.
5. Source compilation and linking.
6. Recipe post-build – Defined using `CY_RECIPE_POSTBUILD` variable.
7. BSP post-build – Defined using `CY_BSP_POSTBUILD` variable.
8. Application post-build – Defined using `POSTBUILD` variable.

The variable value is the relative path to the script to be executed.

Note Pre-builds happen after the auto-discovery process. Therefore, if the pre-build steps generate any source files to be included in a build, they will not be automatically included until the subsequent build. In this scenario, this step should use the `$(shell)` function directly in the application makefile instead of using the provided pre-build make variables. For example:

```
$(shell bash ./custom_gen.sh ARG1 ARG2)
```

4.7 Program and debug

The programming step can be done through the CLI by using the following make targets:

- `program` – Build and program the board.
- `qprogram` – Skip the build step and program the board.
- `debug` – Build and program the board. Then launch the GDB server.
- `qdebug` – Skip the build and program steps. Just launch the GDB server.
- `attach` – Starts a GDB client and attaches the debugger to the running target.

For CLI debugging, the attach target must be run on a separate shell instance. Use the GDB commands to debug the application.

4.8 Available Make Targets

A make target specifies the type of function or activity that the make invocation executes. The build system does not support a make command with multiple targets. Therefore, a target must be called in a separate make invocation. The following tables list and describe the available make targets for all recipes.

4.8.1 General Make Targets

Target	Description
`all`	Same as build. That is, builds the application. This target is equivalent to the `build` target.
`getlibs`	Clones the repositories and checks out the identified commit. The repos are cloned to the libs directory. By default, this directory is created in the application directory. It may be directed to other locations using the `CY_GETLIBS_PATH` variable.
`build`	Builds the application. The build process involves source auto-discovery, code-generation, pre-builds, and post-builds. For faster incremental builds, use the `qbuild` target to skip the auto-discovery step.
`qbuild`	Quick builds the application using the previous build's source list. When no other sources need to be auto-discovered, this target can be used to skip the auto-discovery step for a faster incremental build.
`program`	Builds the artifact and programs it to the target device. The build process performs the same operations as the build target. Upon successful completion, the artifact is programmed to the board.
`qprogram`	Quick programs a built application to the target device without rebuilding. This target allows programming an existing artifact to the board without a build step.
`debug`	Builds and programs. Then launches a GDB server. Once the GDB server is launched, another shell should be opened to launch a GDB client.
`qdebug`	Skips the build and program step and does Quick Debug; that is, it launches a GDB server. Once the GDB server is launched, another shell should be opened to launch a GDB client.
`clean`	Cleans the /build/<TARGET> directory. The directory and all its contents are deleted from disk.
`attach`	Starts a GDB client and attaches the debugger to the running target.
`help`	Prints the help documentation. Use the `CY_HELP=<name of target or variable>` to see the verbose documentation for a given target or a variable.

4.8.2 IDE Make Targets

Target	Description
`eclipse`	Generates Eclipse IDE launch configs (preliminary: Eclipse application). This target expects the `CY_IDE_PRJNAME` variable to be set to the name of the application as defined in the eclipse IDE. For example, `make eclipse CY_IDE_PRJNAME=AppV1`. If this variable is not defined, it will use the `APPNAME` for the launch configs. This target also generates *.cproject* and *.project* files if they do not exist in the application root directory.
`vscode`	Generates VS Code IDE json files (preliminary). This target generates VS Code json files for debug/program launches, IntelliSense, and custom tasks. These overwrite the existing files in the application directory except for *settings.json*.
`ewarm8`	Generates IAR-EW version 8 IDE .ipcf file (preliminary). This target generates an IAR Embedded Workbench v8.x compatible .ipcf file that can be imported into IAR-EW. The .ipcf file is overwritten every time this target is run. **Note** Application generation requires python3 to be installed and present in the PATH variable.
`uvision5`	Generates CMSIS PDSC files for µVision v5. This target generates a CMSIS compatible .cpdsc and .gpdsc files that can be imported into Keil µVision v5. Both files are overwritten every time this target is run. **Note** Application generation requires python3 to be installed and present in the PATH variable.

4.8.3 Tools Make Targets

Target	Description
`open`	Opens/launches a specified tool. This is intended for use by the Eclipse IDE. Use `make config`, `config_bt`, or `config_usbdev` instead. This target accepts two variables: `CY_OPEN_TYPE` and `CY_OPEN_FILE`. At least one of these must be provided. The tool can be specified by setting the `CY_OPEN_TYPE` variable. A specific file can also be passed using the `CY_OPEN_FILE` variable. If only `CY_OPEN_FILE` is given, the build system will launch the default tool associated with the file's extension.
`modlibs`	Launches the library-manager executable for updating libraries. The Library Manager can be used to add/remove libraries and to upgrade/downgrade existing libraries.
`config`	Runs the Device Configurator on the target **.modus* file. If no existing device-configuration files are found, the configurator is launched to create one.
`config_bt`	Runs the Bluetooth Configurator on the target **.cybt* file. If no existing bt-configuration files are found, the configurator is launched to create one.
`config_usbdev`	Runs the USB Configurator on the target **.cyusbdev* file. If no existing usbdev-configuration files are found, the configurator is launched to create one.

4.8.4 Utility Make Targets

Target	Description
progtool	Performs specified operations on the programmer/firmware-loader. This target expects user-interaction on the shell while running it. When prompted, you must specify the command(s) to run for the tool.
bsp	Generates a TARGET_GEN board/kit from TARGET. This target generates a new Board Support Package with the name provided in TARGET_GEN based on the current TARGET. The TARGET_GEN variable must be populated with the name of the new TARGET. Optionally, you may define the target device (DEVICE_GEN) and additional devices (ADDITIONAL_DEVICES_GEN) such as radios. For example: `make bsp TARGET_GEN=NewBoard DEVICE_GEN=CY8C624ABZI-D44 ADDITIONAL_DEVICES_GEN=CYW4343WKUBG`
check	Checks for the necessary tools. Not all tools are necessary for every build recipe. This target allows you to get an idea of which tools are missing if a build fails in an unexpected way.
get_app_info	Prints the app info for the eclipse IDE. As with the get_cfg_file target, the file types can be specified by setting the CY_CONFIG_FILE_EXT variable. For example, `make get_app_info CY_CONFIG_FILE_EXT="modus cybt cyusbdev"`
get_env_info	Prints the make, git, and, app repo info. This allows a quick printout of the current app repo and the make and git tool locations and versions.
printlibs	Prints the status of the library repos. This target parses through the library repos and prints the SHA1 commit. It also shows whether the repo is clean (no changes) or dirty (modified or new files).

4.9 Available Make Variables

The following make variables provide access to most of the available features to customize your build. They can either be defined in the application makefile or be passed through the make invocation. For example:

```
make build TOOLCHAIN=GCC_ARM CONFIG=CustomConfig -j8
```

4.9.1 Basic Configuration Make Variables

Variable	Description
TARGET	Specifies the target board/kit (that is, BSP). For example, CY8CPROTO-062-4343W.
APPNAME	Specifies the name of the application. For example, AppV1. This variable signifies that the application builds an artifact intended for a target board. For applications that need to build into an archive (library), use the LIBNAME variable. **Note** This variable may also be used when generating launch configs when invoking the eclipse target.
LIBNAME	Specifies the name of the library application. For example, LibV1. This variable is used to set the name of the application artifact (prebuilt library). It also signifies that the application will build an archive (library) that is intended to be linked to another application. These library applications can be added as dependencies to an artifact producing application using the SEARCH_LIBS_AND_INCLUDES variable.
TOOLCHAIN	Specifies the toolchain used to build the application. For example, GCC_ARM. Supported toolchains for this include GCC_ARM, IAR, and ARM.
CONFIG	Specifies the configuration option for the build [Debug Release]. The CONFIG variable is not limited to Debug/Release. It can be other values. However in those instances, the build system will not configure the optimization flags. Debug=lowest optimization, Release=highest optimization. The optimization flags are toolchain specific. If you go with your custom config then you can manually set the optimization flag in the CFLAGS.
VERBOSE	Specifies whether the build is silent [false] or verbose [true]. Setting VERBOSE to true may help in debugging build errors/warnings.

4.9.2 Advanced Configuration Make Variables

Variable	Description
SOURCES	Specifies C/C++ and assembly files not under the working directory. This can be used to include files external to the application directory.
INCLUDES	Specifies include paths not under the working directory. **Note** These MUST NOT have `-I` prepended.
DEFINES	Specifies additional defines passed to the compiler. **Note** These MUST NOT have `-D` prepended.
VFP_SELECT	Selects hard/soft ABI for floating-point operations [softfp hardfp]. If not defined, this value defaults to softfp.
CFLAGS	Prepends additional C compiler flags. **Note** If the entire C compiler flags list needs to be replaced, define the `CY_RECIPE_CFLAGS` make variable with the desired C flags.
CXXFLAGS	Prepends additional C++ compiler flags. **Note** If the entire C++ compiler flags list needs to be replaced, define the `CY_RECIPE_CXXFLAGS` make variable with the desired C++ flags.
ASFLAGS	Prepends additional assembler flags. **Note** If the entire assembler flags list needs to be replaced, define the `CY_RECIPE_ASFLAGS` make variable with the desired assembly flags.
LDFLAGS	Prepends additional linker flags. **Note** If the entire linker flags list needs to be replaced, define the `CY_RECIPE_LDFLAGS` make variable with the desired linker flags.
LDLIBS	Includes application-specific prebuilt libraries. **Note** If additional libraries need to be added using `-l` or `-L`, add to the `CY_RECIPE_EXTRA_LIBS` make variable.
LINKER_SCRIPT	Specifies a custom linker script location. This linker script overrides the default. **Note** Additional linker scripts can be added for GCC via the `LDFLAGS` variable as a `-L` option.
PREBUILD	Specifies the location of a custom pre-build step and its arguments. This operation runs before the build recipe's pre-build step. **Note** BSPs can also define a pre-build step. This runs before the application pre-build step. If the default pre-build step needs to be replaced, define the `CY_RECIPE_PREBUILD` make variable with the desired pre-build step.
POSTBUILD	Specifies the location of a custom post-build step and its arguments. This operation runs after the build recipe's post-build step. **Note** BSPs can also define a post-build step. This runs before the application post-build step. **Note** If the default post-build step needs to be replaced, define the `CY_RECIPE_POSTBUILD` make variable with the desired post-build step.
COMPONENTS	Adds component-specific files to the build. Create a directory named *COMPONENT_<VALUE>* and place your files. Then provide `<VALUE>` to this make variable to have that feature library be included in the build. For example, create a directory named *COMPONENT_ACCELEROMETER*. Then include it in the make variable: `COMPONENT=ACCELEROMETER`. If the make variable does not include the `<VALUE>`, then that library will not be included in the build. **Note** If the default `COMPONENT` list must be overridden, define the `CY_COMPONENT_LIST` make variable with the list of component values.
DISABLE_COMPONENTS	Removes component-specific files from the build. Include a `<VALUE>` to this make variable to have that feature library be excluded in the build. For example, to exclude the contents of the *COMPONENT_BSP_DESIGN_MODUS* directory, set `DISABLE_COMPONENTS=BSP_DESIGN_MODUS`.

Variable	Description

SEARCH_LIBS_AND_INCLUDES	List of dependent library application paths. For example, *../bspLib*.
	An artifact-producing application (defined by setting APPNAME) can have a dependency on library applications (defined by setting LIBNAME). This variable defines those dependencies for the artifact-producing application. The actual build invocation of those libraries is handled at the application level by defining the shared_libs target. For example:
	`shared_libs:`
	`  make -C ../bspLib build -j`

4.9.3 BSP Make Variables

Variable	Description
DEVICE	Device ID for the primary MCU on the target board/kit. The device identifier is mandatory for all board/kits.
ADDITIONAL_DEVICES	IDs for additional devices on the target board/kit. These include devices such as radios on the board/kit. This variable is optional.
TARGET_GEN	Name of the new target board/kit. This is a mandatory variable when calling the bsp make target. It is used to name the board/kit files and directory.
DEVICE_GEN	(Optional) Device ID for the primary MCU on the new target board/kit. This is an optional variable when calling the bsp make target to replace the primary MCU on the board (overwrites DEVICE). If it is not defined, the new board/kit will use the existing DEVICE from the board/kit that it is copying from.
ADDITIONAL_DEVICES_GEN	(Optional) IDs for additional devices on the new target board/kit. This is an optional variable when calling the bsp make target to replace the additional devices on the board (overwrites ADDITIONAL_DEVICES). If it is not defined, the new board/kit will use the existing ADDITIONAL_DEVICES from the board/kit that it is copying from.

4.9.4 Getlibs Make Variables

Variable	Description
CY_GETLIBS_NO_CACHE	Disables the cache when running getlibs. To improve the library creation time, the getlibs target uses a cache located in the user's home directory ($HOME for macOS/Linux and $USERPROFILE for Windows). Disabling the cache allows 3rd-party libraries to be brought in to the application using .lib files just like the Cypress libraries.
CY_GETLIBS_OFFLINE	Use the offline location as the library source. Setting this variable signals to the build system to use the offline location (Default: *<HOME>/.modustoolbox/offline*) when running the "getlibs" target. The location of the offline content can be changed by defining the CY_GETLIBS_OFFLINE_PATH variable.
CY_GETLIBS_PATH	Absolute path to the intended location of libs directory. The library repos are cloned into a directory named, *libs* (default: *<CY_APP_PATH>/libs*). Setting this variable allows specifying the location of the *libs* directory to be elsewhere on disk.
CY_GETLIBS_DEPS_PATH	Absolute path to where the library-manager stores .lib files. Setting this path allows relocating the directory that the library-manager uses to store the .lib files in your application. The default location is in a directory named */deps* (Default: *<CY_APP_PATH>/deps*). **Note** This variable requires ModusToolbox tools_2.1 or higher.

Variable	Description
CY_GETLIBS_CACHE_PATH	Absolute path to the cache directory. The build system caches all cloned repos in a directory named */cache* (Default: *<HOME>/.modustoolbox/cache*). Setting this variable allows the cache to be relocated to elsewhere on disk. To disable the cache entirely, set the CY_GETLIBS_NO_CACHE variable to [true]. **Note** This variable requires ModusToolbox tools_2.1 or higher.
CY_GETLIBS_OFFLINE_PATH	Absolute path to the offline content directory. The offline content is used to create/update applications when not connected to the internet (Default: *<HOME>/.modustoolbox/offline*). Setting this variable allows to relocate the offline content to elsewhere on disk. **Note** This variable requires ModusToolbox tools_2.1 or higher.
CY_GETLIBS_SEARCH_PATH	Relative path to the top directory for getlibs operation. The getlibs operation by default executes at the location of the CY_APP_PATH. This can be overridden by specifying this variable to point to a specific location.

4.9.5 Path Make Variables

Variable	Description
CY_APP_PATH	Relative path to the top-level of application. For example, *./* Settings this path to other than *./* allows the auto-discovery mechanism to search from a root directory location that is higher than the app directory. For example, CY_APP_PATH=../../ allows auto-discovery of files from a location that is two directories above the location of *./makefile*.
CY_BASELIB_PATH	Relative path to the base library. For example, *./libs/psoc6make* This directory must be relative to CY_APP_PATH. It defines the location of the library containing the recipe makefiles, where the expected directory structure is *<CY_BASELIB_PATH>/make*. All applications must set the location of the base library.
CY_EXTAPP_PATH	Relative path to an external app directory. For example, *../external* This directory must be relative to CY_APP_PATH. Setting this path allows incorporating files external to CY_APP_PATH. For example, CY_EXTAPP_PATH=../external lets auto-discovery pull in the contents of *../external* directory into the build. **Note** This variable is only supported in CLI. Use the shared_libs mechanism and CY_HELP_SEARCH_LIBS_AND_INCLUDES for tools and IDE support.
CY_DEVICESUPPORT_PATH	Relative path to the *devicesupport.xml* file. This path specifies the location of the *devicesupport.xml* file for the Device Configurator. It is used when the configurator needs to be run in a multi-app scenario.
CY_SHARED_PATH	Relative path to the location of shared *.lib* files. This variable is used in shared library applications to point to the location of external *.libs* files.
CY_COMPILER_PATH	Absolute path to the compiler (default: GCC_ARM in CY_TOOLS_DIR). Setting this path allows custom toolchains to be used instead of the defaults. This should be the location of the */bin* directory containing the compiler, assembler, and linker. For example: `CY_COMPILER_PATH="C:/Program Files (x86)/IAR Systems/Embedded Workbench 8.2/arm/bin"`
CY_TOOLS_DIR	Absolute path to the tools root directory. Applications must specify the *tools_<version>* directory location, which contains the root makefile and the necessary tools and scripts to build an application. Application makefiles are configured to automatically search in the standard locations for various platforms. If the tools are not located in the standard location, you may explicitly set this.
CY_BUILD_LOCATION	Absolute path to the build output directory (default: pwd/build). The build output directory is structured as */TARGET/CONFIG/*. Setting this variable allows the build artifacts to be located in the directory pointed to by this variable.

Variable	Description
CY_PYTHON_PATH	Specifies the path to the Python executable. For make targets that depend on Python, the build system looks for a Python 3 in the user's PATH and uses that to invoke python. If however CY_PYTHON_PATH is defined, it will use that python executable.
TOOLCHAIN_MK_PATH	Specifies the location of a custom *TOOLCHAIN.mk* file. Defining this path allows the build system to use a custom *TOOLCHAIN.mk* file pointed to by this variable. **Note** The make variables in this file should match the variables used in existing *TOOLCHAIN.mk* files.

4.9.6 Miscellaneous Make Variables

Variable	Description
CY_IGNORE	Adds to the directory and file ignore list. E.g. ./file1.c ./inc1 Directories and files listed in this variable are ignored in auto-discovery. This mechanism works in combination with any existing *.cyignore* files in the application.
CY_SKIP_RECIPE	Skip including the recipe makefiles. This allows the application to not include any recipe makefiles and only include the *start.mk* file from the tools install.
CY_EXTRA_INCLUDES	Specifies additional makefiles to add to the build. The application makefile cannot add additional makefiles directly. Instead, use this variable to include these in the build. For example: `CY_EXTRA_INCLUDES=./custom1.mk ./custom2.mk`
CY_LIBS_SEARCH_DEPTH	Directory search depth for *.lib* files (default: 5). This variable controls how deep the search mechanism in `getlibs` looks for *.lib* files. **Note** Deeper searches take longer to process.
CY_UTILS_SEARCH_DEPTH	Directory search depth for *.cyignore* and *TARGET.mk* files (default: 5). This variable controls how deep the search mechanism looks for *.cyignore* and *TARGET.mk* files. Min=1, Max=9. **Note** Deeper searches take longer to process.
CY_IDE_PRJNAME	Name of the Eclipse IDE application. This variable can be used to define the file and target application name when generating Eclipse launch configurations in the `eclipse` target.
CY_CONFIG_FILE_EXT	Specifies the configurator file extension. For example, *.modus. This variable accepts a space-separated list of configurator file extensions to search when running the `get_cfg_file` and `get_app_info` targets.
CY_SUPPORTED_TOOL_TYPES	Defines the supported tools for a BSP. BSPs can define the supported tools that can be launched using the `open` target.

5 Board Support Packages

5.1 Overview

A BSP provides a standard interface to a board's features and capabilities. The API is consistent across Cypress kits. Other software (such as middleware or an application) can use the BSP to configure and control the hardware. BSPs do the following:

- initialize device resources, such as clocks and power supplies to set up the device to run firmware.
- contain default linker scripts and startup code that you can customize for your board.
- contain the hardware configuration (structures and macros) for both device peripherals and board peripherals.
- provide abstraction to the board by providing common aliases or names to refer to the board peripherals, such as buttons and LEDs.
- include the libraries for the default capabilities on the board. For example, the BSP for a kit with CapSense capabilities includes the CapSense library.

5.2 What's in a BSP

This section presents a quick overview of the key resources that are part of a BSP. The contents may vary for different environments.

Each BSP is included in a directory that starts with "TARGET_" such as TARGET_CY8CKIT-062-WIFI-BT or TARGET_CYW920819EVB-02. A basic BSP contains the following:

- *<BSP_NAME>.mk* – This file defines the `DEVICE` and other BSP-specific make variables such as `COMPONENTS`. These are described in the ModusToolbox Build System chapter. It also defines board-specific information such as the device ID, compiler and linker flags, pre-builds/post-builds, and components used with this board implementation.
- *COMPONENT_BSP_DESIGN_MODUS/design.modus* – This is a configuration file (other types may also exist in a BSP) used to define the board peripherals and system settings using a graphical configuration tool.

  Note The "COMPONENT_BSP_DESIGN_MODUS" directory may not exist on all BSPs.
- README.md – A readme file that describes the board.

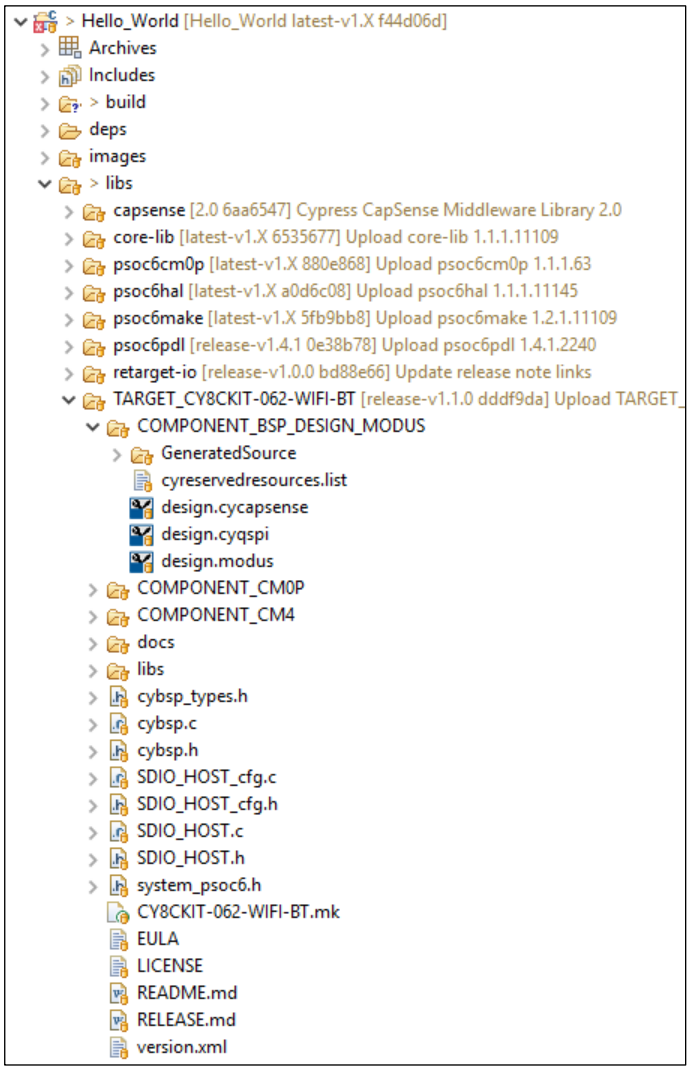
5.2.1 PSoC 6 vs. WICED Bluetooth

BSPs for PSoC 6 and WICED Bluetooth essentially do the same things in the same ways. The main difference between them is where they are located. PSoC 6 BSPs are located inside the project's */libs* directory. WICED Bluetooth BSPs are located in the wiced_btsdk project, under *dev-kit/bsp*.

Also, because of the structure of the wiced_btsdk project and its relationship to various WICED Bluetooth applications, it contains all the BSPs. PSoC 6 projects generally only contain one or two BSPs.

5.3 PSoC 6 BSPs

The following shows a typical PSoC 6 BSP:



5.3.1 cybsp.c /.h & cybsp_types.h

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 `cybsp_init ()` from *cybsp.c* to initialize the board.

The *cybsp_types.h* file contains the aliases (macro definitions) for all the board resources.

5.3.2 linker

Linker scripts for all supported toolchains: GCC ARM, IAR, and ARM.

5.3.3 startup

Contains the startup code with the reset handler for the target device, written in assembly language for all supported toolchains: GCC ARM, IAR, and ARM.

5.3.4 COMPONENT_BSP_DESIGN_MODUS

This directory contains the configuration files (such as *design.modus*) for use with various BSP configurator tools, including Device Configurator, QSPI Configurator, and CapSense Configurator. At the start of a build, the build system invokes these tools to generate the source files in the *GeneratedSource* directory. See Overriding the BSP Configuration Files to learn how the application can override this component.

5.3.5 deps

For newer ModusToolbox applications, this directory contains*.lib* files that specify the required libraries for the underlying device family. These are provided by Cypress and always point to a Cypress owned repo or a repo that is sanctioned by Cypress. You can use the `make getlibs` command to fetch these libraries. The Project Creator and Library Manager tools invoke `make getlibs` automatically.

These files can be modified to point to specific tags or rerouted to internal repos. Cypress uses tags to denote versions when publishing content. To lock down to a certain version of a library, ensure that the tag in the *.lib* file points to a specific version. You may use the library manager to achieve this.

5.3.6 libs

This directory stores the imported library files after running `make getlibs` to process the *.lib* files. For older versions of ModusToolbox applications, this directory also contains *.lib* files. The build system supports both.

All libraries are cloned by default into the *libs* directory in the application root. This location can be modified by specifying the `CY_GETLIBS_PATH` variable. Duplicate libraries are checked to see if they point to the same commit and if so, only one copy is kept in the *libs* directory.

5.3.7 Board Initialization

The *cycfg.c* file generated by the Device Configurator contains a routine `init_cycfg_all()`. This function calls several other routines that initialize the device with the configuration defined through the Device Configurator.

The `cybsp_init()` function in the *cybsp.c* file does not call `init_cycfg_all()`. It calls the `init_cycfg_system()` function to initialize the device resources such as clocks and power supplies. You can call the other routines (for example, `init_cycfg_routing()`) in `main()` if required.

Note that you need to call `init_cycfg_routing()` when you use analog resources such as CapSense and SAR ADC. See the corresponding code examples at the Cypress GitHub webpage for details.

5.3.8 Overriding the BSP Configuration Files

The BSP sets up a default board configuration in the *design.modus* file. Depending on the board, it may also set up other configuration files such as CapSense and SegLCD.

When you create an application that uses a BSP, the configuration files such as *design.modus* file are copied from the original BSP repository and put into your application. You can open the local copy of the file with the appropriate configurator and modify the configuration. However, if you recreate the application, you get a new copy of the original files and your changes are lost.

Rather than make local changes, if necessary, you can override the files in the BSP_DESIGN_MODUS component with a different configuration. Cypress code examples often override this component to use a different configuration than the one provided by a BSP. To do this, follow these steps:

1. Add the following line to the makefile in the application directory. This prevents the build from including the default configuration files from the BSP.
   ```
   DISABLE_COMPONENTS+=BSP_DESIGN_MODUS
   ```
2. Create a directory for each target that you want to support at the top-level directory in your application. The directory name must be *TARGET_(board_name)*. For example, *TARGET_CY8CPROTO-062-4343W*.

   Remember that the build system automatically includes all the source files inside a directory that begins with *TARGET_* followed by the target name for compilation when that target is specified in the application makefile.
3. Copy the *design.modus* file and other configuration files inside this new directory and customize as required. When you save the changes in the configuration file(s), the source files are generated and placed under the *GeneratedSource* directory.

Another way to override the configuration is to use an existing *TARGET* board, but update the *design.modus* configuration. This can be updated directly, or another file can be used instead of the default. For the latter case:

1. Copy the COMPONENT_BSP_DESIGN_MODUS directory to another directory (e.g. `COMPONENT_CUSTOM_DESIGN_MODUS`)
2. Set the make variable `DISABLE_COMPONENTS=BSP_DESIGN_MODUS` in the application makefile to disable the inclusion of the default *design.modus* and its generated sources into the build.
3. Add a `COMPONENTS` variable in the application makefile with the name of the directory (excluding the prefix of `COMPONENTS_`) containing your custom copy of the configuration directory (e.g. `COMPONENTS+= CUSTOM_DESIGN_MODUS`). (**Note** This mechanism is not applicable for BSPs that do not have the `COMPONENT_BSP_DESIGN_MODUS` directory).

5.3.9 Creating a BSP for Your Board

Cypress provides BSPs for its boards. When you design your own board, it is likely to have a different MCU, board peripherals, and other hardware features. You can create a custom BSP for your board to simplify interacting with your board's features.

To create your own board, do the following:

1. Locate the closest-matching BSP to your custom BSP and set that as the default `TARGET` for the project in the makefile.
2. Run the `make bsp` target, specifying the new board name by passing the value to the `TARGET_GEN` variable. Optionally, specify the new device (`DEVICE_GEN`) and additional devices (`ADDITIONAL_DEVICES_GEN`). For example:
   ```
   make bsp TARGET_GEN=MyBSP DEVICE_GEN=CY8C624ABZI-D44
   ADDITIONAL_DEVICES_GEN=CYW4343WKUBG
   ```

   This will create a new BSP with the provided name at the top of the application project. It will also automatically copy the relevant startup and linker scripts based on the MPN specified by DEVICE_GEN into the newly created BSP.

   - If there were any issues with the new device's configuration, open the Device Configurator to address.
   - The BSP used as your starting point may have library references (for example, capsense.lib or udb-sdio-whd.lib) that are not needed by your custom BSP. These can be deleted from the BSP.
3. Define or update the alias for pins in the *cybsp_types.h* file.
4. Customize the *design.modus* file and other configuration files with new settings for clocks, power supplies, and peripherals as required.
5. Update the make `TARGET` variable to point to your new BSP. If you're starting from a Cypress provided example project, you will find this in the makefile file in the root of your project.

6. If using an IDE, regenerate the configuration settings to reflect the new BSP. Pick the appropriate command(s) for the IDE(s) that are being used. For example:

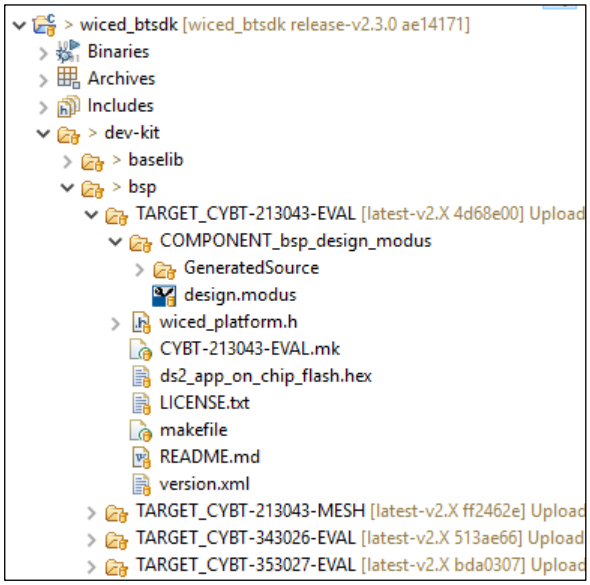
   ```
   make vscode
   ```

Note The full list of IDEs is dependent on the make build system being used. Use `make help` to see all supported IDE make targets. See also the Exporting to IDEs chapter in this document.

If you want to re-use a custom BSP on multiple applications, you can copy it into each application or you can put it into a git repo so that you can use it just like any other BSP during application creation. See the Manifest Files chapter for information on how to create a manifest to include your custom BSPs.

5.4 WICED Bluetooth BSPs (platforms)

All BSPs supported by BTSDK can be found in the *\wiced_btsdk\dev-kit\bsp* directory. The following shows a typical BTSDK BSP:



5.4.1 Selecting an alternative BSP

The application makefile has a default BSP; see `TARGET`. The makefile also has a list of other BSPs supported by the application; see `SUPPORTED_TARGETS`. To select an alternative BSP, set `TARGET` as one of the supported BSPs.

5.4.2 Custom BSP

5.4.2.1 Complete BSP

To create and use a complete custom BSP that you want to use in applications, perform the following steps:

1. Select an existing BSP you wish to use as a template from the list of supported BSPs in the \wiced_btsdk\dev-kit\bsp\ directory.

2. Make a copy in the same directory and rename it. For example \wiced_btsdk\dev-kit\bsp\TARGET_mybsp.

   Note This can be done in the system File Explorer and then refresh the workspace in Eclipse to see the new project. Delete the .git subdirectory from the newly copied directory before refreshing in Eclipse. If done in the IDE, an error dialog may appear complaining about items in the .git directory being out of sync. This can be resolved by deleting the .git subdirectory in the newly copied directory.

3. In the new \wiced_btsdk\dev-kit\bsp\TARGET_mybsp directory, rename the existing/original (BSP).mk file to mybsp.mk.
4. In the application makefile, set TARGET=mybsp and add it to SUPPORTED_TARGETS as well as TARGET_DEVICE_MAP. For example: mybsp/20819A1
5. Update design.modus for your custom BSP if needed using the Device Configurator link under Configurators in the Quick Panel.
6. Update the application makefile as needed for other custom BSP specific attributes and build the application.

5.4.2.2 Custom Pin Config Only

To create a custom pin configuration for applications using an existing BSP that supports the Device Configurator, perform the following steps:

1. Create a directory COMPONENT_(CUSTOM)_design_modus in the existing BSP directory. For example:

   \wiced_btsdk\dev-kit\bsp\TARGET_CYW920819EVB-02\COMPONENT_my_design_modus
2. Copy the file design.modus from the reference BSP COMPONENT_bsp_design_modus directory under \wiced_btsdk\dev-kit\bsp\ and place the file in the newly created COMPONENT_my_design_modus directory.
3. In the application makefile, add the following two lines, for example:

   ```
   DISABLE_COMPONENTS+=bsp_design_modus
   COMPONENTS+=my_design_modus
   ```
4. Building of the application will generate pin configuration source code under a GeneratedSource directory in the new COMPONENT_my_design_modus directory.

6 Manifest Files

6.1 Overview

Manifest files are XML files that provide lists of available boards, example code, or libraries. When you launch the Project Creator and Library Manager, these tools search appropriate servers for manifest files. There are several manifest files, including:

- The "super-manifest" file contains a list of Universal Record Indicators (URIs) that point to board, code example, and middleware manifest files.
- A “board-manifest” file contains a list of the BSPs that are available in the Project Creator and Library Manager tools.
- An "app-manifest" file contains a list of all code examples available for selected BSPs.
- A “middleware-manifest” file contains a list of the available middleware (libraries). Each middleware item can have one or more versions of that middleware available.

6.2 Create Your Own Manifest

By default, the ModusToolbox tools look for Cypress manifest files maintained on a Cypress server. So, the initial lists BSPs, code examples, and middleware available to use are limited to the Cypress manifest files. You can create your own manifest files on your servers or locally on your machine, and you can override where ModusToolbox tools look for manifest files.

To use your own examples, BSPs, and middleware, familiarize yourself with each of the manifest XML file structures. Then, create manifest files for your content and a super-manifest that points to your manifest files.

6.2.1 Overriding the Standard Super-Manifest

The location of the standard super-manifest file is hard coded into all of the tools. However, you may override this location by setting CyRemoteManifestOverride environment variable. When this variable is set, the tools use the value of this variable as the location of the super-manifest file and the hard-coded location is ignored. For example:

```
CyRemoteManifestOverride=https://myURL.com/mylocation/super-manifest.xml
```

6.2.2 Custom Super-Manifest

In addition to the standard super-manifest file, you can specify a custom super-manifest file. This allows you to add additional items (BSPs, code examples, libraries) along with the standard items. Do this by creating a file called *manifest.loc* in a hidden directory in your home directory named .modustoolbox:

~/.modustoolbox/manifest.loc

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
```

If this file exists, then each line in this file is treated as the URL to a super-manifest file. These are called the secondary or custom super-manifest files. The format of these files is exactly like the standard super-manifest file. Each of the custom super-manifest files are treated as separate super-manifest files. See the Conflicting Data section for dealing with conflicts.

6.2.3 Processing

The process for using the manifest files is the same for all tools that use the data.

The first step is to access the super-manifest file(s) to obtain a list of manifest files for each of the categories that the tool cares about. For example, the Library Manager tool cares about the board and middleware manifests.

The second step is to retrieve the manifest data from each manifest file and merge the data into a single global data model in the tool. See the Conflicting Data section for dealing with conflicts.

There is no per-file scoping. All data is merged before it is presented. The combination of a super manifest file and the merging of all of the data allows various contributors, including third party contributors, to make new data available without requiring coordinated releases between the various contributors.

The following table shows how manifests are processed:

Source	Syntax	Effect
CyRemoteManifestOverride	valid URL (e.g., file:/// ... or http:// ...)	Use that URL to fetch the super-manifest.
	Fragment (e.g., my/manifests/super-manifest.xml	Append the home directory to the front (e.g., file:///c/Users/benh/my/manifests/super-manifest.xml)
manifest.loc	valid URL (e.g., file:/// ... or http:// ...)	Use that URL to fetch the super-manifest.
	Fragment (e.g., my/manifests/super-manifest.xml	Append the directory in which *manifest.loc* resides (e.g., file:///c/Users/benh/.ModusToolbox/my/manifests/super-manifest.xml)
Manifest URIs	valid URI (e.g., file:/// ... or http:// ...)	Use that URI to fetch the manifest.
Manifest URIs from a local super-manifest file	fragment (e.g., my/manifests/manifest.xml)	Append the directory in which source super-manifest resides (e.g., file:///c/Users/benh/.modustoolbox/my/manifests/manifest.xml
Manifest URIs from a remote super-manifest file	fragment (e.g., my/manifests/manifest.xml)	Append the home directory to the front (e.g., file:///c/Users/benh/my/manifests/manifest.xml)

6.2.4 Conflicting Data

Ultimately, data from all of the super-manifest and manifest files are combined into a single data collection of BSPs, code examples, and middleware. During the collation of this data, there may be conflicting data entries. There are two types of conflicts.

The first kind is a conflict between data that comes from the primary super-manifest (and linked manifests) and data that comes from the custom super-manifest (and linked manifests). In this case, the data in the custom location overwrites the data from the standard location. This mechanism allows you to intentionally override data that is in the standard location. In this case, no error or warning is issued. It is a valid use case.

The second kind of conflict is between data coming from the same source (that is, both from primary or both from secondary). In this case, an error message is printed and all pieces of conflicting data are removed from the data model. This is done because in this case, it is not clear which data item is the correct one.

The following are how multiple super-manifests are handled:

- User manifests are processed first
- Multiple user manifests are treated as separate manifests
- Conflict within super-manifests; all items are discarded
- Conflicts between super-manifests; first super-manifest wins

6.3 Using Offline Content

In normal mode, ModusToolbox tools look for Cypress manifest files maintained on GitHub and downloads the firmware libraries from git repositories referenced by the manifests. If a network connection to online resources is not available, you can download a copy of all manifests and content, and then point the tools to use this copy in offline mode. This section describes how to download, install, and use the offline content.

1. Download modustoolbox-offline-content.zip from the Cypress website:

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

2. If you do not already have a hidden directory named *.modustoolbox* in your home directory, create one. Using Cygwin on Windows for example:

   ```
   mkdir -p "$USERPROFILE/.modustoolbox”
   ```

3. Extract the ZIP archive to the */.modustoolbox* sub-directory in your home directory. The resulting path should be:

   ~/.modustoolbox/offline

   The following is a Cygwin on Windows command-line example to use for extracting the content:

   ```
   unzip -qbod "$USERPROFILE/.modustoolbox" modustoolbox-offline-content.zip
   ```

   Note If you previously installed a copy of the offline content, you should delete the existing *~/.modustoolbox/offline* directory before extracting the archive. Using Cygwin on Windows for example:

   ```
   rm -rf "$USERPROFILE/.modustoolbox/offline"
   ```

4. To use the Project Creator GUI or Library Manager GUI in offline mode, select **Offline** from the **Settings** menu (refer to the appropriate user guide for details).

5. To use the Project Creator CLI in offline mode, execute the tool with the `--offline` argument. For example:

   ```
   project-creator-cli --board-id CY8CPROTO-062-4343W --app-id mtb-example-psoc6-hello-world --offline
   ```

6. The Project Creator and Library Manager tools execute the `make getlibs` command under the hood to download/update the firmware libraries. To execute the `make getlibs` target in offline mode, pass the `CY_GETLIBS_OFFLINE=true` argument:

   ```
   make getlibs CY_GETLIBS_OFFLINE=true
   ```

   To override the location of the offline content, set the `CY_GETLIBS_OFFLINE_PATH` variable:

   ```
   make getlibs CY_GETLIBS_OFFLINE=true CY_GETLIBS_OFFLINE_PATH="~/custom/offline/content"
   ```

   Refer to the ModusToolbox Build System chapter for more details about make targets and variables.

7. Once network connectivity is available, you can use the Library Manager tool to update the ModusToolbox project previously created offline to use the latest available content. Or you can execute the `make getlibs` command **without** the `CY_GETLIBS_OFFLINE` argument.

6.4 Access Private Repositories

You can extend the custom manifest with additional content from git repositories hosted on GitHub or any other online git server. To access private git repositories, you must configure the git client so that the ModusToolbox Project Creator and Library Manager tools can authenticate over HTTP/HTTPS protocols without an interactive password prompt.

To configure git credentials for non-interactive remote operations over HTTP protocols, refer to the git documentation:

- https://git-scm.com/book/en/v2/Git-Tools-Credential-Storage
- https://git-scm.com/docs/git-credential-store

The simplest way is to configure a git-credential-store and save the HTTP credentials is in a plain text file. Note that this option is less secure than other git credential helpers that use OS credentials storage.

The following steps show how to configure a git client to access GitHub private repositories without a password prompt:

1. Login to GitHub and go to Personal access tokens: https://github.com/settings/tokens
2. Click **Generate new token** to open the New personal access token screen.
3. On that screen:
   a. Type some text in the Note field.
   b. Under **Select scopes**, click on **repo**.
   c. Click **Generate token** (scroll down to see the button).
   d. Copy the generated token.
4. Open an interactive shell (for example, *modus-shell\Cygwin.bat* on Windows), and type the following commands (replace the user name and token with your information):

```
git config --global credential."https://github.com".helper store
GITHUB_USER=<your-github-username>
GITHUB_TOKEN=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx # generated at
https://github.com/settings/tokens
echo "https://$GITHUB_USER:$GITHUB_TOKEN@github.com" >> ~/.git-credentials
```

After entering the commands, you can clone private GitHub repositories without an interactive user/password prompt.

Note A GitHub account password can be used instead of `GITHUB_TOKEN`, in case the 2FA (two-factor authentication) is not enabled for the GitHub account. However, this option is not recommended.

6.5 Manifest XML File Structure

The following sections describe the XML file structure for each type of manifest file.

6.5.1 Super Manifest

6.5.1.1 Element and Attribute Descriptions

- super-manifest – The top-level XML element that encloses a list of manifests.
- super-manifest attribute: version – The version of this schema. Default is 1.0.
- app-manifest-list – The section that lists all sources for apps (i.e., code examples).
- app-manifest – Describes a single source for apps (see the section below for details about the app-manifest file).
- board-manifest-list – The section that lists all sources for board.
- board-manifest – Describes a single source for boards (see the section below for details about the board-manifest file).
- middleware-manifest-list – The section that lists all sources for middleware.
- middleware-manifest – Describes a single source for middleware. (See the section below for details about the middleware-manifest file).
- URI – Points to a manifest file of the correct type (based on the element that this attribute is in).

Note The app-manifest-list, app-manifest, board-manifest-list, board-manifest, middleware-manifest-list, middleware-manifest are all optional, so you don't need to specify all of them when you want to define, say, 1 or 2 custom BSPs.

6.5.1.2 Schema

```
<?xml version="1.0" encoding="utf-8"?>
<!--*************************************************************************
* File Name: super_schema.xsd
```

```
*
* Version: 1.0
*
* Description:
*  This file contains super manifest schema.
*
********************************************************************************
* Copyright 2018-2020, Cypress Semiconductor Corporation.  All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
********************************************************************************-->
<xs:schema attributeFormDefault="qualified"
           elementFormDefault="qualified"
           xml:lang="EN"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0">
  <!-- URI constraints
       - starts with http(s)|file|ftp://
       - w/o whitespaces
  -->
  <xs:simpleType name="validURI">
    <xs:restriction base="xs:anyURI">
      <xs:pattern value="(https?|f(ile|tp)://\S+"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="super-manifest">
    <xs:complexType>
      <xs:all>
        <xs:element name="app-manifest-list" maxOccurs="1" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="app-manifest" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="uri" type="validURI" maxOccurs="1" minOccurs="0"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="board-manifest-list" maxOccurs="1" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="board-manifest" maxOccurs="unbounded" minOccurs="1">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="uri" type="validURI" maxOccurs="1" minOccurs="0"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="middleware-manifest-list" maxOccurs="1" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="middleware-manifest" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="uri" type="validURI" maxOccurs="1" minOccurs="0"/>
                  </xs:sequence>
```

```
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:all>
      <xs:attribute name="version" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

6.5.1.3 Example

```
<super-manifest version="1.0">
  <app-manifest-list>
    <app-manifest>
      <uri>http://www.cypress.com/asset_psoc/app_manifest.xml</uri>
    </app-manifest>
    <app-manifest>
      <uri>http://www.cypress.com/asset_bt/app_manifest.xml</uri>
    </app-manifest>
  </app-manifest-list>
  <board-manifest-list>
    <board-manifest>
      <uri>http://www.cypress.com/asset_psoc/board_manifest.xml</uri>
    </board-manifest>
    <board-manifest>
      <uri>http://www.cypress.com/asset_bt/board_manifest.xml</uri>
    </board-manifest>
  </board-manifest-list>
  <middleware-manifest-list>
    <middleware-manifest>
      <uri>http://www.cypress.com/asset_psoc/middleware_manifest.xml</uri>
    </middleware-manifest>
    <middleware-manifest>
      <uri>http://www.cypress.com/asset_bt/middleware_manifest.xml</uri>
    </middleware-manifest>
  </middleware-manifest-list>
</super-manifest>
```

6.5.2 Board Manifest

6.5.2.1 Element and Attribute Descriptions

Name	Parent	Need	Description
boards	-	required	The top-level XML element that encloses a list of boards. The list can be empty.
boards:version	-	optional	The version attribute specifies the schema version. Default is "1.0".
board	boards	optional	A description of a single board.
id	board	required	A unique identifier that identifies a board. The manifest processing code will give an error if multiple boards have the same id.
name	board	required	A user-friendly name for the board. This is what is displayed in the UIs.
uri	board	required	The URI for the git repository holding the board. Must start with "http(s)://", "file:///", "ftp://". Must have no whitespaces
versions	board	required	Used to group version components. Can have one or more version element
version	versions	required	An element that defines a specific version for a board. All version of a board must come from the same git repo but they will each have a different commit string and num string (number, name, tag)
num	version	required	Used to store readable format of board version

Name	Parent	Need	Description
commit	version	required	Used to store board version commit
chips	board	required	The chips (MCU and radio) that form the core of the board's functionality
mcu	chips	required	A MCU chip part number
radio	chips	optional	A radio chip part number
category	board	optional	A user-friendly text string that specifies the category for displaying this board item in a GUI. It is expected that all board in the same category will be shown together in the library management GUI.
description	board	optional	An html description of the board. This is meant to be displayed in the UIs.
prov_capabilities	board	optional	A list of capabilities that this board provides. The list is whitespace delimited and each item in the list must be a valid C identifier. If this element is missing or empty, that means that this board provides no capabilities. Only apps or middleware whose capability is empty can work with this kind of board.
documentation_url	board	optional	The URI for the board documentation. Must start with "http(s)://", "file:///", "ftp://". Must have no whitespaces
summary	board	optional	A text description of the board. This is meant to show up in UIs.

6.5.2.2 Schema

```
<?xml version="1.0" encoding="utf-8"?>
<!--*****************************************************************************
* File Name: board_schema.xsd
*
* Version: 1.0
*
* Description:
*  This file contains board manifest schema.
*
******************************************************************************
* Copyright 2018-2020, Cypress Semiconductor Corporation.  All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
*****************************************************************************-->
<xs:schema attributeFormDefault="qualified"
           elementFormDefault="qualified"
           xml:lang="EN"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0">
  <!-- URI constraints
       - starts with http(s)|file|ftp://
       - w/o whitespaces
  -->
  <xs:simpleType name="validURI">
    <xs:restriction base="xs:string">
      <xs:pattern value="(https?|f(ile|tp)://\S+"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- commit constraints
       - no
  -->
  <xs:simpleType name="validCommit">
    <xs:restriction base="xs:string">
    </xs:restriction>
  </xs:simpleType>
  <!-- capabilities constraints
       - whitespace-separated list
  -->
  <xs:simpleType name="validCapabilities">
    <xs:restriction base="xs:string">
```

```
      <xs:pattern value="([\w_]+(\s*[\w_]+)*)?"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- ID constraints
       - whitespace-separated list
  -->
  <xs:simpleType name="validID">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:pattern value="\s*\S+\s*"/>
      <xs:whiteSpace value="collapse"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- NAME constraints
       - no leading/trailing whitespaces
  -->
  <xs:simpleType name="validName">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:pattern value="\S+(\s+\S+)*"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="boards">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="board" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all>
              <xs:element type="validID" name="id" maxOccurs="1" minOccurs="1"/>
              <xs:element type="validURI" name="board_uri" maxOccurs="1" minOccurs="1"/>
              <xs:element name="category" type="xs:string" maxOccurs="1" minOccurs="0"/>
              <xs:element type="validCommit" name="commit" maxOccurs="1" minOccurs="0"/>
              <xs:element name="versions" maxOccurs="1" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="version" maxOccurs="unbounded" minOccurs="1">
                      <xs:complexType>
                        <xs:all>
                          <xs:element name="num" type="xs:string" maxOccurs="1"
minOccurs="1" />
                          <xs:element name="commit" type="validCommit" maxOccurs="1"
minOccurs="1"/>
                        </xs:all>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="chips" maxOccurs="1" minOccurs="1">
                <xs:complexType>
                  <xs:all>
                    <xs:element type="xs:string" name="mcu" maxOccurs="1" minOccurs="1"/>
                    <xs:element type="xs:string" name="radio" maxOccurs="1" minOccurs="0"/>
                  </xs:all>
                </xs:complexType>
              </xs:element>
              <xs:element type="validName" name="name" maxOccurs="1" minOccurs="1"/>
              <xs:element type="xs:string" name="summary" maxOccurs="1" minOccurs="0"/>
              <xs:element type="validCapabilities" name="prov_capabilities" maxOccurs="1"
minOccurs="0"/>
              <xs:element type="xs:string" name="description" maxOccurs="1" minOccurs="0"/>
              <xs:element type="validURI" name="documentation_url" maxOccurs="1"
minOccurs="0"/>
```

```
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="version" type="xs:string"/>
    </xs:complexType>
    <xs:key name="id">
      <xs:selector xpath="./*"/>
      <xs:field xpath="id"/>
    </xs:key>
  </xs:element>
</xs:schema>
```

6.5.2.3 Example

```
<?xml version="1.0" encoding="UTF-8"?>
<boards version="1.0">
  <board>
    <id>CY8CPROTO-062-4343W</id>
    <board_uri>http://git-ore.aus.cypress.com/asset-bt/board/cy8cproto-062-
4343w.git</board_uri>

    <chips>
      <mcu>CY8C6247BZI-D54</mcu>
      <radio>CYW4343WKUBG</radio>
    </chips>

    <name>CY8CPROTO-062-4343W</name>
    <summary>The CY8CPROTO-062-4343W PSoC 6 Wi-Fi BT Prototyping Kit is a low-cost hardware
platform that enables design and debug of PSoC 6 MCUs. It comes with a Murata LBEE5KL1DX
module, based on the CYW4343W combo device, industry-leading CapSense for touch buttons and
slider, on-board debugger/programmer with KitProg3, microSD card interface, 512-Mb Quad-SPI
NOR flash, PDM-PCM microphone, and a thermistor. This kit is designed with a snap-away
form-factor, allowing the user to separate the different components and features that come
with this kit and use independently. In addition, support for Digilent's Pmod interface is
also provided with this kit.</summary>
    <prov_capabilities>led switch wifi bt usb_device sdhc qspi capsense</prov_capabilities>
    <description>
      <![CDATA[
      <div class="category">Kit Features:</div>
      <ul>
      <li>Support of up to 2MB Flash and 1MB SRAM</li>
      <li>Dedicated SDHC to interface with WICED wireless devices.</li>
      <li>Delivers dual-cores, with a 150-MHz Arm Cortex-M4 as the primary application
processor and a 100-MHz Arm Cortex-M0+ as the secondary processor for low-power
operations.</li>
      <li>Supports Full-Speed USB, capacitive-sensing with CapSense, a PDM-PCM digital
microphone interface, a Quad-SPI interface, 13 serial communication blocks, 7 programmable
analog blocks, and 56 programmable digital blocks.</li>
      </ul>

      <div class="category">Kit Contents:</div>
      <ul>
      <li>PSoC 6 Wi-Fi BT Prototyping Board</li>
      <li>USB Type-A to Micro-B cable</li>
      <li>Quick Start Guide</li>
      </ul>"
    ]]>
    </description>
    <documentation_url>http://www.cypress.com/CY8CPROTO-062-4343W</documentation_url>
    <versions>
      <version>
```

```
            <num>Latest 1.x</num>
            <commit>latest_1.x</commit>
        </version>
        <version>
            <num>Release 1.1</num>
            <commit>release_1.1</commit>
        </version>
        <version>
            <num>Release 1.0</num>
            <commit>release_1.0</commit>
        </version>
    </versions>
  </board>
</boards>
```

6.5.3 App Manifest

6.5.3.1 Element and Attribute Descriptions

Name	Parent	Need	Description
apps	-	required	The top-level XML element that encloses a list of app elements. The list can be empty.
apps:version	-	optional	The version attribute specifies the schema version. Default is "1.0".
app	apps	optional	A description of a single app (i.e., code example).
id	app	required	A unique identifier that identifies an app. The manifest processing code will give an error if multiple apps have the same id.
name	app	required	A user-friendly name for the app. This is what is displayed in the UI.
uri	app	required	The URI for the git repository holding the app. Must start with "http(s)://", "file:///", "ftp://". Must have no whitespaces
versions	app	required	Used to group version components. Can have one or more version of element
version	versions	required	An element that defines a specific version for a app. All version of a app must come from the same git repo but they will each have a different commit string and num string (number, name, tag).
num	version	required	Used to store readable format of version.
commit	version	required	Used to store board version commit.
desc	app	optional	A user-friendly text description of the app. This is meant to be displayed in the UI.
req_capabilities	app	optional	A list of capabilities that this application requires. This list is treated as an "and" list. That is, all capabilities need to be met. The list is whitespace delimited and each item in the list must be a valid C identifier. If this element is missing or empty, it means that this application has no capability requirements. That is, it works with all boards.

6.5.3.2 Schema

```
<?xml version="1.0" encoding="utf-8" ?>
<!--*****************************************************************************
* File Name: app_schema.xsd
*
* Version: 1.0
*
* Description:
*  This file contains application manifest schema.
*
******************************************************************************
* Copyright 2018-2020, Cypress Semiconductor Corporation.  All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
```

```
*****************************************************************************-->
<xs:schema attributeFormDefault="qualified"
           elementFormDefault="qualified"
           xml:lang="EN"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0">
  <!-- URI constraints
       - starts with http(s)|file|ftp://
       - w/o whitespaces
  -->
  <xs:simpleType name="validURI">
    <xs:restriction base="xs:string">
      <xs:pattern value="(https?|f(ile|tp)://\S+"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- commit constraints
       - no
  -->
  <xs:simpleType name="validCommit">
    <xs:restriction base="xs:string">
    </xs:restriction>
  </xs:simpleType>
  <!-- capabilities constraints
       - whitespace-separated list
  -->
  <xs:simpleType name="validCapabilities">
    <xs:restriction base="xs:string">
      <xs:pattern value="([\w_]+(\s*[\w_]+)*)?"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- ID constraints
       - whitespace-separated list
  -->
  <xs:simpleType name="validID">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:pattern value="\s*\S+\s*"/>
      <xs:whiteSpace value="collapse"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- NAME constraints
       - no leading/trailing whitespaces
  -->
  <xs:simpleType name="validName">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:pattern value="\S+(\s+\S+)*"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="apps">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="app" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all>
              <xs:element type="validName" name="name" maxOccurs="1" minOccurs="1"/>
              <xs:element type="validID" name="id" maxOccurs="1" minOccurs="1"/>
              <xs:element type="validURI" name="uri" maxOccurs="1" minOccurs="1"/>
              <xs:element type="validCommit" name="commit" maxOccurs="1" minOccurs="0"/>
              <xs:element name="versions" maxOccurs="1" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="version" maxOccurs="unbounded" minOccurs="1">
```

```
                        <xs:complexType>
                          <xs:all>
                           <xs:element name="num" type="xs:string" maxOccurs="1"
minOccurs="1" />
                           <xs:element name="commit" type="validCommit" maxOccurs="1"
minOccurs="1"/>
                          </xs:all>
                        </xs:complexType>
                      </xs:element>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
                <xs:element type="xs:string" name="description" maxOccurs="1" minOccurs="0"/>
                <xs:element type="validCapabilities" name="req_capabilities" maxOccurs="1"
minOccurs="0"/>
              </xs:all>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="version" type="xs:string"/>
      </xs:complexType>
      <xs:key name="id">
        <xs:selector xpath="./*"/>
        <xs:field xpath="id"/>
      </xs:key>
    </xs:element>
  </xs:schema>
```

6.5.3.3 Example

```
<apps version="1.0">
    <app>
           <name>Blinky LED</name>
        <id>BlinkyLED</id>
        <uri>http://git-ore.aus.cypress.com/asset-psoc/apps/blinky-led.git</uri>
        <commit>abcdef</commit>
        <description> ... </description>
        <req_capabilities>led</req_capabilities>
    </app>
    <app>
           <name>Capsense Slider</name>
           <id>CapsenseSlider</id>
        <uri>http://git-ore.aus.cypress.com/asset-psoc/apps/capsense-slider.git</uri>
        <versions>
                <version>
                <num>Latest 1.x</num>
                <commit>latest_1.x</commit>
                </version>
          </versions>
          <description> ... </description>
        <req_capabilities>capsense led</req_capabilities
    </app>
    ...
</apps>
<apps version="1.0">
    <app>
           <name>BLE Beacon</name>
        <id>BLE_Beacon</id>
        <uri>http://git-ore.aus.cypress.com/asset-bt/apps/ble-beacon.git</uri>
        <commit>abcdef</commit>
        <description> ... </description>
        <req_capabilities>ble</req_capabilities>
```

```
    </app>
    <app>
          <name>BLE MeshDimmer</name>
        <id>BLE_MeshDimmer</id>
        <uri>http://git-ore.aus.cypress.com/asset-bt/apps/ble-mesh-dimmer.git</uri>
        <versions>
                <version>
                <num>Latest 1.x</num>
                <commit>latest_1.x</commit>
                </version>
          </versions>
        <description> ... </description>
        <req_capabilities>ble mesh</req_capabilities>
    <app>
    ...
</apps>
```

6.5.4 Middleware Manifest

6.5.4.1 Element and Attribute Descriptions

Name	Parent	Need	Description
middleware	-	required	The top-level XML element that encloses a list of middleware elements. The list can be empty.
middleware:version	-	optional	The version attribute specifies the schema version. Default is "1.0".
middleware	middleware	optional	A description of a single middleware item.
id	middleware	required	A unique identifier that identifies the middleware. The manifest processing code will give an error if multiple middleware items have the same id.
name	middleware	required	A user-friendly name for the middleware. This is what is displayed in the UI.
uri	middleware	required	The URI for the git repository holding the middleware.
desc	middleware	optional	A user-friendly text description of the middleware item. This is meant to be displayed in the UI.
category	middleware	optional	A user-friendly text string that specifies the category for displaying this middleware item in a GUI. It is expected that all middleware in the same category will be shown together in the library management GUI.
mutex_group	middleware	optional	Defines that this middleware participates in a "mutual exclusive" group with the specified name (user-friendly text). The GUI will ensure that for all middleware items in the same mutex_group, only one can be selected at a time. Participation in the same mutex_group can be spread across multiple middleware-manifest files.
versions	middleware	optional	Used to group version components. Can have from one to unbound number of version element
version	versions	required	An element that defines a specific version for a middleware item. All versions of a middleware item must come from the same git repo but they will each have a different commit string.
num	version	required	The version number for a middleware item.
commit	version	required	The git commit identifier for a particular version of a middleware item.
desc	version	optional	A user-friendly text description relevant to this specific version of the middleware item. The GUI will show the general middleware item description AND the specific version information for the selected version.
req_capabilities	middleware	optional	A list of capabilities that this middleware requires. This list is treated as an "and" list. That is, all capabilities need to be met. The list is whitespace delimited and each item in the list must be a valid C identifier. If this element is missing or empty, it means that this middleware has no capability requirements. That is, it works with all boards.

6.5.4.2 Schema

```
<?xml version="1.0" encoding="utf-8"?>
<!--*****************************************************************************
* File Name: middleware_schema.xsd
*
* Version: 1.0
*
* Description:
*  This file contains middleware manifest schema.
*
******************************************************************************
* Copyright 2018-2020, Cypress Semiconductor Corporation.  All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
*****************************************************************************-->
<xs:schema attributeFormDefault="qualified"
           elementFormDefault="qualified"
           xml:lang="EN"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0">
  <!-- URI constraints
       - starts with http(s)|file|ftp://
       - w/o whitespaces
  -->
  <xs:simpleType name="validURI">
    <xs:restriction base="xs:string">
      <xs:pattern value="(https?|f(ile|tp)://\S+"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- commit constraints
       - no
  -->
  <xs:simpleType name="validCommit">
    <xs:restriction base="xs:string">
    </xs:restriction>
  </xs:simpleType>
  <!-- capabilities constraints
       - whitespace-separated list
  -->
  <xs:simpleType name="validCapabilities">
    <xs:restriction base="xs:string">
      <xs:pattern value="([\w_]+(\s*[\w_]+)*)?"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- ID constraints
       - whitespace-separated list
  -->
  <xs:simpleType name="validID">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:pattern value="\s*\S+\s*"/>
      <xs:whiteSpace value="collapse"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="middleware">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="middleware" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:all>
              <xs:element name="name" type="xs:string" maxOccurs="1" minOccurs="1"/>
              <xs:element name="id" type="validID" maxOccurs="1" minOccurs="1"/>
```

```
              <xs:element name="uri" type="validURI" maxOccurs="1" minOccurs="1"/>
              <xs:element name="desc" type="xs:string" maxOccurs="1" minOccurs="0"/>
              <xs:element name="category" type="xs:string" maxOccurs="1" minOccurs="0"/>
              <xs:element name="mutex_group" type="xs:string" maxOccurs="1" minOccurs="0"/>
              <xs:element name="req_capabilities" type="validCapabilities" maxOccurs="1"
minOccurs="0"/>
              <xs:element name="versions" maxOccurs="1" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="version" maxOccurs="unbounded" minOccurs="1">
                      <xs:complexType>
                        <xs:all>
                         <xs:element name="num" type="xs:string" maxOccurs="1"
minOccurs="1" />
                         <xs:element name="commit" type="validCommit" maxOccurs="1"
minOccurs="0"/>
                         <xs:element name="desc" type="xs:string" maxOccurs="1"
minOccurs="0"/>
                        </xs:all>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
     <xs:attribute name="version" type="xs:string"/>
    </xs:complexType>
    <xs:key name="id">
      <xs:selector xpath="./*"/>
      <xs:field xpath="id"/>
    </xs:key>
  </xs:element>
</xs:schema>
```

6.5.4.3 *Example*

```
<middleware version="1.0">
  <middleware>
    <name>USB Device</name>
    <id>USBDevice</id>
    <uri>http://git-ore.aus.cypress.com/asset-psoc/middleware/usbdev.git</uri>
    <desc></desc>
    <category>misc</category>
    <req_capabilities>led usb</req_capabilities>
    <versions>
      <version>
        <num>1.0.0.0</num>
        <commit>abcdef</commit>
        <desc></desc>
      </version>
      <version>
        <num>1.2.0.0</num>
        <commit>123456</commit>
        <desc></desc>
      </version>
  </versions>
  </middleware>
  <middleware>
    <name>Retarget IO</name>
```

```
      <id>RetargetIO</id>
      <uri>http://git-ore.aus.cypress.com/asset-psoc/middleware/retarget-io.git</uri>
      <desc></desc>
      <category>misc</category>
      <req_capabilities></req_capabilities>
      <versions>
        <version>
          <num>1.0.0.0</num>
          <commit>abcdef</commit>
          <desc></desc>
        </version>
     </versions>
    </middleware>
    <middleware>
      <name>CapSense Hard-FP</name>
      <id>Capsense_hfp</id>
      <uri>http://git-ore.aus.cypress.com/asset-psoc/middleware/capsense-hard.git</uri>
      <desc></desc>
      <category>misc</category>
      <mutex_group>CapSense</mutex_group>
      <req_capabilities>capsense</req_capabilities>
      <versions>
        <version>
          <num>1.0.0.0</num>
          <commit>abcdef</commit>
          <desc></desc>
        </version>
      </versions>
    </middleware>
    <middleware>
      <name>CapSense Soft-FP</name>
      <id>Capsense_sfp</id>
      <uri>http://git-ore.aus.cypress.com/asset-psoc/middleware/capsense-soft.git</uri>
      <desc></desc>
      <category>misc</category>
      <mutex_group>CapSense</mutex_group>
      <req_capabilities>capsense</req_capabilities>
      <versions>
       <version>
          <num>1.0.0.0</num>
          <commit>abcdef</commit>
          <desc></desc>
        </version>
      </versions>
  </middleware>

</middleware>
```

7 Exporting to IDEs

7.1 Overview

This chapter describes how to export a ModusToolbox application to various supported IDEs in addition to the provided Eclipse IDE. As noted in the ModusToolbox Build System chapter, the make command includes various targets for the following IDEs:

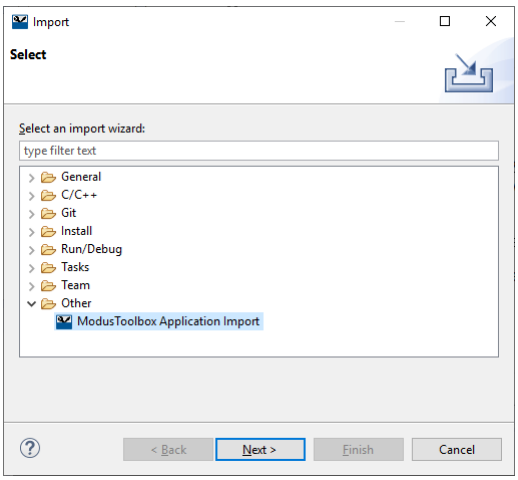
- Visual Studio (VS) Code – `make vscode`
- IAR Embedded Workbench – `make ewarm8`
- Keil µVision – `make uvision5`

7.2 Import to Eclipse

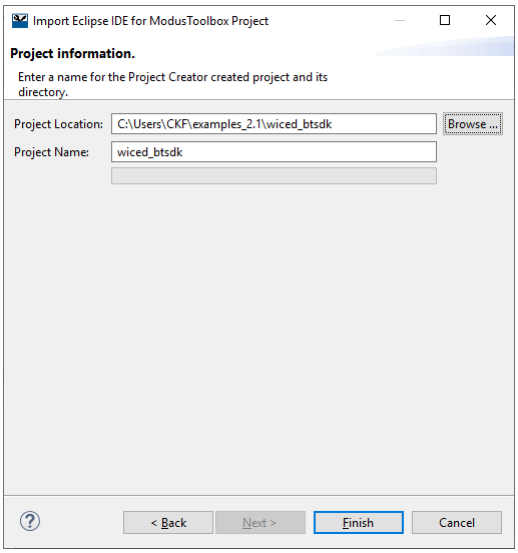
The easiest way to create a ModusToolbox application for Eclipse is to use the Eclipse IDE included with the ModusToolbox software. ModusToolbox includes an Eclipse plugin that provides links to launch the Project Creator tool and then import the application into Eclipse. For details, refer to the Eclipse IDE for ModusToolbox Quick Start Guide or User Guide.

If you already have a ModusToolbox application that was not created using the Eclipse IDE flow, you can import it for use in Eclipse as follows:

1. Open the Eclipse IDE included with ModusToolbox, and select **File > Import… > Other > ModusToolbox Application Import**.



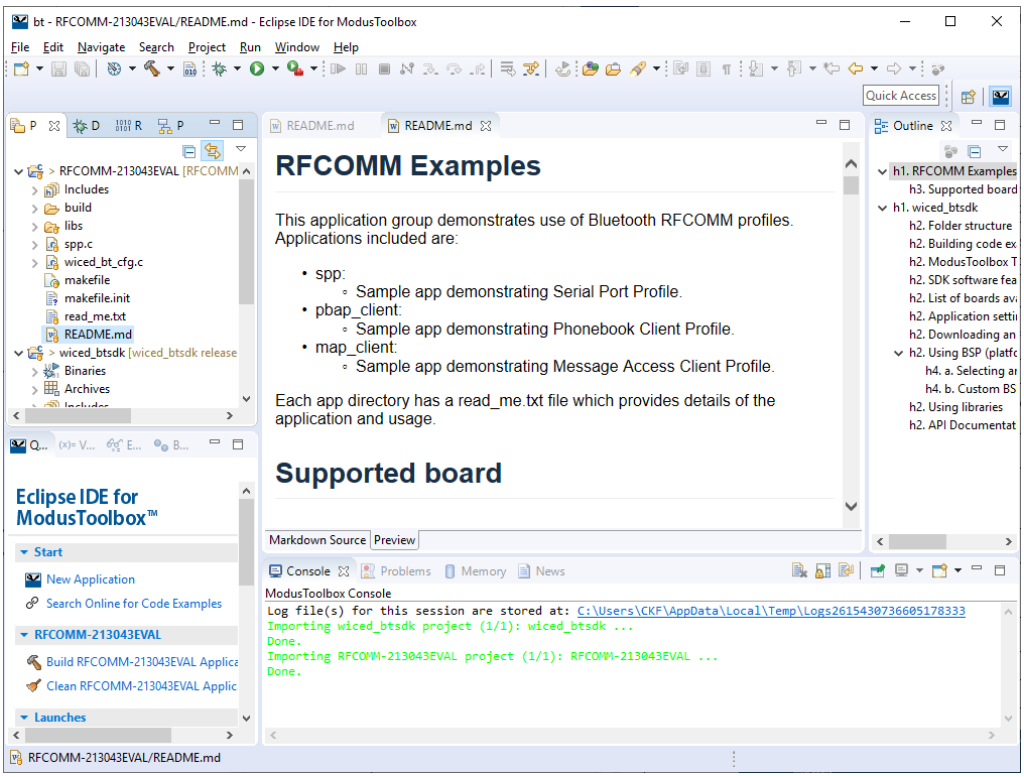
2. Click **Next >**. In the **Project Location** field, click the **Browse…** button and navigate to the application's directory.



3. Click **Finish**.

   The application displays in the Eclipse IDE Project Explorer.

   Note For a WICED Bluetooth application, you must repeat the import process for the separate applications, such as RFCOMM-213043EVAL, Audio-20819EVB02, etc.

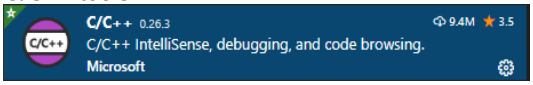


7.3 Export to VS Code

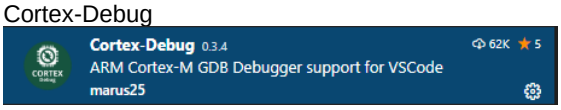
This section describes how to export a ModusToolbox application to VS Code.

7.3.1 Prerequisites

- ModusToolbox 2.1 software and application
- VS Code version 1.42.x or later
- VS Code extensions. Install the following:
  - C/C++ tools

  - Cortex-Debug

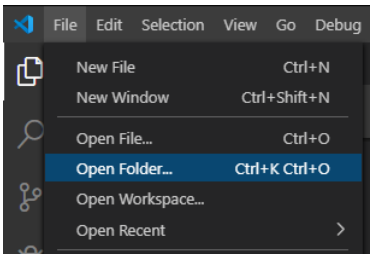
    

7.3.2 Process for PSoC 6 Application

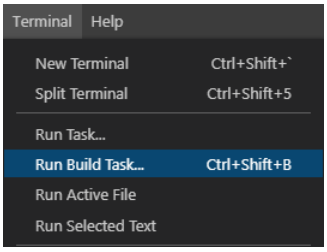
1. Create a ModusToolbox application.
2. Open an appropriate shell program (see CLI Set-up Instructions), and navigate to the application directory, and run the following command:

   ```
   make vscode
   ```

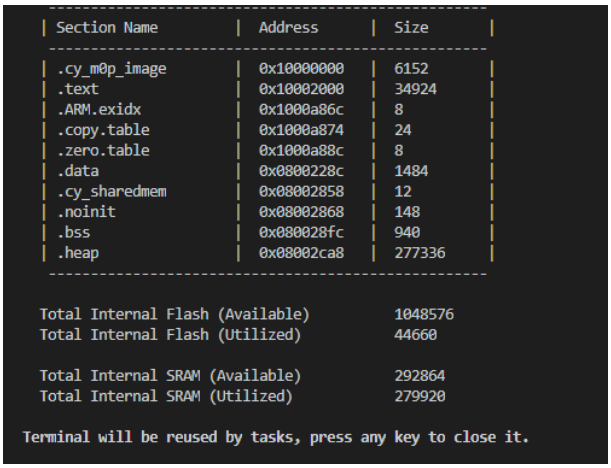
   This command generates json files for debug/program launches, IntelliSense, and custom tasks.
3. Open the VS Code tool, and select **File > Open Directory…**

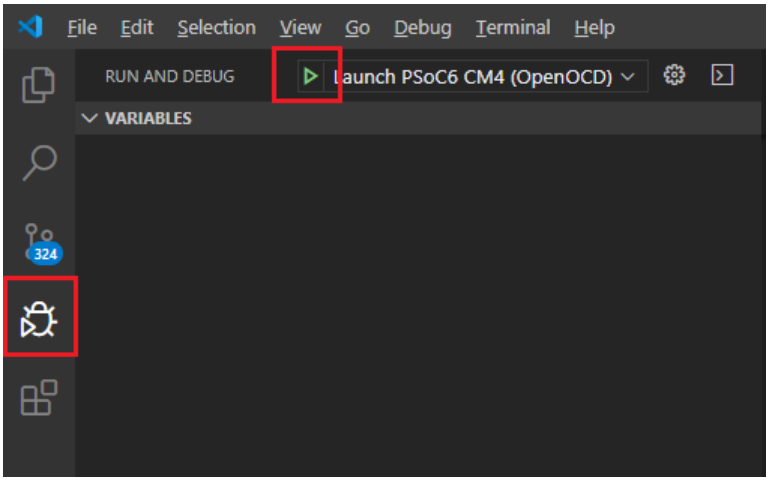
   Note On macOS, this command is **File > Open…**
4. Navigate to and select the application directory, and then click **Select Directory**.
5. Select **Terminal > Run Build Task…**

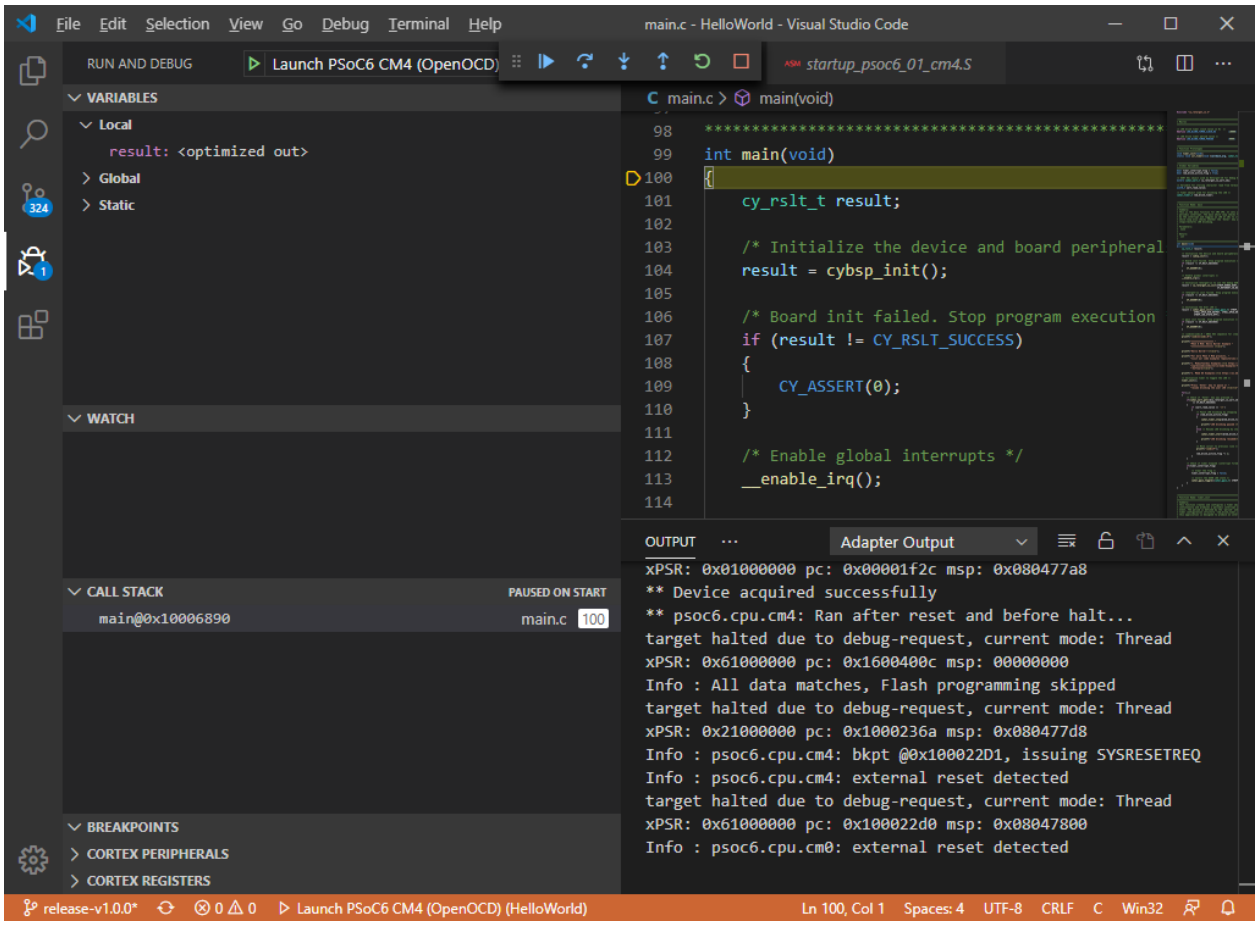
6. Then, select **Build:Build Debug**. After building, the VS Code terminal should display messages similar to the following:



7. Click the “bug” icon on the left and then click the **Play** button.



The VS Code tool runs in debug mode.



7.4 Export IAR EWARM (Windows Only)

This section describes how to export a ModusToolbox application to IAR Embedded Workbench and debug it with CMSIS-DAP or J-Link.

7.4.1 Prerequisites

- ModusToolbox 2.1 software and application
- Python 3.7: Use the Windows x86-64 executable installer available at python.org:

  https://www.python.org/ftp/python/3.7.6/python-3.7.6-amd64.exe

  Note Add *python.exe* to your PATH during the installation.
- IAR Embedded Workbench version 8.42.2 or later
- PSoC 6 Kit (for example, CY8CPROTO-062-4343W) with KitProg3 FW
- For J-Link debugging, download and install J-Link software:

  https://www.segger.com/downloads/jlink/JLink_Windows.exe

7.4.2 Process for PSoC 6 Application

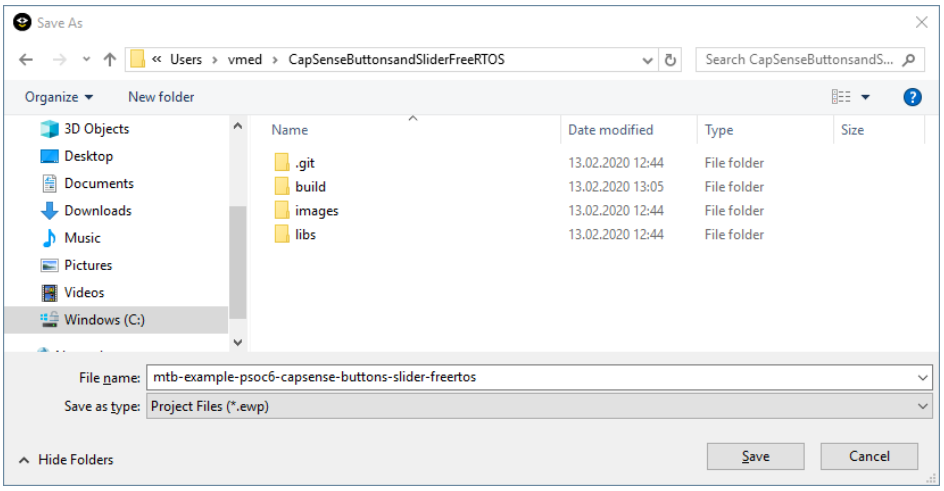
1. Create a ModusToolbox application.
2. Open an appropriate shell program (see CLI Set-up Instructions), and navigate to the application directory.
3. Run the following command:

   ```
   make ewarm8 TOOLCHAIN=IAR
   ```

   Note Alternately, you can edit the application's makefile to specify the IAR toolchain.

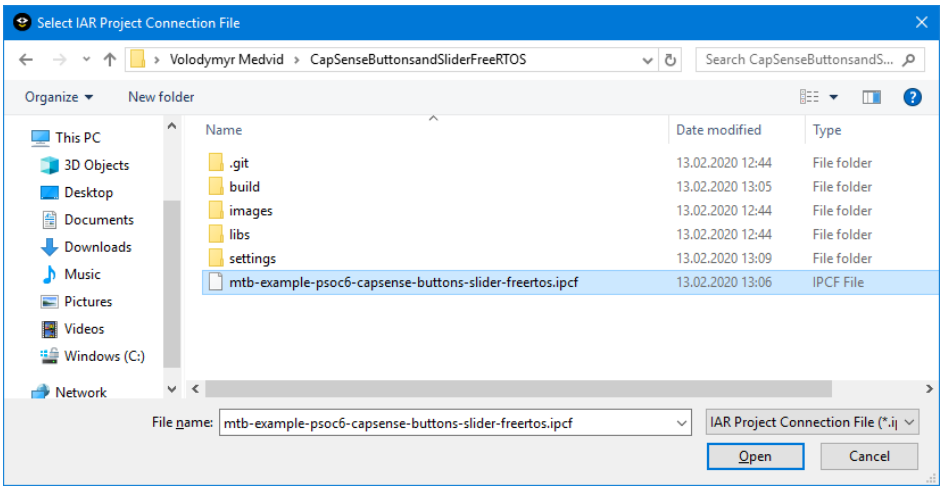
   An IAR connection file appears in the application directory. For example:

   mtb-example-psoc6-capsense-buttons-slider-freertos.ipcf
4. Start IAR Embedded Workbench.
5. On the main menu, select **Project > Create New Project > Empty project** and click **OK**.
6. Browse to the ModusToolbox application directory, enter an arbitrary application name, and click **Save**.



7. After the application is created, select **File > Save Workspace**, enter an arbitrary workspace name and click **Save**.
8. Select **Project > Add Project Connection** and click **OK**.

9. On the Select IAR Project Connection File dialog, select the *.ipcf* file and click **Open**:



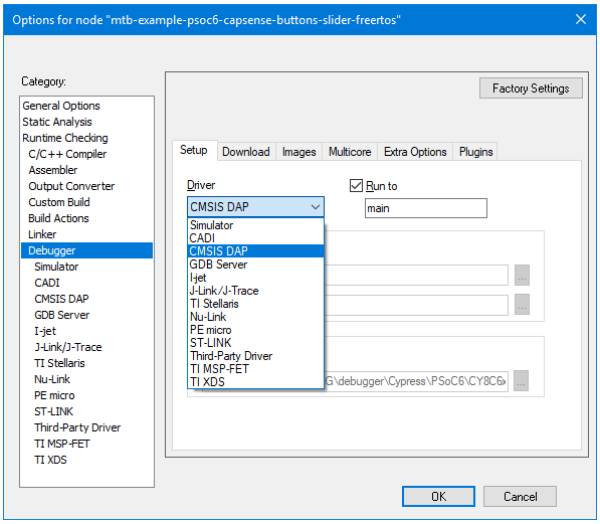
10. On the main menu, Select **Project > Make**.

11. Connect the PSoC 6 kit to the host PC.

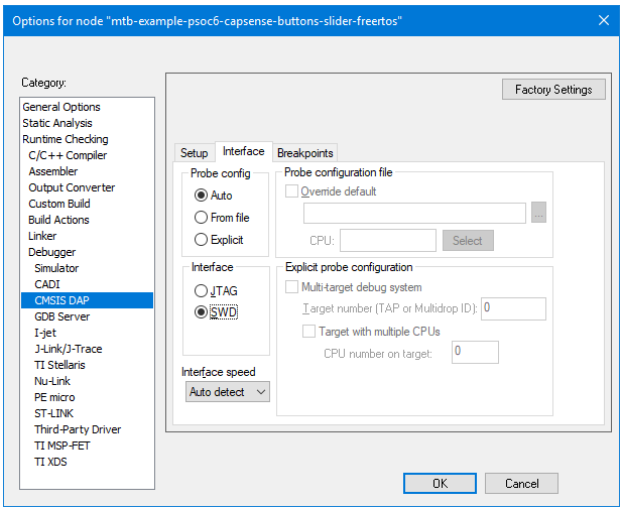
12. As needed, run the fw-loader tool to make sure the board firmware is upgraded to KitProg3. See KitProg3 User Guide for details. The tool is in the following directory by default:

    <user_home>/ModusToolbox/tools_2.1/fw-loader/bin/

13. Select **Project > Options > Debugger** and select **CMSIS-DAP** in the Driver list:



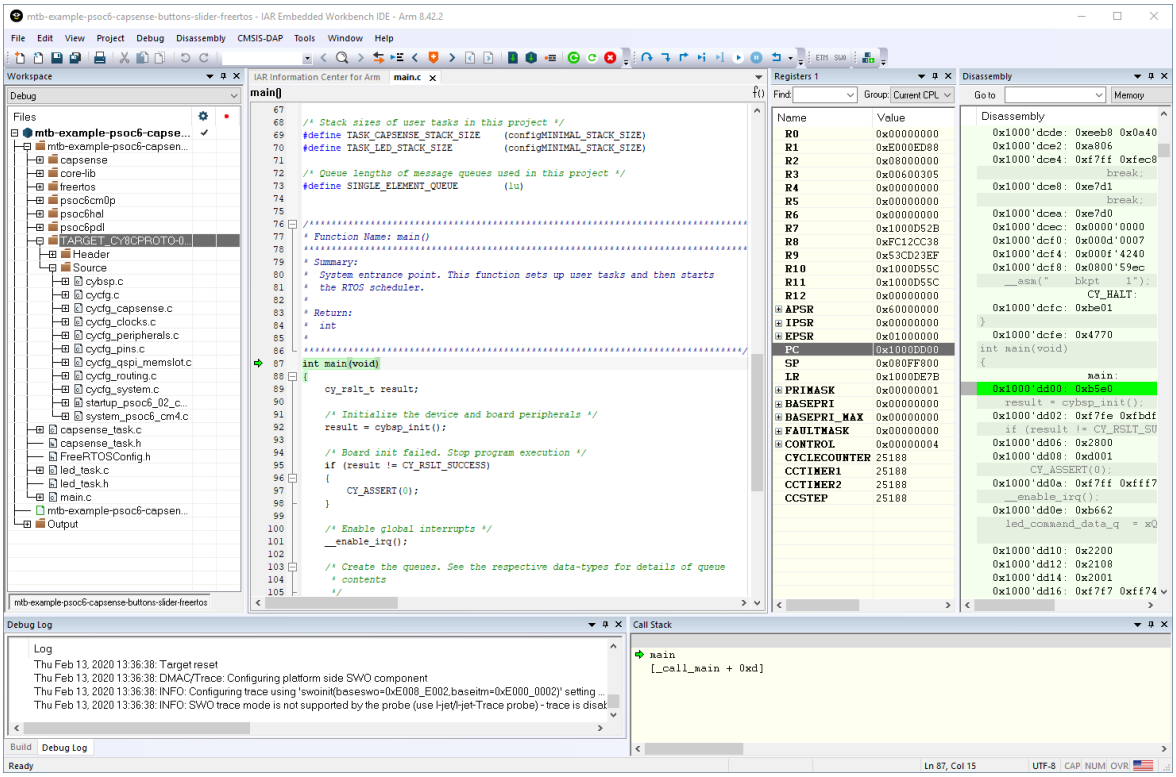
14. Select the **CMSIS-DAP** node and switch the interface from **JTAG** to **SWD**:



15. Click **OK**.

16. Select **Project > Download and Debug**.

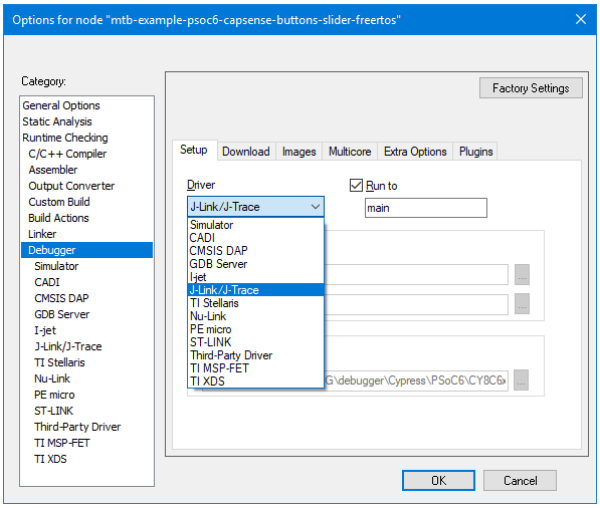
    The IAR Embededed Workbench starts a debugging session and jumps to the main function.



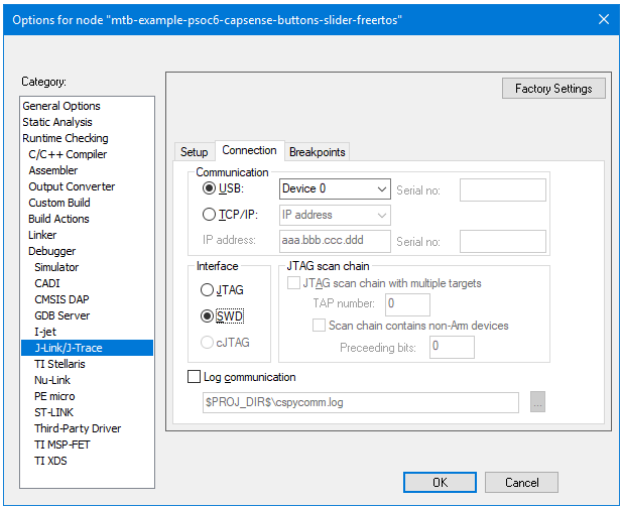
7.4.2.1 To Use J-Link

You can use a J-Link debugger probe to debug the application.

1. Open the Options dialog and select the **Debugger** item under **Category**.
2. Then select **J-Link/J-Trace** as the active driver:



3. Select the **J-Link/J-Trace** item under **Category**, and under the **Connection** tab, switch the interface to **SWD**:



4. Connect a J-Link debug probe to the 10-pin adapter (needs to be soldered on the prototyping kits), and start the debugging session.

7.5 Export to Keil µVision 5 (Windows Only)

This section describes how to export ModusToolbox application to Keil µVision and debug it with CMSIS-DAP or J-Link.

7.5.1 Prerequisites

- ModusToolbox 2.1 software and application
- Python 3.7: Use the Windows x86-64 executable installer available at python.org:

  https://www.python.org/downloads/

  Note Add *python.exe* to your PATH during the installation.
- Keil µVision version 5.28 or later
- PSoC 6 Kit (for example, CY8CPROTO-062-4343W) with KitProg3 Firmware
- For J-Link debugging, download and install J-Link software:

  https://www.segger.com/downloads/jlink/JLink_Windows.exe

7.5.2 Process for PSoC 6 Application

1. Create a ModusToolbox application.
2. Open an appropriate shell program (see CLI Set-up Instructions), and navigate to the application directory.
3. Run the following command:

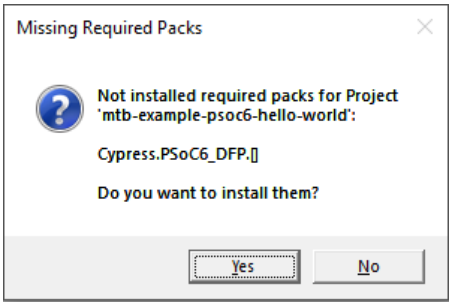
   ```
   make uvision5 TOOLCHAIN=ARM
   ```

   Note Alternately, you can edit the application's makefile to specify a toolchain.

   This generates two files in the application directory:

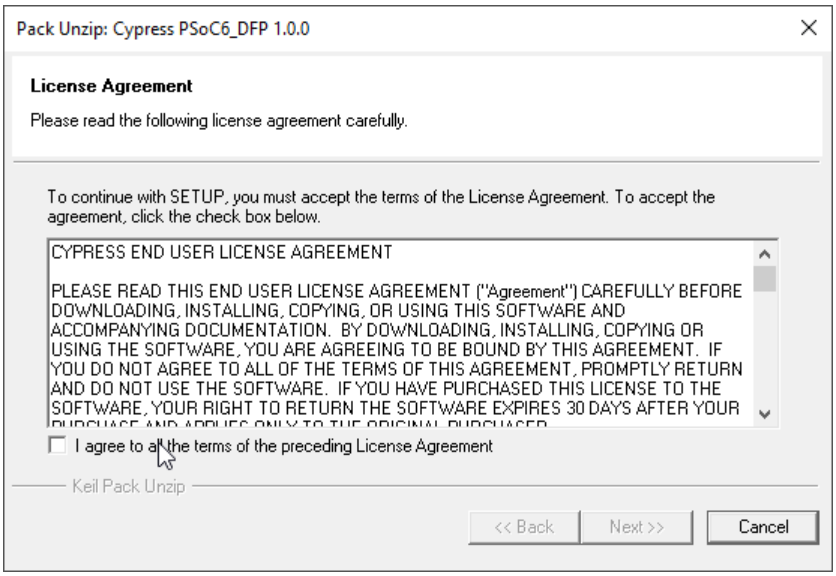
   - *mtb-example-psoc6-hello-world.cpdsc*
   - *mtb-example-psoc6-hello-world.gpdsc*

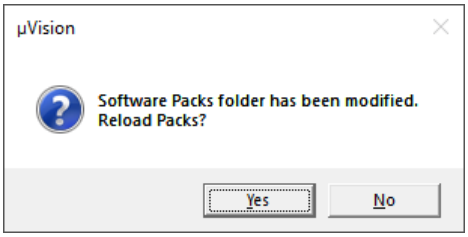
   The cpdsc file extension should have the association enabled to open it in Keil µVision.
4. Double-click the *mtb-example-psoc6-hello-world.cpdsc* file. This launches Keil µVision IDE. The first time you do this, the following dialog displays:

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

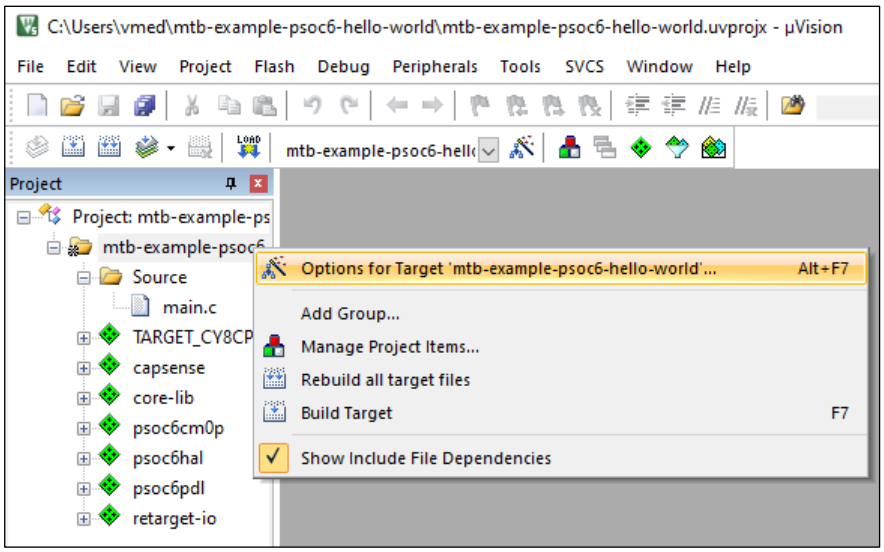
6. Follow the steps in the Pack Installer to properly install the device pack.



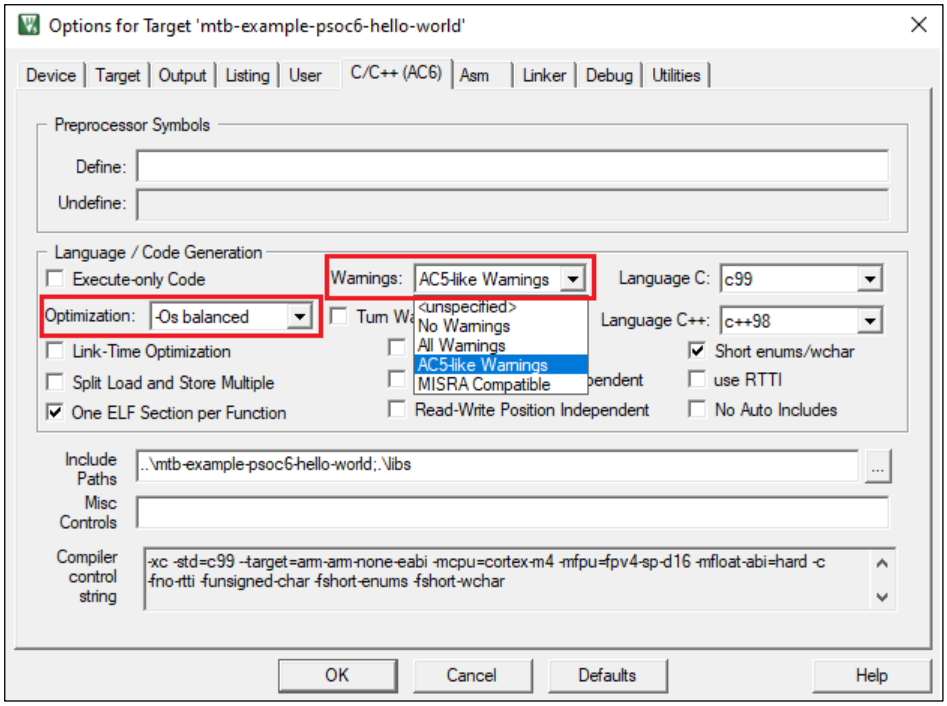


When complete, close the Pack Installer and close the Keil µVision IDE. Then double-click the .cpdsc file again and the application will be created for you in the IDE.

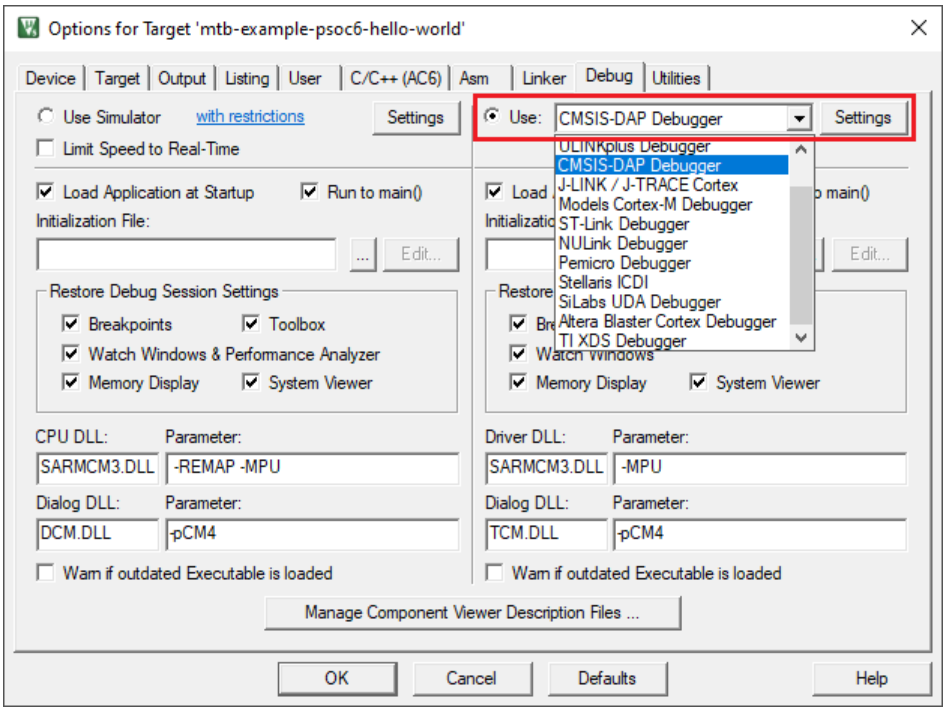
7. Right-click on the **mtb-example-psoc6-hello-world** directory in the µVision Project view, and select **Options for Target '<application-name>' …**



8. On the dialog, select the **C/C++ (AC6)** tab.
   a. Check that the **Language C** version was automatically set to c99.
   b. Select "AC5-like warnings" in the **Warnings** drop-down list.
   c. Select "-Os balanced" in the **Optimization** drop-down list.

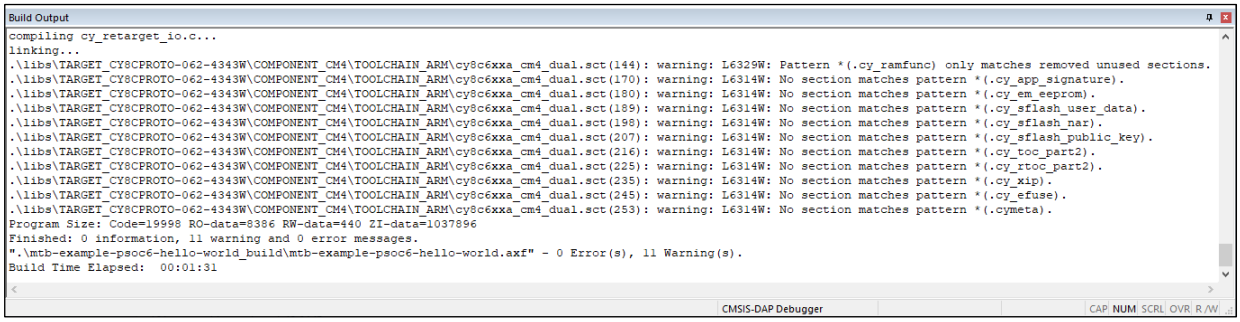


9. Select the **Debug** tab, and select KitProg3 CMSIS-DAP as an active debug adapter:



10. Click **OK** to close the Options dialog.

11. Select **Project > Build target**.



To suppress the linker warnings about unused sections defined in the linker scripts, add “6314,6329” to the **Disable Warnings** setting in the Project Linker Options.

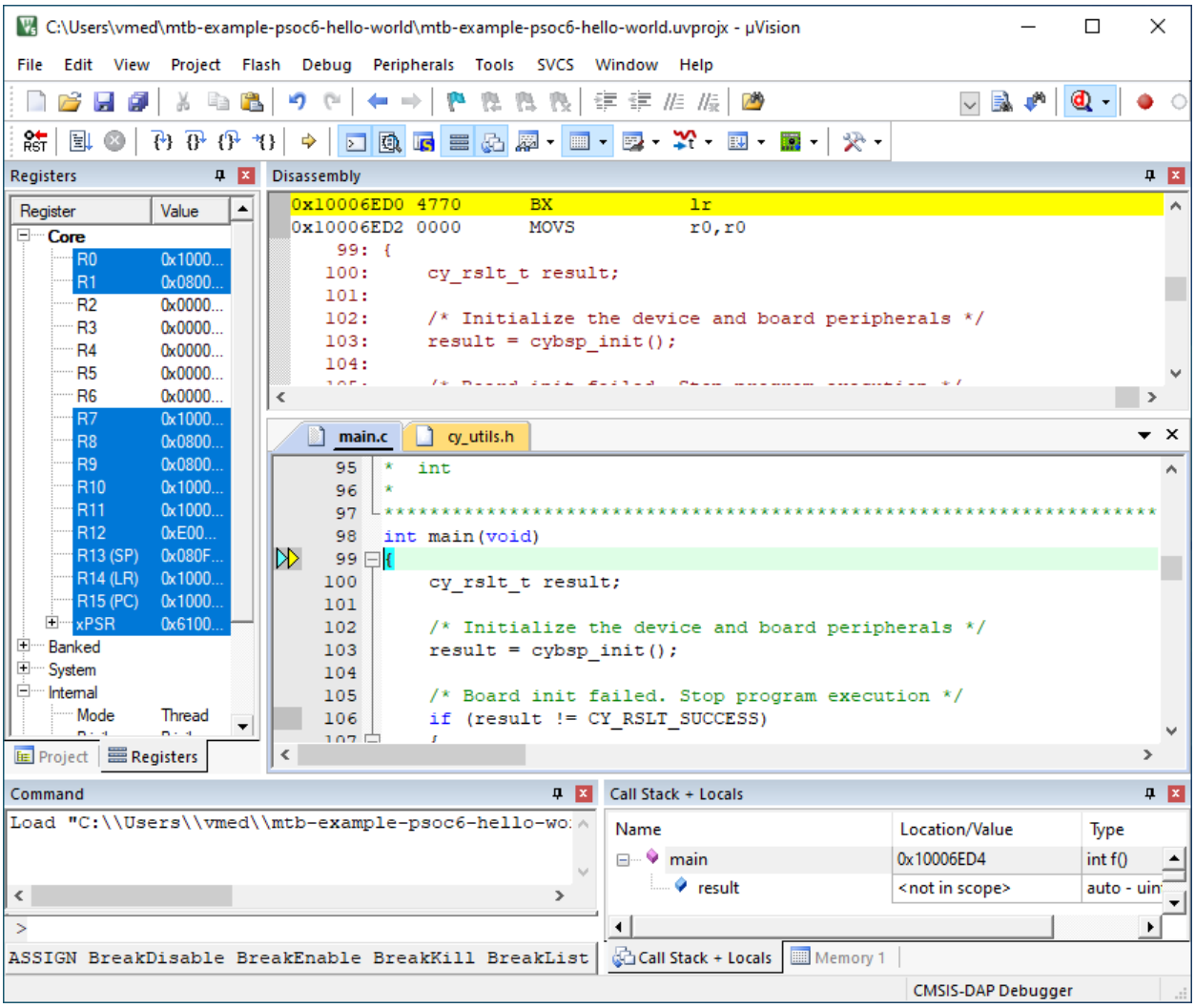
12. Connect the PSoC 6 kit to the host PC.

13. As needed, run the fw-loader tool to make sure the board firmware is upgraded to KitProg3. See KitProg3 User Guide for details. The tool is located in this directory by default:

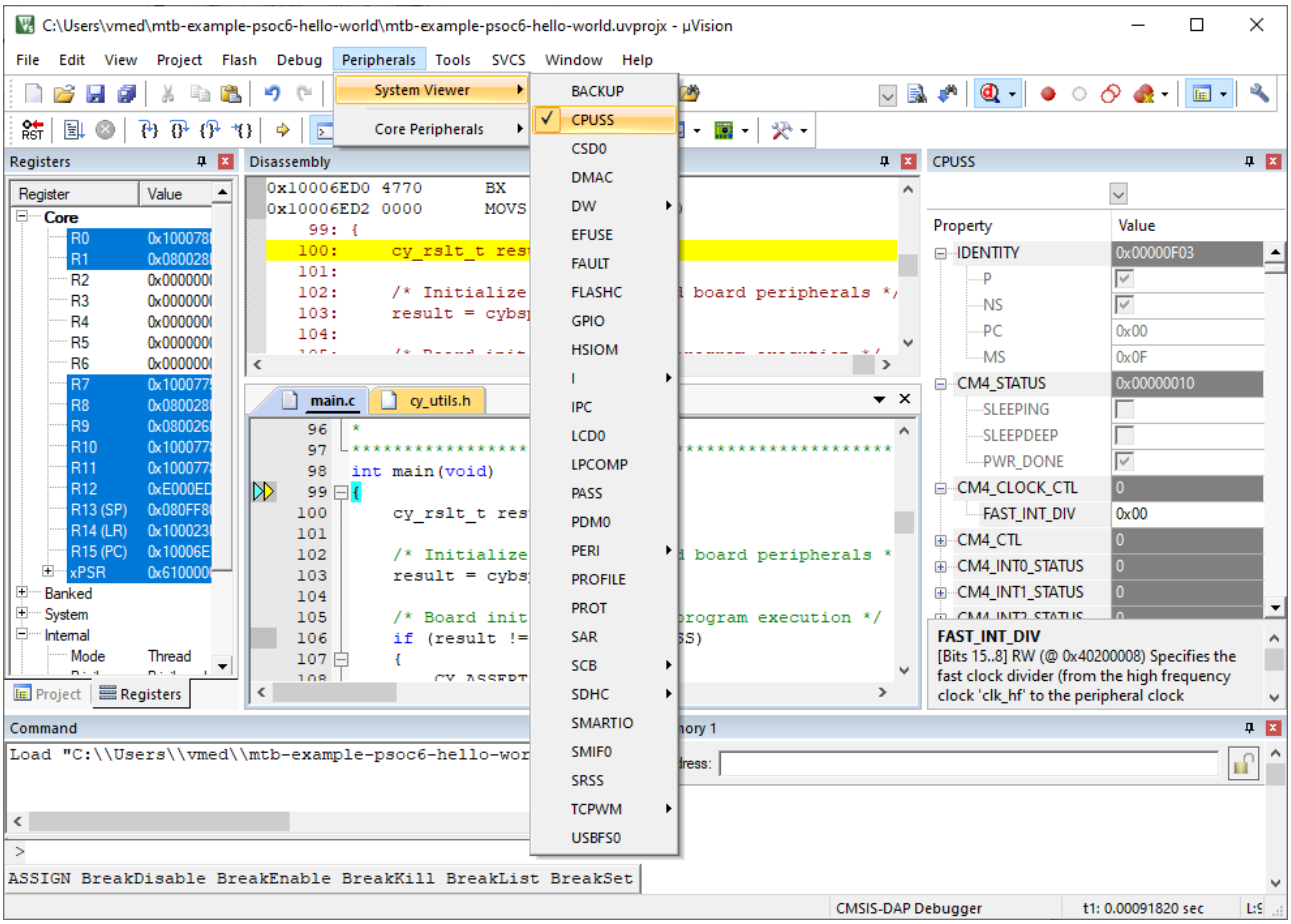
    <user_home>/ModusToolbox/tools_2.1/fw-loader/bin/

14. Select **Debug > Start/Stop Debug Session.**

15. Click **Continue** a few times to enter the main procedure and debug the application code.

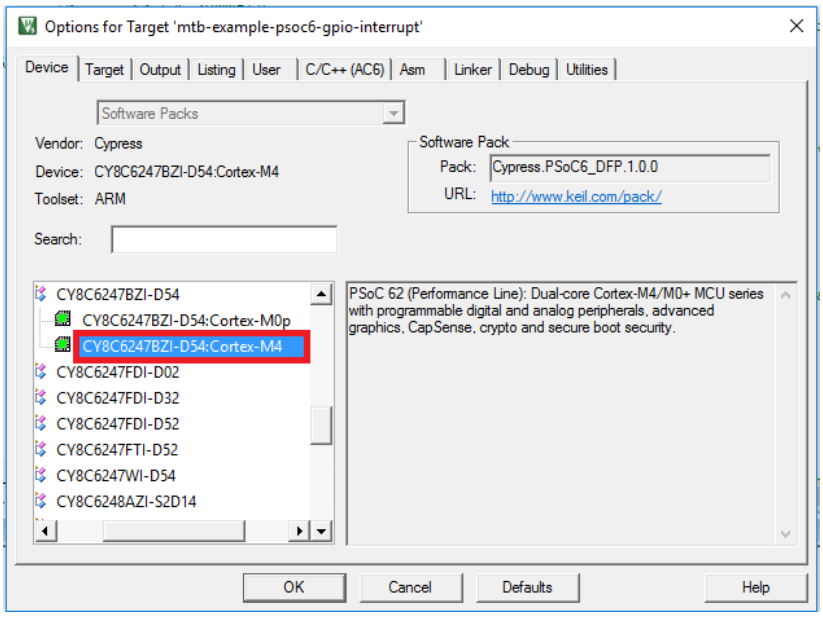


You can view the system and peripheral registers in the SVD view.

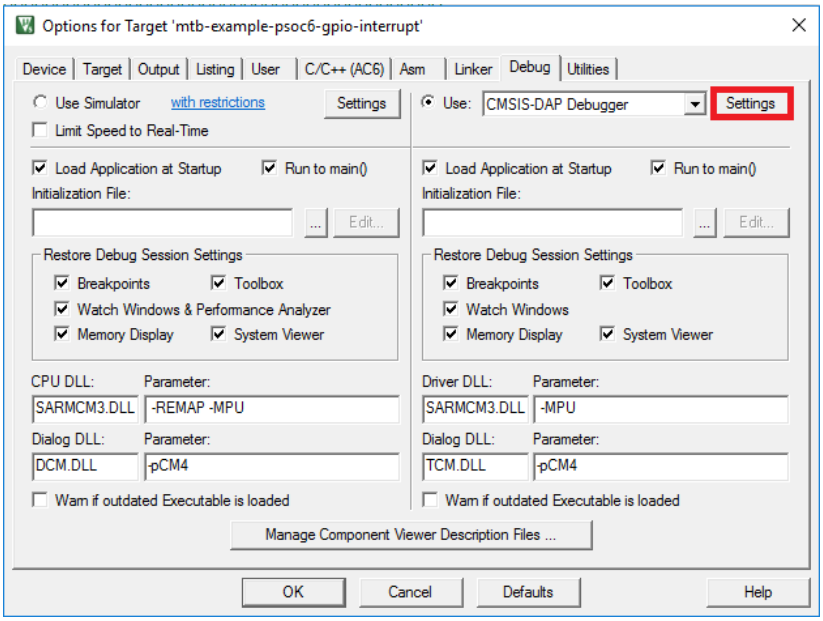


7.5.2.1 To Use KitProg3 CMSIS-DAP, ULink2 and ULink Pro debuggers

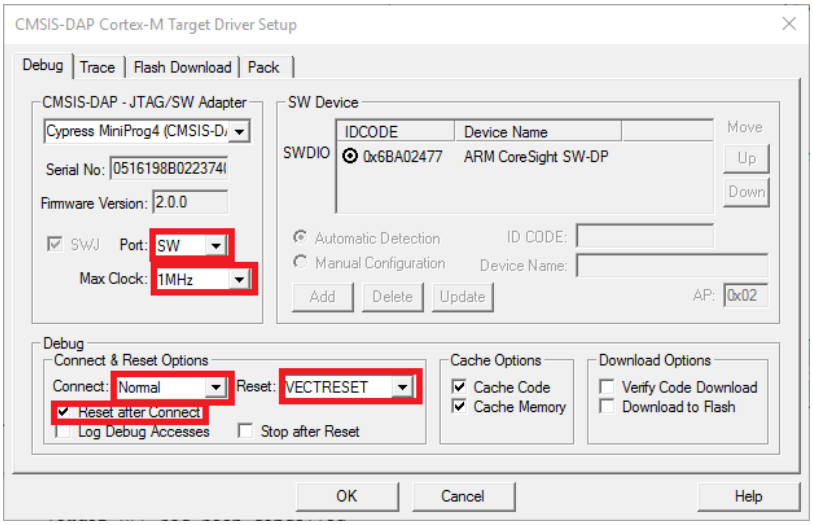
1. Select the **Device** tab in the **Options for Target** dialog and check that M4 core is selected:



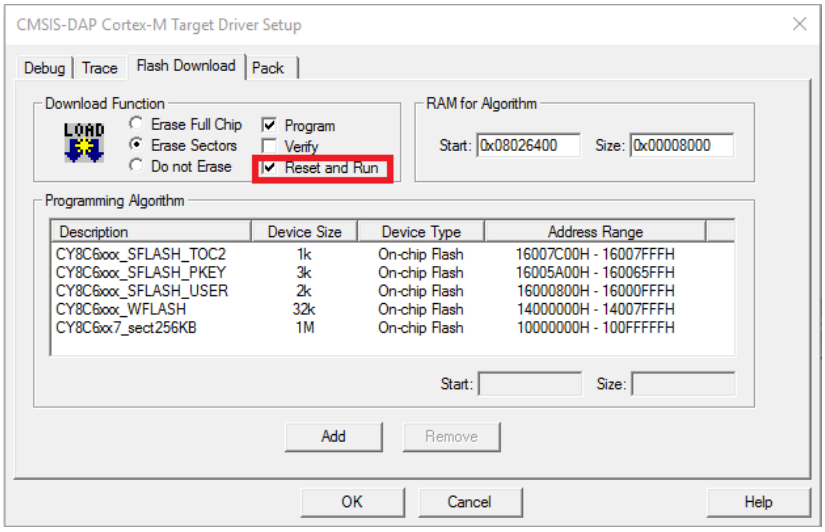
2. Select the **Debug** tab and click "Settings" to display the dialog **Target Driver Setup**:



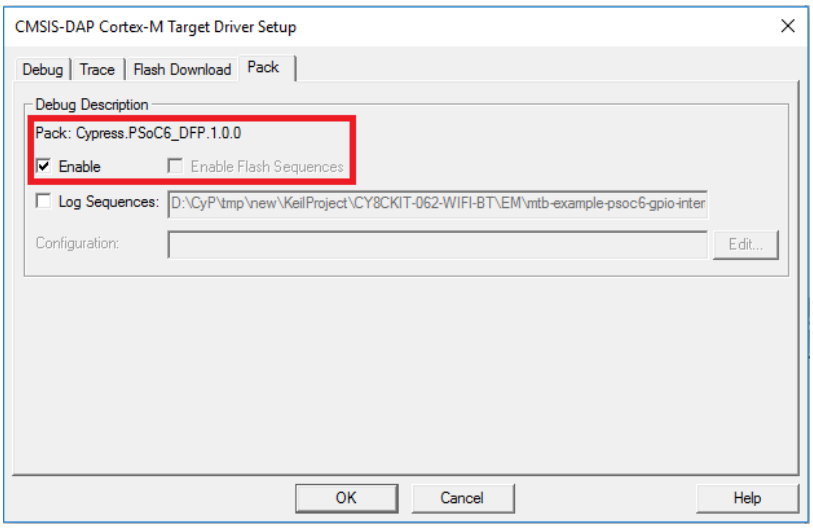
3. On the Target Driver Setup dialog, on the **Debug** tab, select the following:
   - set **Port** to “SW”
   - set **Max Clock** to “1 MHz”
   - set **Connect** to “Normal”
   - set **Reset** to “VECTRESET”
   - enable **Reset after Connect** option



4. Select the **Flash Download** tab and select “Reset and Run” option after download, if needed:



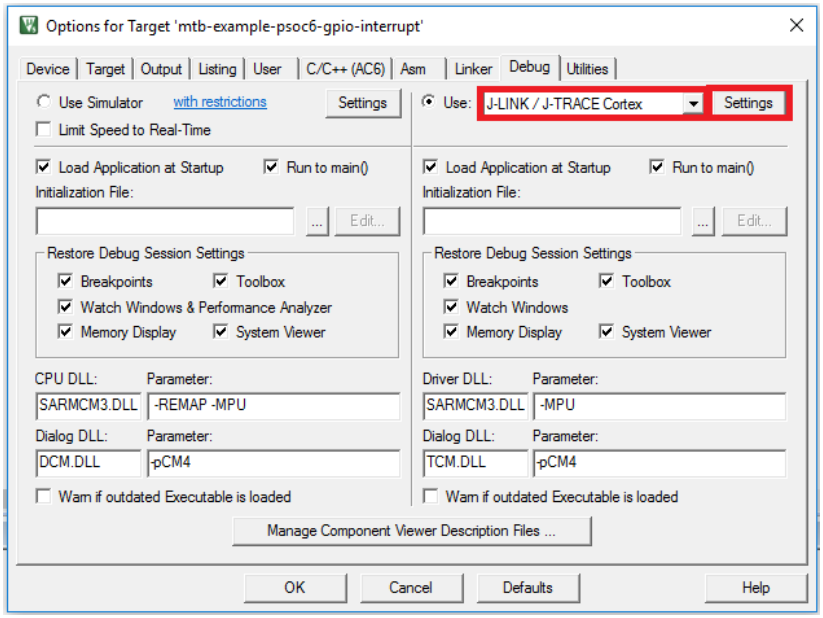
5. Select the **Pack** tab and check if Cypress PSoC6 DFP is enabled:



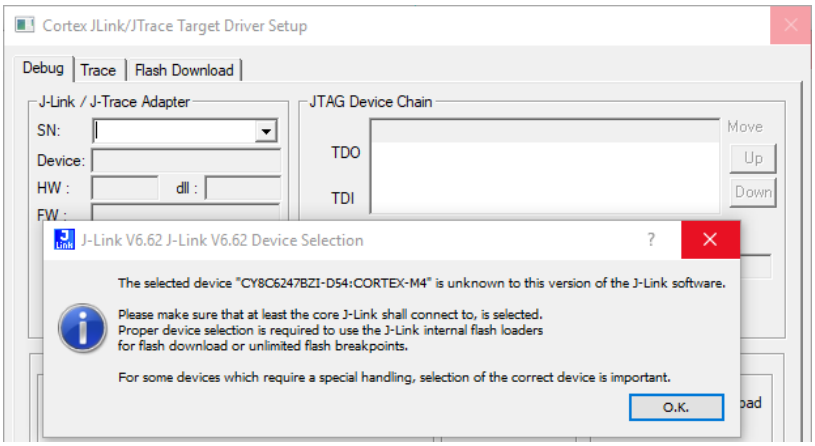
7.5.2.2 To Use J-Link debugger

1. Make sure you have J-Link software version 6.62 or newer.

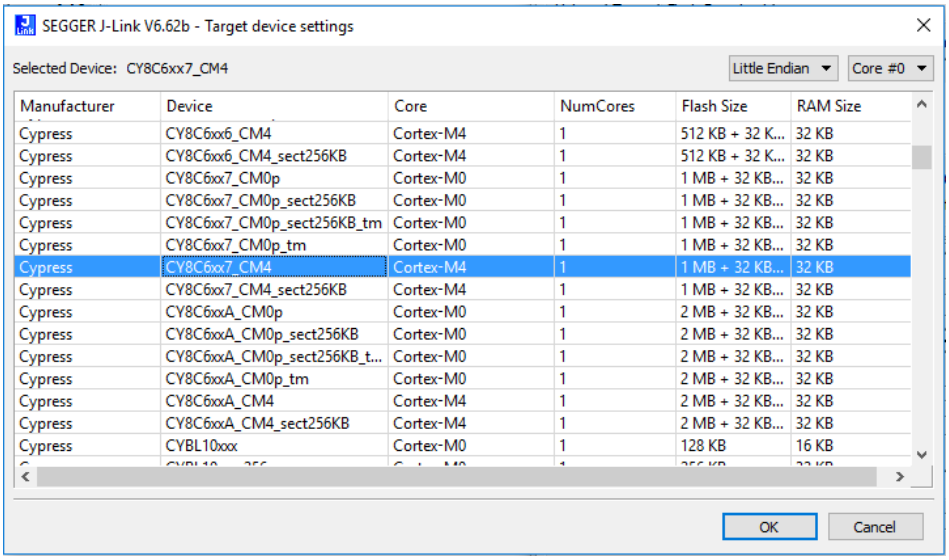
2. Select the **Debug** tab in the **Options for Target** dialog, select J-LINK / J-TRACE Cortex as debug adapter, and click "Settings":



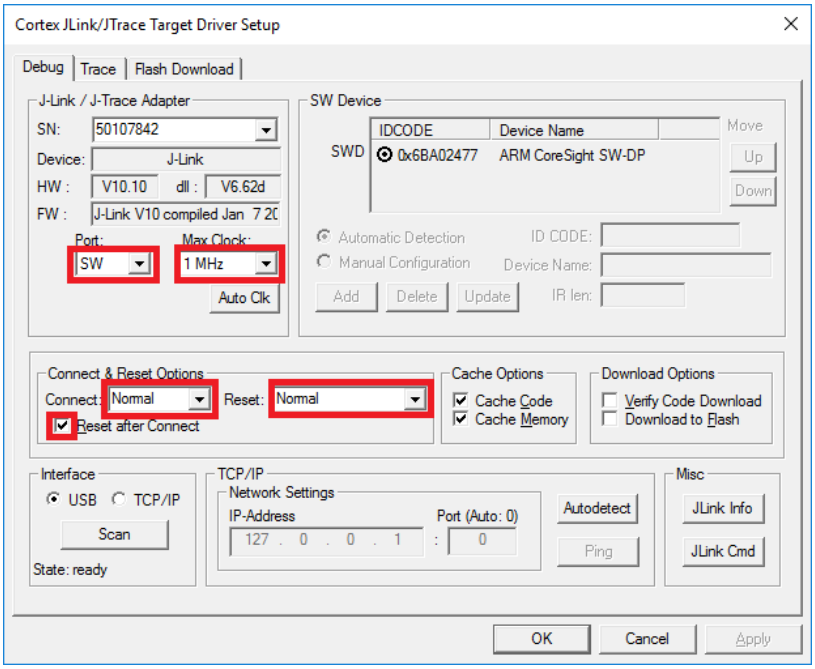
3. Click **OK** in the Device selection message box:



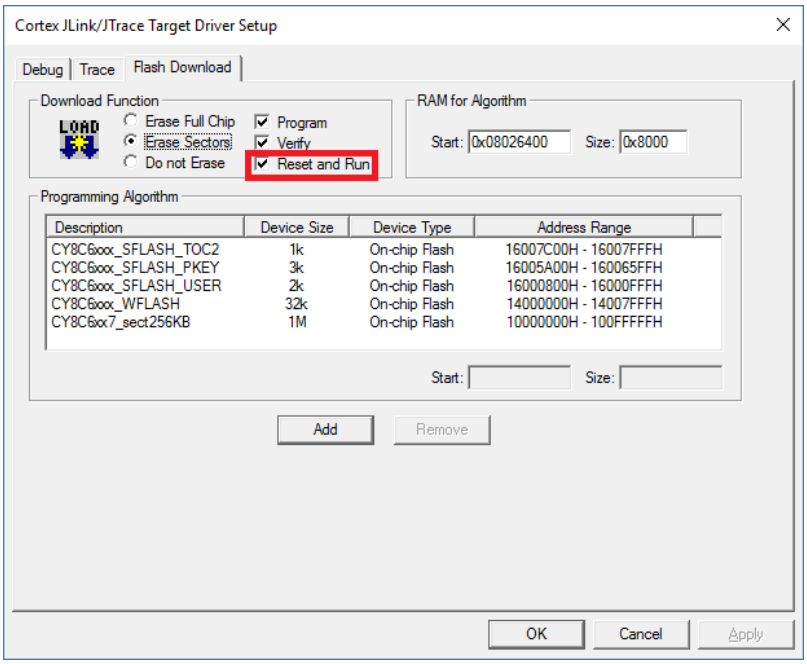
4. Select appropriate target in Wizard:



5. Go to **Debug** tab in **Target Driver Setup** dialog and select:
   - set **Port** to “SW”
   - set **Max Clock** to “1 MHz”
   - set **Connect** to “Normal”
   - set **Reset** to “Normal”
   - enable **Reset after Connect** option



6. Select the **Flash Download** tab in **Target Driver Setup** dialog and select “Reset and Run” option after download if needed:

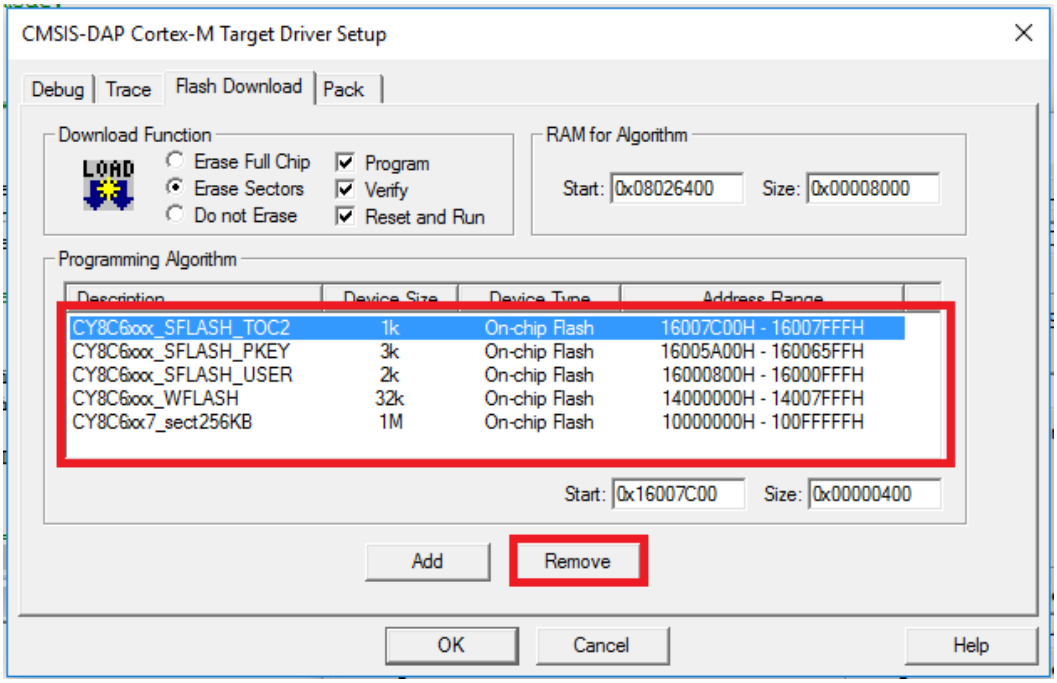


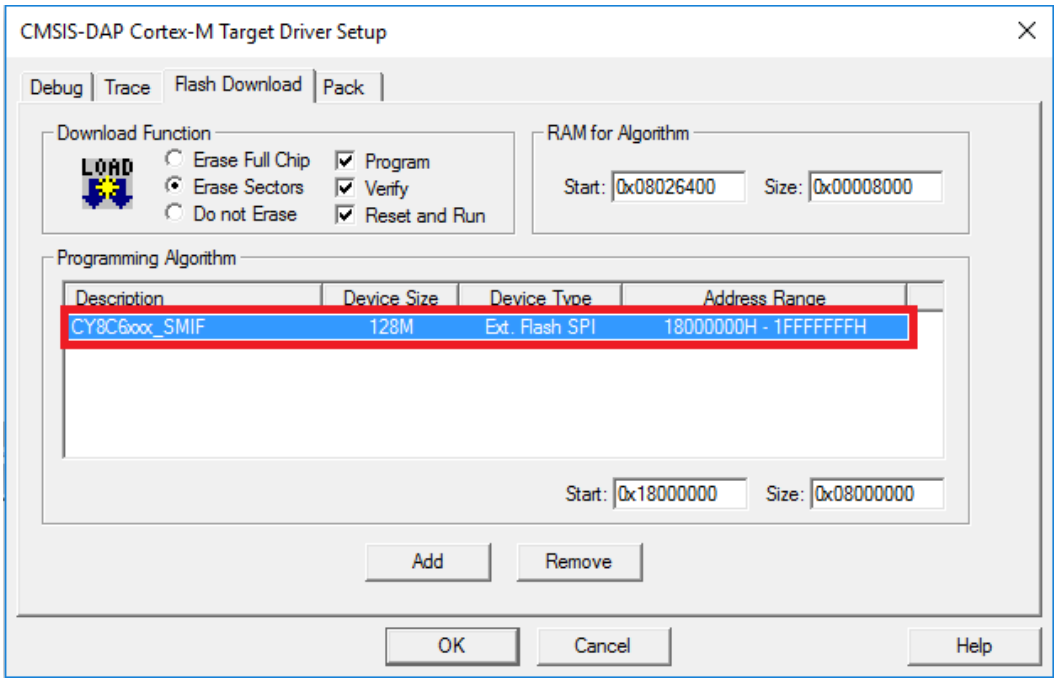
7.5.2.3 Program External Memory

1. Download internal flash as described above.

   Notice “No Algorithm found for: 18000000H - 1800FFFFH” warning.

2. Select the **Flash Download** tab in **Target Driver Setup** dialog and remove all programming algorithms for On-chip Flash and add programming algorithm for External Flash SPI:





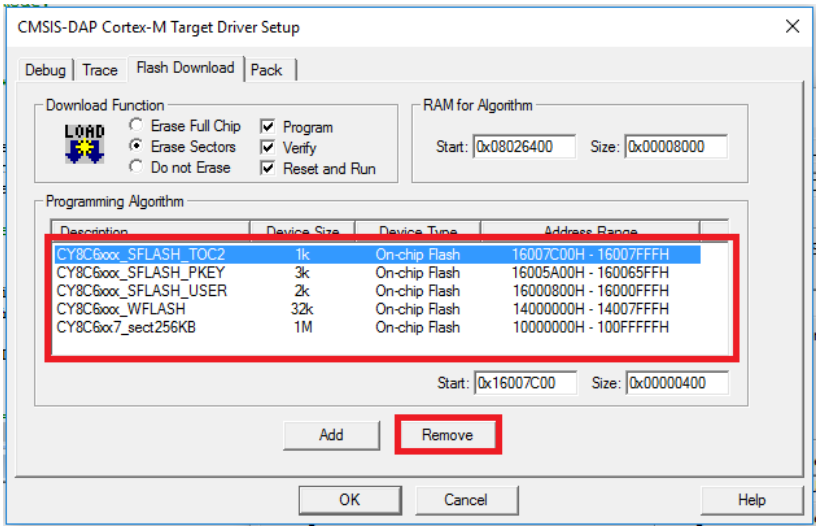
3. Download flash.

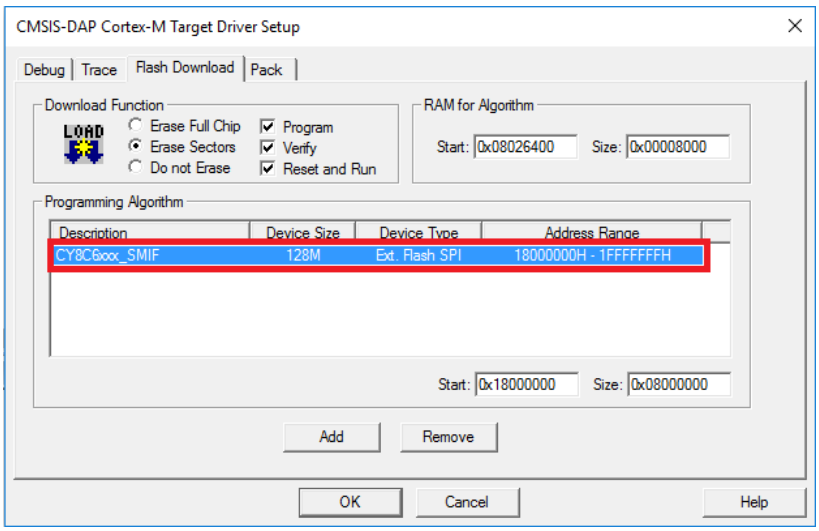
   Notice warnings:

   - No Algorithm found for: 10000000H - 1000182FH
   - No Algorithm found for: 10002000H - 10007E5BH
   - No Algorithm found for: 16007C00H - 16007DFFH

7.5.2.4 Erase External Memory

1. Select the **Flash Download** tab in **Target Driver Setup** dialog and remove all programming algorithms for On-chip Flash and add programming algorithm for External Flash SPI:





2. Click **Flash > Erase** in menu bar.

Document Revision History

Document Title: ModusToolbox™ User Guide Document Number: 002-29893		
Revision	**Date**	**Description of Change**
**	3/24/2020	New document.
*A	3/27/2020	Updates to screen captures and associate text.
*B	4/1/2020	Fix broken links.
*C	4/13/2020	Fix incorrect link.