ModusToolbox™ 2.3

User Guide

Document Number: 002-29893 Rev. *J

Cypress Semiconductor
An Infineon Technologies Company
198 Champion Court
San Jose, CA 95134-1709
www.cypress.com
www.infineon.com

Contents

1 Introduction

1.1 About this Guide

This guide provides information and instructions for using the ModusToolbox software tools provided by the version 2.3.0 installer and the make build system. This document contains the following chapters:

- This chapter describes ModusToolbox from a high level.
- Chapter 2 provides instructions 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).

1.2 What is ModusToolbox?

ModusToolbox is a modern, extensible development environment supporting a wide range of Infineon microcontroller devices. It provides a flexible set of tools and a diverse, high-quality collection of application-focused software. These include configuration tools, low-level drivers, libraries, and operating system support, most of which are compatible with Linux®, macOS®, and Windows®-hosted environments. ModusToolbox does **not** include proprietary tools or custom build environments. This means you choose your compiler, your IDE, your RTOS, and your ecosystem without compromising usability or access to our industry-leading CapSense®, AIROC™ Bluetooth®, Wi-Fi®, security, and low-power features.

1.2.1 Supported Devices

ModusToolbox supports development on the following Arm Cortex-M devices.

- PSoC 4 Configurable Microcontroller
- XMC Industrial Microcontroller
- PMG1 USB-C Power Delivery Microcontroller
- PSoC 6 MCU
- PSoC 6 Secure MCU
- AIROC Bluetooth SoC

1.2.2 Development Tools

The ModusToolbox tools package provides you with all the desktop products needed to build sophisticated, low-power embedded, connected and IoT applications. The tools enable you to create new applications (Project Creator), add or update software components (Library Manager), set up peripherals and middleware (Configurators), program and debug (OpenOCD and Device Firmware Updater), and compile (GNU C compiler).

Infineon Technologies understands that you want to pick and choose the tools and products to use, merge them into your own flows, and develop applications in ways we cannot predict. That's why ModusToolbox is not a monolithic, proprietary software tool that dictates the use of any particular IDE. For convenience, the tools package installation includes the Eclipse IDE for ModusToolbox. However, we fully support the following IDEs and their corresponding compiler technology, so you are free to develop the way you wish:

- Microsoft Visual Studio Code (VS Code)
- IAR Embedded Workbench (EW-ARM)
- Arm Microcontroller Developers Kit (µVision 5)

For detailed instructions developing ModusToolbox applications with third-party IDEs, see the Exporting to IDEs chapter in this guide.

1.2.3 Run-Time Software

ModusToolbox tools also includes an extensive collection of GitHub-hosted repos comprising Code Examples, Board Support Packages (BSP), plus middleware and applications support. We release run-time software on a quarterly “train model” schedule, and access to new or updated libraries typically does not require you to update your ModusToolbox installation.

New projects start with one of our many Code Examples that showcase everything from simple peripheral demonstrations to complete application solutions. Every Infineon kit is backed by a comprehensive BSP implementation that simplifies the software interface to the board, enables applications to be re-targeted to new hardware in no time, and can be easily extended to support your custom hardware without the usual porting and integration hassle.

The extensive middleware collection includes an ever-growing set of sensor interfaces, display support, and connectivity-focused libraries. ModusToolbox also conveniently bundles packages of all the necessary run-time components you need to leverage the following key Infineon technology focus areas:

- CapSense
- AnyCloud Wi-Fi (with AIROC Wi-Fi+Bluetooth combo devices)
- AIROC Bluetooth and Bluetooth Mesh
- Machine Learning
- Device Security (PSoC 64 MCU)

1.3 Partner Ecosystems

To support Infineon microcontrollers in our partner ecosystems, some tools and middleware from ModusToolbox are also integrated into Mbed OS and Amazon FreeRTOS. Refer to mbed.com and aws.amazon.com/freertos, respectively, to learn more about developing applications in those environments.

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. Plus, 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 tools package is located on the Cypress website:

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

You can install the software 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 tools package work out of the box without any changes required. Simply launch the executable for the applicable GUI tool. On Windows, most tools are on the **Start** menu.

2.1.2 CLI Set-up Instructions

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

- For **Windows**, the tools package provides a command-line utility called "modus-shell." You can run this from the **Start** menu, or navigate to the following installation directory and run *Cygwin.bat* :

  <install_path>/ModusToolbox/tools_2.3/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 of the tool, as well as how to use them. 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
```

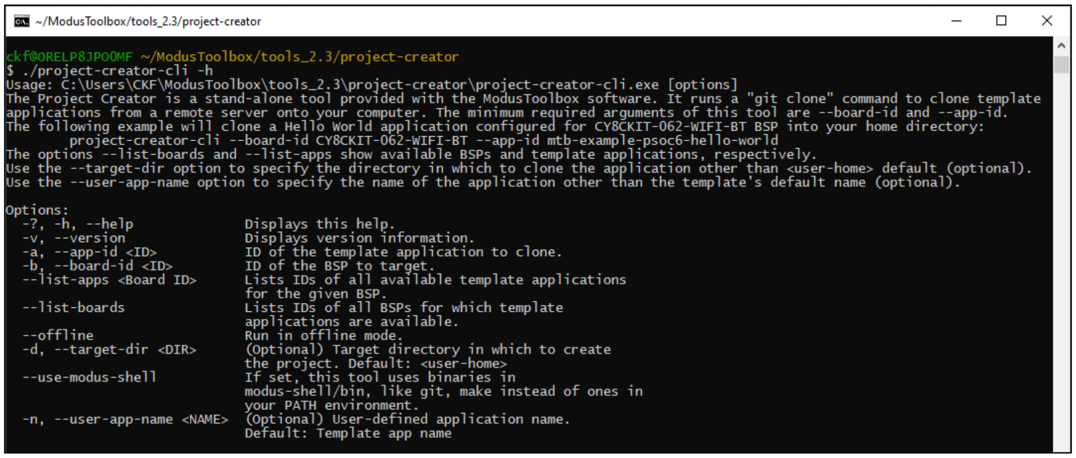
Note This help documentation is part of the base library, and it may also contain additional information specific to a BSP.

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
```



~/ModusToolbox/tools_2.3/project-creator

```
ckf@ORELP8JPOOMF ~/ModusToolbox/tools_2.3/project-creator
$ ./project-creator-cli -h
Usage: C:\Users\CKF\ModusToolbox\tools_2.3\project-creator\project-creator-cli.exe [options]
The Project Creator is a stand-alone tool provided with the ModusToolbox software. It runs a "git clone" command to clone template
applications from a remote server onto your computer. The minimum required arguments of this tool are --board-id and --app-id.
The following example will clone a Hello World application configured for CY8CKIT-062-WIFI-BT BSP into your home directory:
        project-creator-cli --board-id CY8CKIT-062-WIFI-BT --app-id mtb-example-psoc6-hello-world
The options --list-boards and --list-apps show available BSPs and template applications, respectively.
Use the --target-dir option to specify the directory in which to clone the application other than <user-home> default (optional).
Use the --user-app-name option to specify the name of the application other than the template's default name (optional).

Options:
  -?, -h, --help              Displays this help.
  -v, --version               Displays version information.
  -a, --app-id <ID>           ID of the template application to clone.
  -b, --board-id <ID>         ID of the BSP to target.
  --list-apps <Board ID>      Lists IDs of all available template applications
                              for the given BSP.
  --list-boards               Lists IDs of all BSPs for which template
                              applications are available.
  --offline                   Run in offline mode.
  -d, --target-dir <DIR>      (Optional) Target directory in which to create
                              the project. Default: <user-home>
  --use-modus-shell           If set, this tool uses binaries in
                              modus-shell/bin, like git, make instead of ones in
                              your PATH environment.
  -n, --user-app-name <NAME>  (Optional) User-defined application name.
                              Default: Template app name
```

2.3 Create Applications

ModusToolbox provides the Project Creator as both a GUI tool and a command line tool to easily create one or more ModusToolbox applications. See Project Creator Tools. If you prefer not to use the Project Creator tools, you can use the `git clone` command directly. See git clone. However, be sure to also run the `make getlibs` command in the application directory. See make getlibs. You can then use those application files in your preferred IDE or from the command line.

Note Beginning with the ModusToolbox 2.2 release, we structure applications with the MTB flow. Using this flow, applications can share BSPs and libraries. If needed, different applications can use different versions of the same BSP/library. Sharing resources reduces the number of files on your computer and speeds up subsequent application creation time. Shared BSPs, libraries, and versions are located in the *mtb_shared* directory adjacent to your application directories. You can easily switch a shared BSP or library to become local to a specific application, or back to being shared. Refer to the Library Manager User Guide for details.

Looking ahead, most example applications will use the MTB flow. However, there are still various applications that use the previous flow, now called the LIB flow, and these applications generally do not share BSPs and libraries. ModusToolbox fully supports both flows, but it only supports one flow or the other for a given application.

For simplicity, this guide focuses on the MTB flow. For details about how the LIB flow works, refer to the ModusToolbox 2.1 revision of this guide, located here:

https://www.cypress.com/file/504361/download

2.3.1 Project Creator Tools

The Project Creator tools run the `git clone` command for the selected code example(s) and create a directory at the specified location with the specified name. The tools also updates the application makefile and create a *<BSP-NAME>.mtb* file based on the specified BSP. That *.mtb* file contains the following:

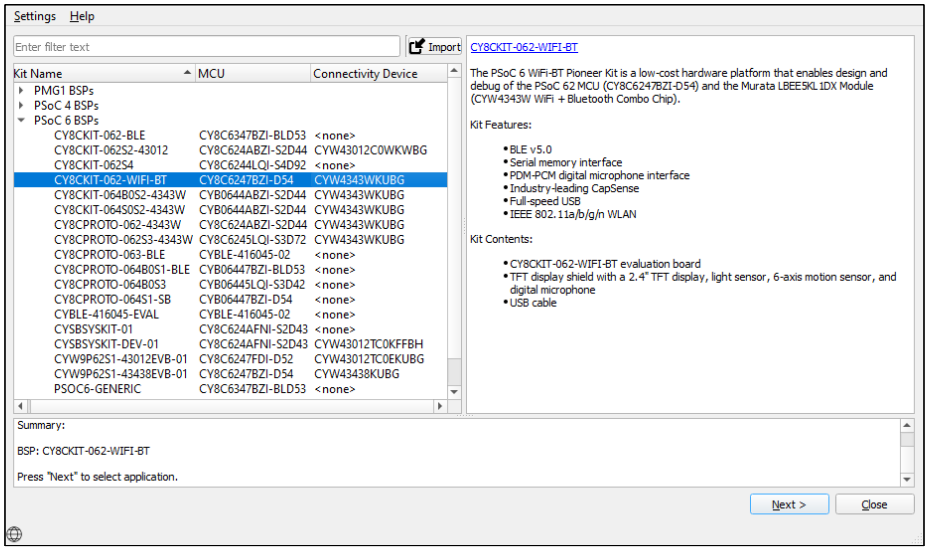
- The URL of the git repo where the BSP contents can be found.
- The commit (version of the library) to checkout / make visible / use in the application.
- A variable of where to put the BSP on disk (shared or local to the application).

The Project Creator tools then run the `make getlibs` command to read the BSP manifest file, resolve dependencies, and import libraries. Depending on the settings in the application and manifest, the tools put everything into application directories and an *mtb_shared* directory. In most cases, BSPs are placed local to the application while libraries are shared.

2.3.1.1 Project Creator GUI

The Project Creator GUI tool provides a series of screens to select a BSP and code example, specify the application name and location, as well as select target IDE. The tool displays various messages during the application creation process. Refer to the Project Creator Guide for more details. Open the Project Creator GUI tool from the Windows **Start** menu or by running the executable file installed in the following directory by default:

<install_path>/ModusToolbox/tools_2.3/project-creator/



Kit Name	MCU	Connectivity Device
PMG1 BSPs		
PSoC 4 BSPs		
PSoC 6 BSPs		
CY8CKIT-062-BLE	CY8C6347BZI-BLD53	<none>
CY8CKIT-062S2-43012	CY8C624ABZI-S2D44	CYW43012C0WKWBG
CY8CKIT-062S4	CY8C6244LQI-S4D92	<none>
CY8CKIT-062-WIFI-BT	CY8C6247BZI-D54	CYW4343WKUBG
CY8CKIT-064B0S2-4343W	CYB0644ABZI-S2D44	CYW4343WKUBG
CY8CKIT-064S0S2-4343W	CYB0644ABZI-S2D44	CYW4343WKUBG
CY8CPROTO-062-4343W	CY8C624ABZI-S2D44	CYW4343WKUBG
CY8CPROTO-062S3-4343W	CY8C6245LQI-S3D72	CYW4343WKUBG
CY8CPROTO-063-BLE	CYBLE-416045-02	<none>
CY8CPROTO-064B0S1-BLE	CYB06447BZI-BLD53	<none>
CY8CPROTO-064B0S3	CYB06445LQI-S3D42	<none>
CY8CPROTO-064S1-SB	CYB06447BZI-D54	<none>
CYBLE-416045-EVAL	CYBLE-416045-02	<none>
CYSBSYSKIT-01	CY8C624AFNI-S2D43	<none>
CYSBSYSKIT-DEV-01	CY8C624AFNI-S2D43	CYW43012TC0KFFBH
CYW9P62S1-43012EVB-01	CY8C6247FDI-D52	CYW43012TC0EKUBG
CYW9P62S1-43438EVB-01	CY8C6247BZI-D54	CYW43438KUBG
PSOC6-GENERIC	CY8C6347BZI-BLD53	<none>

CY8CKIT-062-WIFI-BT

The PSoC 6 WiFi-BT Pioneer Kit is a low-cost hardware platform that enables design and debug of the PSoC 62 MCU (CY8C6247BZI-D54) and the Murata LBEE5KL1DX Module (CYW4343W WiFi + Bluetooth Combo Chip).

Kit Features:

- BLE v5.0
- Serial memory interface
- PDM-PCM digital microphone interface
- Industry-leading CapSense
- Full-speed USB
- IEEE 802.11a/b/g/n WLAN

Kit Contents:

- CY8CKIT-062-WIFI-BT evaluation board
- TFT display shield with a 2.4" TFT display, light sensor, 6-axis motion sensor, and digital microphone
- USB cable

Summary:

BSP: CY8CKIT-062-WIFI-BT

Press "Next" to select application.

The option to select a target IDE generates necessary files for that IDE. If you launch the Project Creator GUI tool from the included Eclipse-based IDE, it seamlessly exports the created application for use in the Eclipse IDE.

2.3.1.2 project-creator-cli

You can also use the 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_2.3/project-creator/*). To see all the options available, run the tool with the `-h` option:

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

The following example shows 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"
```

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 *.mtb* 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.

2.3.2 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):

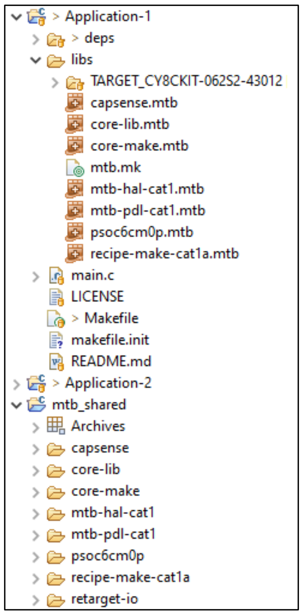
```
git clone <URL>
```

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.

Note When using the `git clone` command directly, be sure to also run the `make getlibs` command in the application directory. See make getlibs. Also, each code example has a default BSP included in the application's *deps* subdirectory. If you want to use a different BSP, you must create a *.mtb* file for it in the *deps* subdirectory before running `make getlibs`, and you must change the `TARGET` variable in the Makefile.

2.3.3 Typical Application Contents

After an application has been created for the MTB flow and all the libraries have been imported, it contains the following basic files and directories as shown in the following image:



```
> Application-1
    > deps
    libs
        TARGET_CY8CKIT-062S2-43012
        capsense.mtb
        core-lib.mtb
        core-make.mtb
        mtb.mk
        mtb-hal-cat1.mtb
        mtb-pdl-cat1.mtb
        psoc6cm0p.mtb
        recipe-make-cat1a.mtb
    main.c
    LICENSE
    > Makefile
    makefile.init
    README.md
> Application-2
mtb_shared
    Archives
    capsense
    core-lib
    core-make
    mtb-hal-cat1
    mtb-pdl-cat1
    psoc6cm0p
    recipe-make-cat1a
    retarget-io
```

2.3.3.1 *Application Directory*

This directory contains the application source code, makefile, readme file, as well as the *deps* and *libs* subdirectories. If you create multiple applications, there will be multiple application directories contained in the same directory structure or workspace.

- **Source Code** – This is one or more files for your application's code. Often it is named *main.c*, but it could be more than one file and the files could have almost any name. Source code files can also be grouped into a subdirectory anywhere in the application's directory (for example, *sources/main.c*).
- **Makefile** – This is the application's makefile, which contains configuration information. See the ModusToolbox Build System chapter for more details.
- **deps Subdirectory** – By default, this subdirectory contains *.mtb* files using the MTB flow.
    - Initially, this subdirectory contains only the *<BSP>.mtb* file for the BSP you selected for the application.
    - It could also contain *<library>.mtb* files for libraries that were included directly or for which you changed using the

Library Manager. See the Update BSPs and Libraries section for details.

- This subdirectory also contains the *locking_commit.log* file, which keeps track of the version for each dependent library.

- **libs Subdirectory** – This subdirectory may contain different types of files generated by the make getlibs process, based on how the application is created. You can regenerate these files using the `make getlibs` command, so you do not need to add these files to source control.
  - This subdirectory contains BSPs that are local to the application (that is, not shared).
  - If you update your application to specify any libraries to be local as well, then this directory will also contain source code for those libraries.
  - By default, this subdirectory contains the *<library>.mtb* files for libraries included as indirect dependencies of the BSP or other libraries.
  - This directory also contains the *mtb.mk* file that lists the shared libraries and their versions.

2.3.3.2 *mtb_shared Directory*

Typically, a new application also includes a *mtb_shared* directory adjacent to the application directory, and this is where the shared BSP and libraries are cloned by default. 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 *mtb_shared* directory.

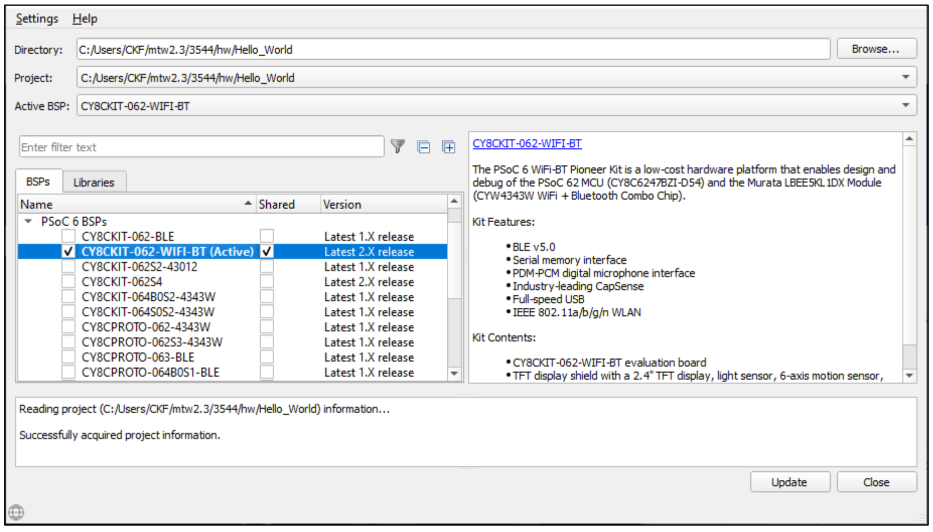
2.4 Update BSPs and Libraries

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 will 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. Open the Library Manager tool from the application directory using the `make modlibs` command.

The Library Manager opens for the selected application and its available BSPs and libraries.



Directory: C:/Users/CKF/mtw2.3/3544/hw/Hello_World

Project: C:/Users/CKF/mtw2.3/3544/hw/Hello_World

Active BSP: CY8CKIT-062-WIFI-BT

Name	Shared	Version
PSoC 6 BSPs		
CY8CKIT-062-BLE		Latest 1.X release
✓ CY8CKIT-062-WIFI-BT (Active)	✓	Latest 2.X release
CY8CKIT-062S2-43012		Latest 1.X release
CY8CKIT-062S4		Latest 2.X release
CY8CKIT-064B0S2-4343W		Latest 1.X release
CY8CKIT-064S0S2-4343W		Latest 1.X release
CY8CPROTO-062-4343W		Latest 1.X release
CY8CPROTO-062S3-4343W		Latest 1.X release
CY8CPROTO-063-BLE		Latest 1.X release
CY8CPROTO-064B0S1-BLE		Latest 1.X release

CY8CKIT-062-WIFI-BT

The PSoC 6 WiFi-BT Pioneer Kit is a low-cost hardware platform that enables design and debug of the PSoC 62 MCU (CY8C6247BZI-D54) and the Murata LBEE5KL1DX Module (CYW4343W WiFi + Bluetooth Combo Chip).

Kit Features:

- BLE v5.0
- Serial memory interface
- PDM-PCM digital microphone interface
- Industry-leading CapSense
- Full-speed USB
- IEEE 802.11a/b/g/n WLAN

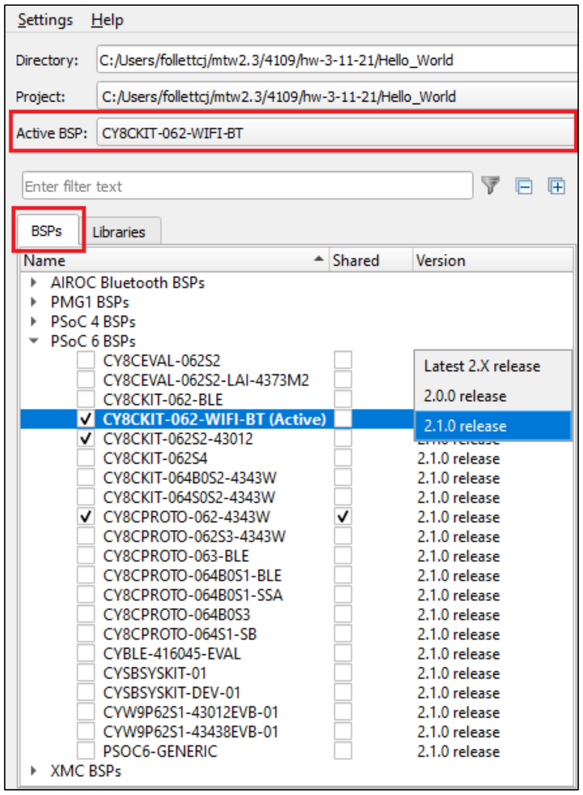
Kit Contents:

- CY8CKIT-062-WIFI-BT evaluation board
- TFT display shield with a 2.4" TFT display, light sensor, 6-axis motion sensor,

```
Reading project (C:/Users/CKF/mtw2.3/3544/hw/Hello_World) information...

Successfully acquired project information.
```

Note There are several ways to open the Library Manager; refer to the Library Manager Guide for more details.

The Library Manager tool provides a field to select the **Active BSP**. It also includes two tabs to view and update **BSPs** and **Libraries**.



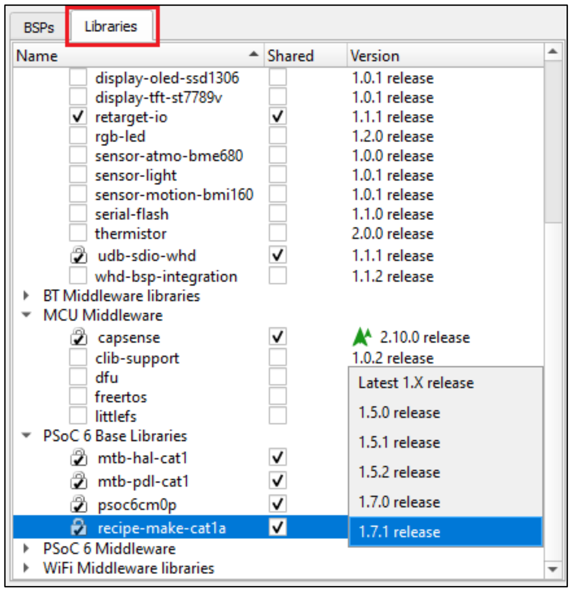
Directory: C:/Users/follettcj/mtw2.3/4109/hw-3-11-21/Hello_World

Project: C:/Users/follettcj/mtw2.3/4109/hw-3-11-21/Hello_World

Active BSP: CY8CKIT-062-WIFI-BT

Name	Shared	Version
AIROC Bluetooth BSPs		
PMG1 BSPs		
PSoC 4 BSPs		
PSoC 6 BSPs		
CY8CEVAL-062S2		
CY8CEVAL-062S2-LAI-4373M2		
CY8CKIT-062-BLE		
✓ CY8CKIT-062-WIFI-BT (Active)		2.1.0 release
✓ CY8CKIT-062S2-43012		
CY8CKIT-062S4		2.1.0 release
CY8CKIT-064B0S2-4343W		2.1.0 release
CY8CKIT-064S0S2-4343W		2.1.0 release
✓ CY8CPROTO-062-4343W	✓	2.1.0 release
CY8CPROTO-062S3-4343W		2.1.0 release
CY8CPROTO-063-BLE		2.1.0 release
CY8CPROTO-064B0S1-BLE		2.1.0 release
CY8CPROTO-064B0S1-SSA		2.1.0 release
CY8CPROTO-064B0S3		2.1.0 release
CY8CPROTO-064S1-SB		2.1.0 release
CYBLE-416045-EVAL		2.1.0 release
CYSBSYSKIT-01		2.1.0 release
CYSBSYSKIT-DEV-01		2.1.0 release
CYW9P62S1-43012EVB-01		2.1.0 release
CYW9P62S1-43438EVB-01		2.1.0 release
PSOC6-GENERIC		2.1.0 release
XMC BSPs		

Latest 2.X release
2.0.0 release
2.1.0 release



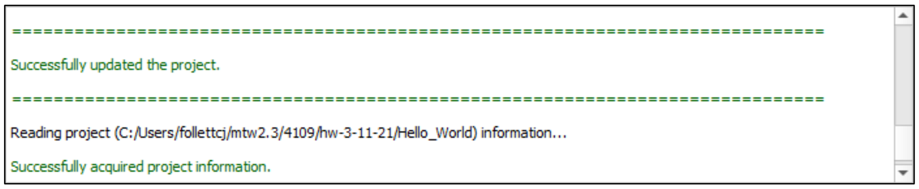
Name	Shared	Version
display-oled-ssd1306		1.0.1 release
display-tft-st7789v		1.0.1 release
✓ retarget-io	✓	1.1.1 release
rgb-led		1.2.0 release
sensor-atmo-bme680		1.0.0 release
sensor-light		1.0.1 release
sensor-motion-bmi160		1.0.1 release
serial-flash		1.1.0 release
thermistor		2.0.0 release
udb-sdio-whd	✓	1.1.1 release
whd-bsp-integration		1.1.2 release
BT Middleware libraries		
MCU Middleware		
capsense	✓	2.10.0 release
clib-support		1.0.2 release
dfu		
freertos		
littlefs		
PSoC 6 Base Libraries		
mtb-hal-cat1	✓	
mtb-pdl-cat1	✓	
psoc6cm0p	✓	
recipe-make-cat1a	✓	
PSoC 6 Middleware		
WiFi Middleware libraries		

Latest 1.X release
1.5.0 release
1.5.1 release
1.5.2 release
1.7.0 release
1.7.1 release

Make changes to BSPs and libraries as follows:

- Select one or more check boxes under **Name** for the items to add. Deselect check boxes for items to remove.
- Specify whether items are shared (placed in the *mtb_shared* directory) or local to the application (placed in the *libs* subdirectory) by selecting/deselecting the **Shared** check box.
- Choose an appropriate **Version** for each item.

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



```
================================================================================
Successfully updated the project.
================================================================================
Reading project (C:/Users/follettcj/mtw2.3/4109/hw-3-11-21/Hello_World) information...
Successfully acquired project information.
```

2.4.2 make getlibs

In the MTB flow, the Project Creator tools and the Library Manager tool run the `make getlibs` command to search for all *.mtb* files in the application directory. Each *.mtb* file contains information used when the application is created. These files are parsed, and the libraries are cloned into a directory named *mtb_shared*.

If you ran the `git clone` command manually and did not use the Library Manager, then your application will contain only default *.mtb* files. You must 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 *.mtb* file in the */deps* subdirectory that includes a URL to the desired BSP location.

Note ModusToolbox applications that use the LIB flow contain*.lib* files in the *deps* subdirectory. If an application uses the MTB flow, then all *.lib* files are ignored.

When you are ready to update your application, open the appropriate shell (see CLI Set-up Instructions) and run the following command in the application directory:

```
make getlibs
```

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. 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.

Before configuring your device, you must decide how your application will interact with the hardware; see Application Layers. That decision affects how you configure settings for devices, peripherals, and libraries.

IMPORTANT Before you make changes to settings in configurators, you should first copy the configuration information to the application and override the BSP configuration or create a custom BSP. See details about BSPs in the Board Support Packages chapter. If you make changes to a standard BSP library, it will cause the repo to become dirty. Additionally, if the BSP is in the shared asset repository, changes will impact all applications that use the shared BSP. If this happens, refer to KBA231252.

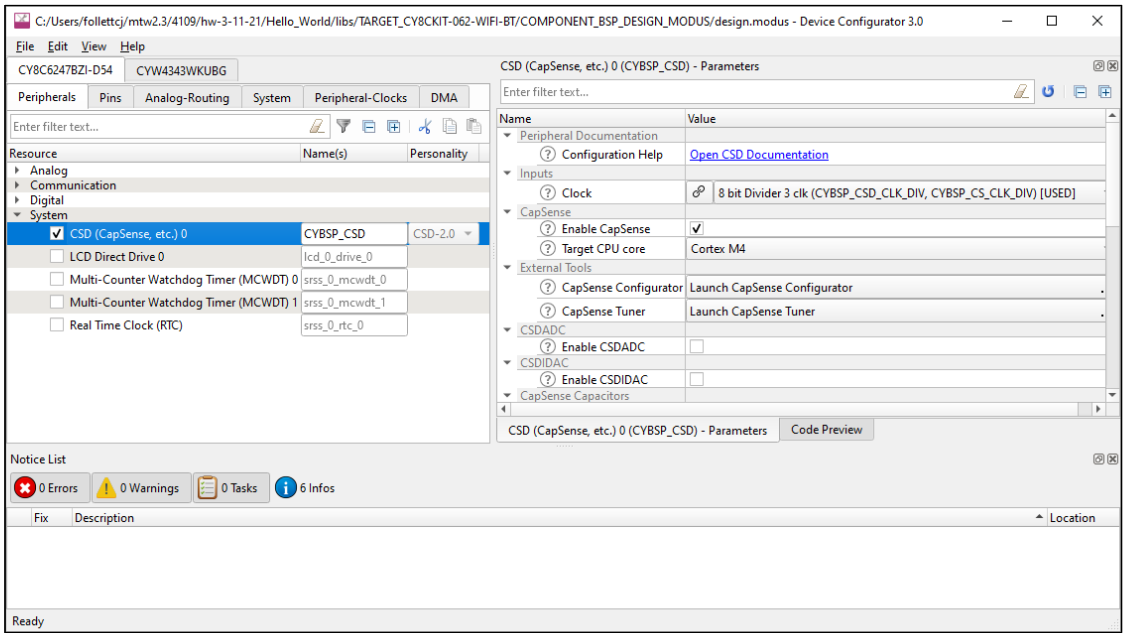
The configurators can be run as GUIs to easily update various parameters and settings. Most can also be run as command line tools to regenerate code as part of a script. For more information about configurators, see the Configurators section in the ModusToolbox Software Overview chapter. Also, each configurator 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

You can open various configurator GUIs using the appropriate make command from the application directory. For example, to open the Device Configurator, run:

```
make config
```

This opens the Device Configurator with the current application's *design.modus* configuration file.



C:/Users/follettcj/mtw2.3/4109/hw-3-11-21/Hello_World/libs/TARGET_CY8CKIT-062-WIFI-BT/COMPONENT_BSP_DESIGN_MODUS/design.modus - Device Configurator 3.0

File Edit View Help

CY8C6247BZI-D54 | CYW4343WKUBG

Peripherals | Pins | Analog-Routing | System | Peripheral-Clocks | DMA

Resource	Name(s)	Personality
Analog		
Communication		
Digital		
System		
✓ CSD (CapSense, etc.) 0	CYBSP_CSD	CSD-2.0
LCD Direct Drive 0	lcd_0_drive_0	
Multi-Counter Watchdog Timer (MCWDT) 0	srss_0_mcwdt_0	
Multi-Counter Watchdog Timer (MCWDT) 1	srss_0_mcwdt_1	
Real Time Clock (RTC)	srss_0_rtc_0	

CSD (CapSense, etc.) 0 (CYBSP_CSD) - Parameters

Name	Value
Peripheral Documentation	
Configuration Help	Open CSD Documentation
Inputs	
Clock	8 bit Divider 3 clk (CYBSP_CSD_CLK_DIV, CYBSP_CS_CLK_DIV) [USED]
CapSense	
Enable CapSense	✓
Target CPU core	Cortex M4
External Tools	
CapSense Configurator	Launch CapSense Configurator
CapSense Tuner	Launch CapSense Tuner
CSDADC	
Enable CSDADC	
CSDIDAC	
Enable CSDIDAC	
CapSense Capacitors	

CSD (CapSense, etc.) 0 (CYBSP_CSD) - Parameters | Code Preview

Notice List

0 Errors | 0 Warnings | 0 Tasks | 6 Infos

Fix	Description	Location

Ready

As described under Tools Make Targets, you can use the `make open` command with appropriate arguments to open any configurator. For example, to open the CapSense Configurator, run:

```
make open CY_OPEN_TYPE=capsense-configurator
```

You can also use the Eclipse IDE provided with ModusToolbox to open configurators. For example, if you select the "Device Configurator" link in the IDE Quick Panel, the tool opens with the application's *design.modus* file. Refer to the Eclipse IDE for ModusToolbox User Guide for more details about the Eclipse IDE.

One other way to open BSP configurators (such as CapSense and SegLCD) is by using a link from inside the Device Configurator. However, this does not apply to Library configurators (such as Bluetooth and USB).

2.5.2 Configurator CLI Tools

Most of the configurators 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. When writing application code, you must decide how the application will interact with the hardware; see Application Layers.

ModusToolbox software is designed to enable your workflow. It includes an integrated Eclipse IDE, as well as support for Visual Studio (VS) Code, IAR Embedded Workbench, and Keil µVision (see Exporting to IDEs). You can also use a text editor and command line tools. 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 with various versions of the `TOOLCHAIN` values listed in the following table. Refer to the release information for each product for specific versions of the toolchains.

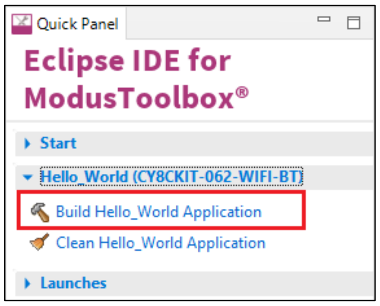
`TOOLCHAIN`	Tools	Host OS
`GCC_ARM`	GNU Arm Embedded Compiler	Mac OS, Windows, Linux
`ARM`	Arm compiler	Windows, Linux
`IAR`	Embedded Workbench	Windows

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 toolchain.

ModusToolbox software installs the GNU Arm toolchain and uses it by default. If you wish to use another toolchain, 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.


Quick Panel
Eclipse IDE for ModusToolbox®
Start
Hello_World (CY8CKIT-062-WIFI-BT)
Build Hello_World Application
Clean Hello_World Application
Launches

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.

If you do change settings in the makefile (for example, `TARGET` or `CONFIG`), you must re-create the launch configs using the link in the Quick Panel; refer to the Eclipse IDE for ModusToolbox User Guide for more details.

2.7.2 Export to another IDE

If you prefer to use an IDE other than Eclipse, you can select the appropriate IDE from the **Target IDE** pull-down menu when creating an application using the Project Creator tool. You can also use the appropriate `make <ide>` command. For example, to export to Visual Studio Code, run:

```
make vscode
```

For more details about using other IDEs, see the Exporting to IDEs chapter. 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

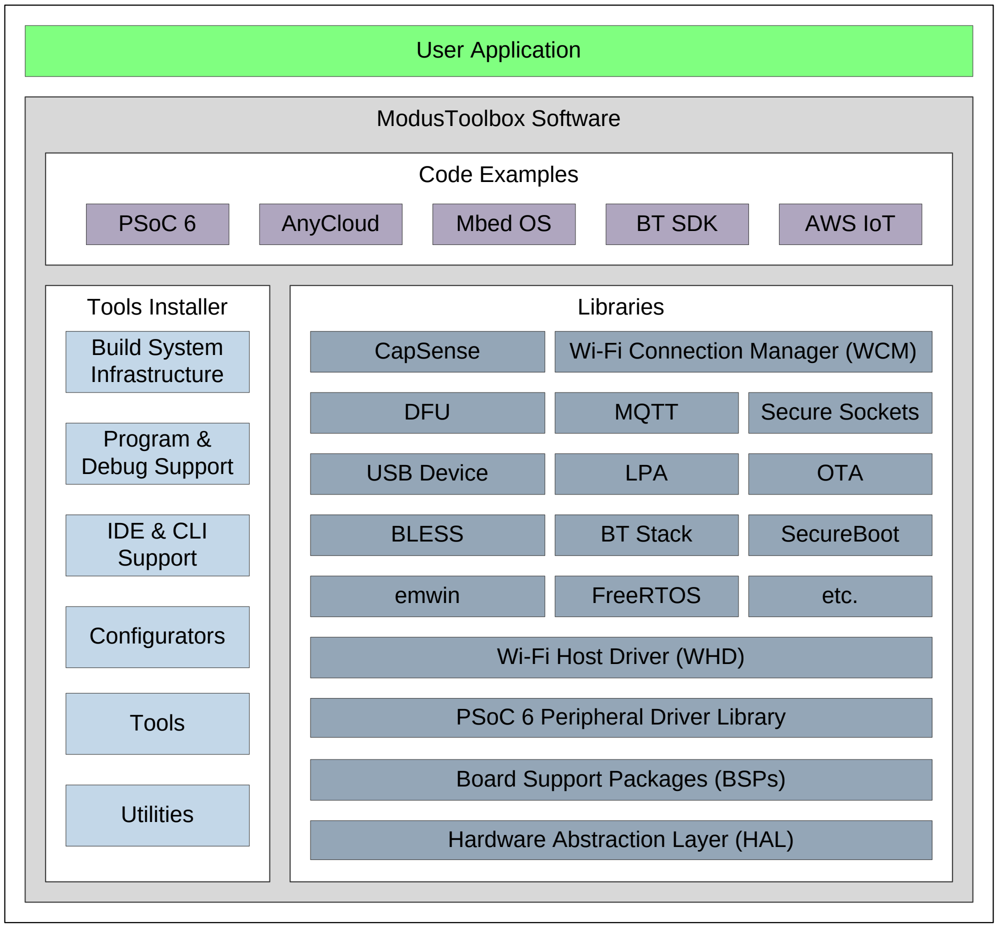
The following commands can be used to debug the application, as follows:

- `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

This chapter provides an overview of the ModusToolbox software environment. As described in the Introduction chapter, ModusToolbox is set of Reference Flows, Products, and Solutions. From a practical standpoint, ModusToolbox is delivered in various ways, such as Installation Resources, Code Examples, and BSPs & Libraries, and you only use the resources you need. When you create applications, you use these resources and interact with the hardware through the Hardware Abstraction Layer (HAL) and/or the Peripheral Driver Library (PDL).

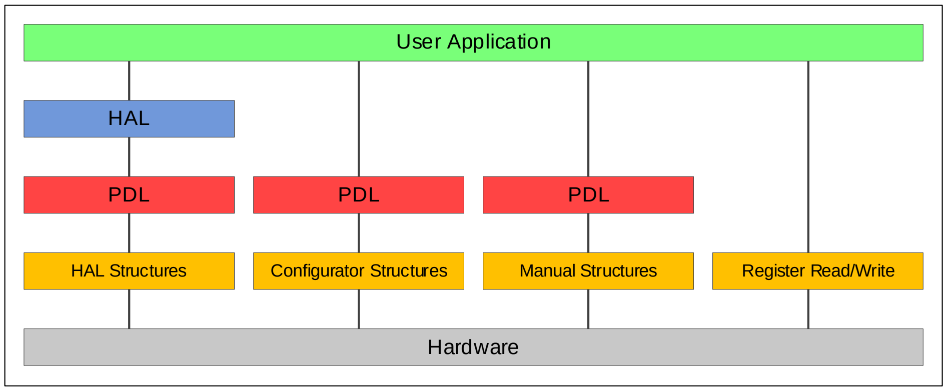
The following block diagram shows a very high-level of the software available in ModusToolbox. This is not a comprehensive list. It merely conveys the idea that there are multiple resources available to you.



Another important aspect of the ModusToolbox software is that each product is versioned. This ensures that each product can be updated on an ongoing basis, but it also allows you to lock down specific versions of the tools for your specific environment. See Product Versioning for more details.

3.1 Application Layers

There are four distinct ways for an application to interact with the hardware as shown in the following diagram:



- **HAL Structures**: Application code uses the HAL, which interacts with the PDL through structures created by the HAL
- **Configurator Structures**: Application code uses PDL through structures created by a Configurator.
- **Manual Structures**: Application code uses PDL through structures created manually.
- **Register Read/Write**: Application code uses direct register read and writes.

Note that a single application may use different methods for different peripherals.

3.1.1 HAL

Using the HAL is more portable than the other methods. It is the preferred method for simpler functions and those that don't have extremely strict flash size limitations. It is a high-level interface to the hardware that allows many common functions to be done quickly and easily. This allows the same code to be used even if there are changes to pin assignments, different devices in the same family, or even to a different family that may have radically different underlying architectures.

The advantages include:

- Easy hardware changes. Just change the pin assignment in the BSP and the code remains the same. For example, if LED1 changes from P0_0 to P0_1, the code remains the same as long as the code uses the name LED1 with the HAL. The only change is to the BSP pin assignment.
- Easy migration to a different device as product requirements change.
- Ability to use the same code base across multiple projects and generations, even if underlying architectures are different.

The disadvantages include:

- The HAL may not support every feature that the hardware has. It supports the most common features but not all of them to maintain simplicity.
- The HAL will use additional flash space. The additional flash depends on which HAL APIs are used.

3.1.2 PDL

The PDL is a lower-level interface to the hardware (but still simpler than direct register access) that supports all hardware features. Usually the PDL goes hand-in-hand with Configurators, which will be described next. Since the PDL interacts with the hardware at a lower level it is less portable between devices, especially those with different architectures.

The advantages/disadvantages are the exact opposite of those for the HAL. The main advantage is that it provides access to every hardware feature.

3.1.3 Configurators

Configurators make initial setup easier for hardware accessed using the PDL. The Configurators create structures that the PDL requires without you needing to know the exact composition of each structure, and they create the proper structure based on your selections. See Configurators for more information.

If you use the HAL for a peripheral, it will create the necessary structures for you, so you should NOT use a Configurator to set them up. The HAL structure is accessible, and once you initialize a peripheral with the HAL you can view and even modify that structure (that is, a HAL object). Keep in mind that the underlying structures are hardware-specific, so you may be sacrificing portability if you modify the structure manually. There are a few exceptions. For example, it is reasonable to configure system items (such as clocks) and use them with the HAL.

3.2 Installation Resources

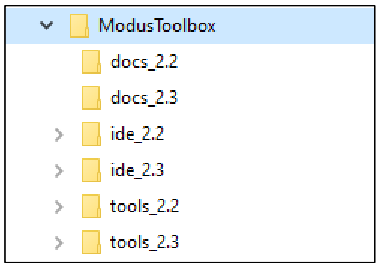
The ModusToolbox tools package installer provides required and optional core resources for any application. This section provides an overview of the available resources:

- Directory Structure
- Documentation
- IDE Support
- Tools

The installer does not include Code Examples or Libraries, but it does provide the tools to access them.

3.2.1 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 2.2 and 2.3 installed. Your installation may only include ModusToolbox version 2.3. Refer to the Product Versioning section for more details.

The ModusToolbox directory contains the following subdirectories for version 2.3:

- **docs_2.3 –** 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.
- **ide_2.3:**
  - **eclipse (or ModusToolbox.app on macOS) –** This contains the optional Eclipse IDE for ModusToolbox. It includes the ModusToolbox perspective, 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. For more details, refer to the Eclipse IDE for ModusToolbox User Guide.
- **tools_2.3:** This contains all the various tools and scripts installed as part of ModusToolbox. See Tools for more information.

3.2.2 Documentation

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

3.2.2.1 Release Notes

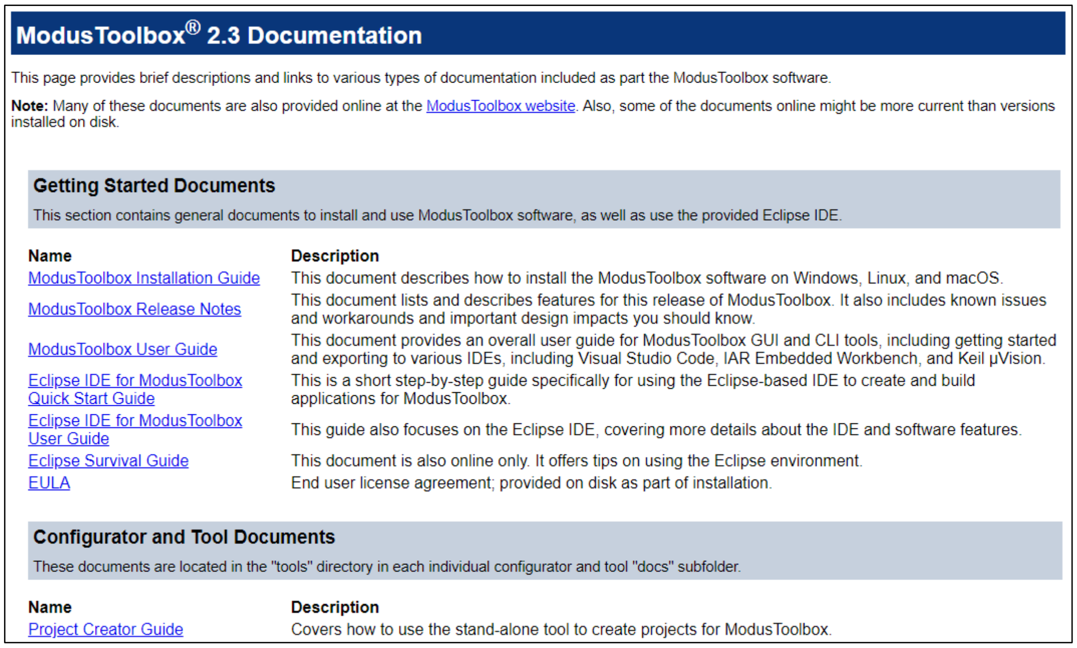
For the 2.3 release, the release notes document is for all of the ModusToolbox software included in the installation.

3.2.2.2 Top-Level Documents

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

3.2.2.3 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® 2.3 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 and use ModusToolbox software, as well as use the provided Eclipse IDE.

Name	Description
ModusToolbox Installation Guide	This document 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.

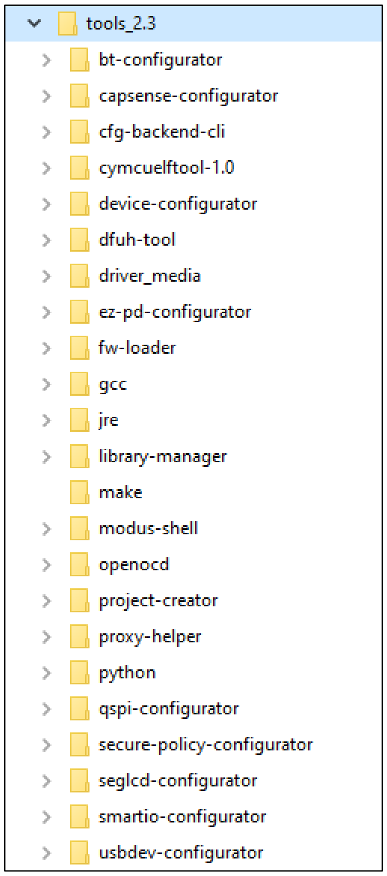
Name	Description
Project Creator Guide	Covers how to use the stand-alone tool to create projects for ModusToolbox.

3.2.3 IDE Support

ModusToolbox includes an optional Eclipse IDE that is a full-featured, cross-platform IDE. ModusToolbox also provides support for Visual Studio (VS) Code, IAR Embedded Workbench, and Keil µVision. See the Exporting to IDEs chapter for more details.

3.2.4 Tools

The *tools_2.3* folder contains the following tools:



tools_2.3
- bt-configurator
- capsense-configurator
- cfg-backend-cli
- cymcuelftool-1.0
- device-configurator
- dfuh-tool
- driver_media
- ez-pd-configurator
- fw-loader
- gcc
- jre
- library-manager
- make
- modus-shell
- openocd
- project-creator
- proxy-helper
- python
- qspi-configurator
- secure-policy-configurator
- seglcd-configurator
- smartio-configurator
- usbdev-configurator

3.2.4.1 Configurators

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.

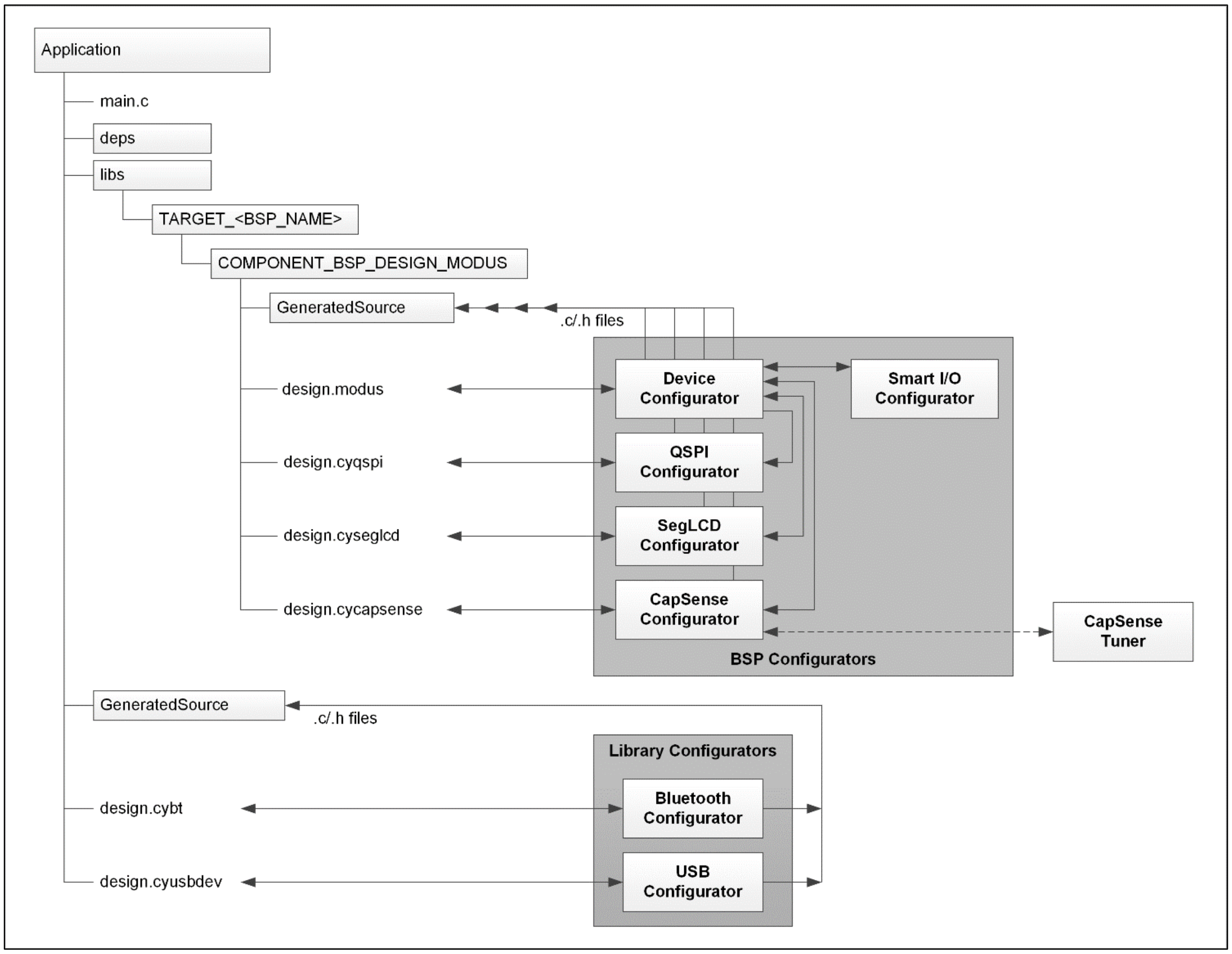
Configurators perform tasks such as:

- 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. Some of the 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 on the board connected to your computer. 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 Chip, I/O, Data Unit, and LUT signals between port pins and the HSIOM. 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. Some of the 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.

3.2.4.2 Other Tools

ModusToolbox software includes other tools that provide support for application creation, device firmware updates, and so on. All tools are installed by the ModusToolbox 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
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

3.2.4.3 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.
JRE	Java Runtime Environment; required by the Eclipse IDE integration layer.

3.2.4.4 Build System Infrastructure

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

- create an application, update and clone dependencies
- create an executable
- provide debug capabilities

A makefile defines everything required for your application, including:

- target hardware (board/board support package to use)
- source code and libraries to use for the application
- ModusToolbox tools version, as well as compiler toolchain to use
- compiler/assembler/linker flags to control the build
- assorted variables to define things like file and directory locations

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 *.mtb* file. This file contains the URL to a git repository, a commit tag, and a variable for where to put the library on disk. For example, a *capsense.mtb* file might contain the following line:

```
http://github.com/cypresssemiconductorco/capsense#latest-v2.X#$$ASSET_REPO$$/capsense/latest-v2.X
```

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

3.2.4.5 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.

You can use various IDEs to program devices and establish a debug session (see Exporting to IDEs). 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 update KitProg3 firmware, if you need to do so. The fw-loader tool is installed with the ModusToolbox software. The latest version of the tool 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 needed.	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 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, PSoC 6 MCU, PSoC 4, among others. For most code examples, you can use git clone or the Project Creator tool to create an application and use it directly with ModusToolbox tools. For some examples, like Mbed OS, you will need to 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.

In the ModusToolbox build infrastructure, any example application that requires a library downloads that library automatically.

You can control the versions of the libraries being downloaded and also their location on disk, and whether they are shared or local to the application. Refer to the Library Manager Guide for more details.

3.4 BSPs & Libraries

In addition to the installer and code examples, there are many other parts of ModusToolbox that are provided as libraries. These libraries are essential for taking full advantage of the various features of the various devices. When you create a ModusToolbox application, the system downloads all the libraries your application needs. See ModusToolbox Build System chapter to understand how all this works.

3.4.1 Board Support Packages

The Board Support Package (BSP) is a central feature of ModusToolbox software. The BSP specifies several critical items for the application, including:

- hardware configuration files for the device (for example, *design.modus*)
- startup code and linker files for the device
- other libraries that are required to support a kit

BSPs are aligned with our development/evaluation 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. You can modify the configuration to suit your application.

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_*kitnumber*. 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
- the BSP Overview
- 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 Board Support Packages chapter for how they work and how to create your own for a custom board.

3.5 Libraries

All current ModusToolbox libraries can be found through the GitHub ModusToolbox Software page. A ModusToolbox application can use different libraries based on the Active BSP. In general, there are several categories of libraries:

- **Common Library Types**: Most BSPs have some form of the following types of libraries:
  - Abstraction Layers – This is usually the RTOS Abstraction Layer.
  - Base Libraries – These are core libraries, such as core-lib and core-make.
  - Board Utilities – These are board-specific utilities, such as RGB LED support or BTSpy.
  - MCU Middleware – These include MCU-specific libraries such as freeRTOS or Clib support.
- **AIROC Bluetooth Libraries**: For the AIROC Bluetooth BSPs, there specific libraries that do not apply to any other BSPs, including:
  - BTSDK Chip Libraries
  - BTSDK Core Support
  - BTSDK Shared Source Libraries
  - BTSDK Utilities and Host/Peer Apps

- **BSP-Specific Base Libraries**: BSP-specific libraries include mtb-hal, mtb-pdl, and recipe-make. Some of these are identified as device-specific using the following categories:
  - cat1/cat1a = PSoC 6 (mtb-hal-cat1, recipe-make-cat1a, etc.)
  - cat2 = PSoC 4 and XMC (mtb-hal-cat2, mtb-pdl-cat2)
  - cat3 = XMC (recipe-make-cat3)
- **PSoC 6 Additional Libraries**: Due to the nature of the PSoC 6 MCU, plus the combo devices, certain PSoC 6 BSPs have additional libraries, including:
  - BT Middleware Libraries – These are for the BTStack and Bluetooth FreeRTOS.
  - PSoC 6 Middleware – These are libraries specific to the PSoC 6 MCU, such as EMEEPROM and DFU.
  - Wi-Fi Middleware Libraries – These are libraries for AnyCloud applications for enabling Wi-Fi and Bluetooth on a PSoC 6 hosted CYW43xxx device.

Each library is delivered in its own repository, complete with documentation. The following information includes links to each repository, and various resources associated with each library. Go to each repository to see what releases are available for that library.

3.6 Product Versioning

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 every tool and library may have more than one version. The tools package that provides the set of tools also has its own version. This section describes how ModusToolbox products are versioned.

3.6.1 General Philosophy

ModusToolbox is not a monolithic entity. 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. We deliver libraries via GitHub, and we deliver tools though 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 minor defects. They are very low-risk (fix the essential defect without unnecessary complexity).

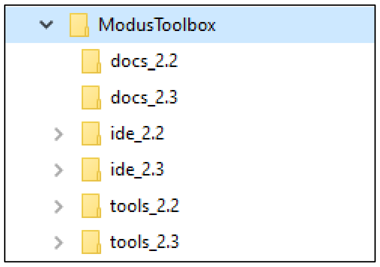
Code Examples include various libraries automatically. Prior to the ModusToolbox 2.3 release, these libraries were typically the latest versions. As of the 2.3 release, when you create a new application from a code example, any of the included libraries specified with a “latest-style” tag are converted to the “release-vX.Y.Z” style tag.

If you use the Library Manager to add a library to your project, the tool automatically finds and adds any required dependent libraries. As of the 2.3 release using the MTB flow, these dependencies are created using “release-vX.Y.Z” style tags. The tool also creates and updates a file named *locking_commit.log* in the deps subdirectory inside your application directory. This file maintains a history of all latest to release conversions made to ensure consistency with any libraries added in the future.

3.6.2 Tools Package Versioning

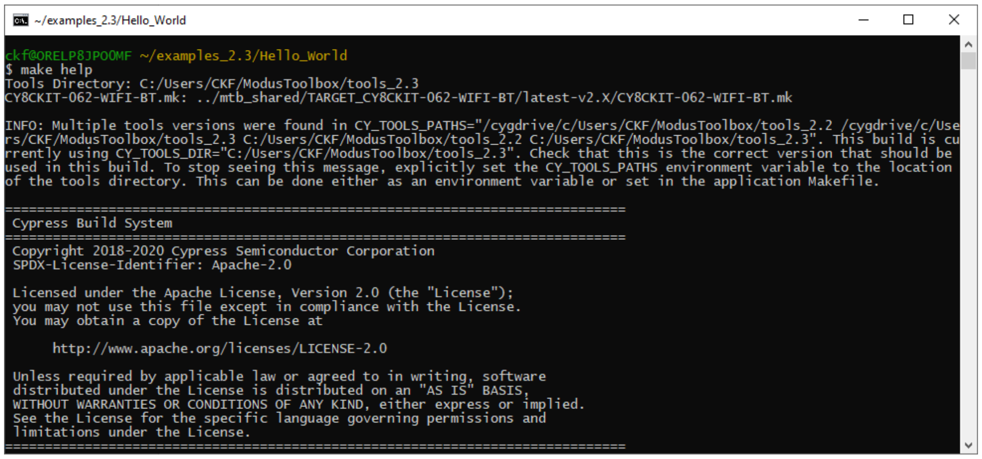
The ModusToolbox tools installation package is versioned as MAJOR.MINOR.PATCH. The file located at *<install_path>/ModusToolbox/tools_2.3/version-2.3.0.xml* also indicates the build number.

Every MAJOR.MINOR version of ModusToolbox products is 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:



3.6.3 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.



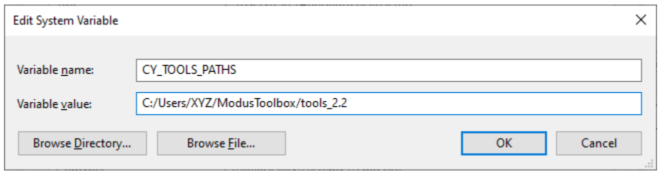
```
~/examples_2.3/Hello_World

ckf@ORELP8JPOOMF ~/examples_2.3/Hello_World
$ make help
Tools Directory: C:/Users/CKF/ModusToolbox/tools_2.3
CY8CKIT-062-WIFI-BT.mk: ../mtb_shared/TARGET_CY8CKIT-062-WIFI-BT/latest-v2.X/CY8CKIT-062-WIFI-BT.mk

INFO: Multiple tools versions were found in CY_TOOLS_PATHS="/cygdrive/c/Users/CKF/ModusToolbox/tools_2.2 /cygdrive/c/Use
rs/CKF/ModusToolbox/tools_2.3 C:/Users/CKF/ModusToolbox/tools_2.2 C:/Users/CKF/ModusToolbox/tools_2.3". This build is cu
rrently using CY_TOOLS_DIR="C:/Users/CKF/ModusToolbox/tools_2.3". Check that this is the correct version that should be
used in this build. To stop seeing this message, explicitly set the CY_TOOLS_PATHS environment variable to the location
of the tools directory. This can be done either as an environment variable or set in the application Makefile.

==============================================================================
 Cypress Build System
==============================================================================
 Copyright 2018-2020 Cypress Semiconductor Corporation
 SPDX-License-Identifier: Apache-2.0

 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License.
==============================================================================
```

3.6.4 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 2.2 and 2.3 installed, and if you launch the Eclipse IDE from the ModusToolbox 2.2 installation, the IDE will use the tools from the "tools_2.3" directory to launch configurators and build an application. This section describes how to specify the path to the desired version.

3.6.4.1 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:


Edit System Variable

Variable name: CY_TOOLS_PATHS

Variable value: C:/Users/XYZ/ModusToolbox/tools_2.2

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

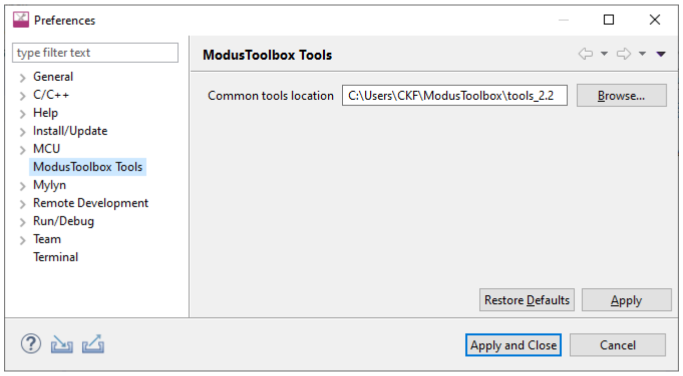
C:/Users/XYZ/ModusToolbox/tools_2.2

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

3.6.4.2 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**.


Preferences

ModusToolbox Tools

Common tools location: C:\Users\CKF\ModusToolbox\tools_2.2

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

Note This will generate messages in the IDE console indicating that is using the appropriate tools path.

3.6.4.3 Specific Project Makefile

To preserve a specific "tools" path for the specific project, edit that project's makefile, as follows:

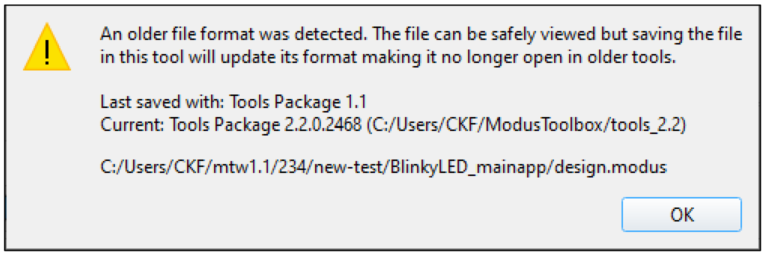
```
# If you install the IDE in a custom location, add the path to its
# "tools_X.Y" folder (where X and Y are the version number of the tools
# folder).
CY_TOOLS_PATHS+=C:/Users/XYZ/ModusToolbox/tools_2.2
```

3.6.5 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.

3.6.5.1 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:

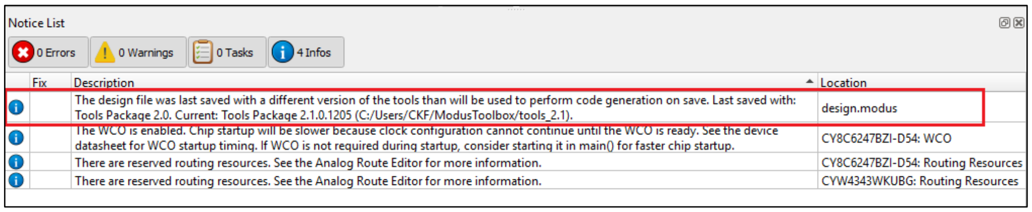


An older file format was detected. The file can be safely viewed but saving the file in this tool will update its format making it no longer open in older tools.

Last saved with: Tools Package 1.1
Current: Tools Package 2.2.0.2468 (C:/Users/CKF/ModusToolbox/tools_2.2)

C:/Users/CKF/mtw1.1/234/new-test/BlinkyLED_mainapp/design.modus

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



	Fix	Description	Location
●		The design file was last saved with a different version of the tools than will be used to perform code generation on save. Last saved with: Tools Package 2.0. Current: Tools Package 2.1.0.1205 (C:/Users/CKF/ModusToolbox/tools_2.1).	design.modus
●		The WCO is enabled. Chip startup will be slower because clock configuration cannot continue until the WCO is ready. See the device datasheet for WCO startup timing. If WCO is not required during startup, consider starting it in main() for faster chip startup.	CY8C6247BZI-D54: WCO
●		There are reserved routing resources. See the Analog Route Editor for more information.	CY8C6247BZI-D54: Routing Resources
●		There are reserved routing resources. See the Analog Route Editor for more information.	CYW4343WKUBG: Routing Resources

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

3.6.6 GitHub Libraries Versioning

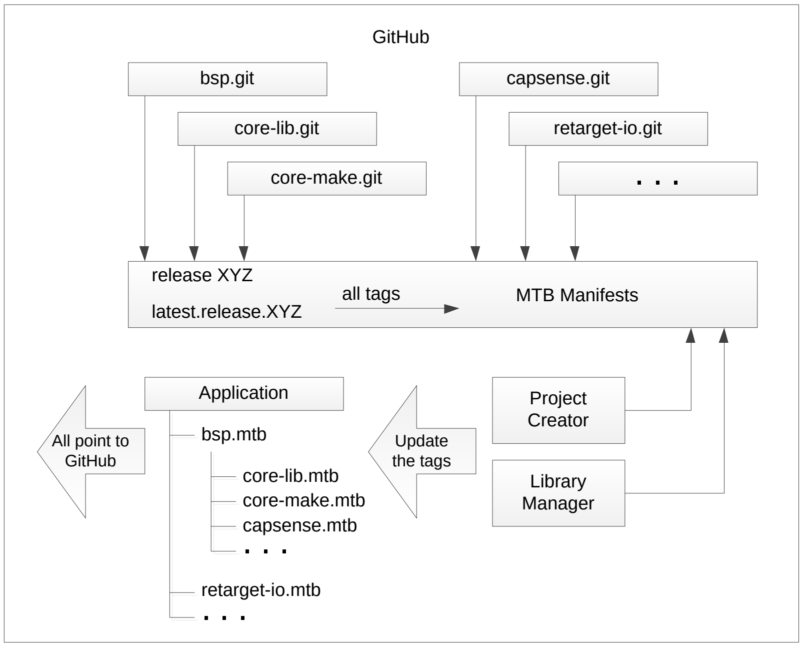
GitHub libraries follow the same versioning scheme: MAJOR.MINOR.PATCH. The GitHub libraries, besides the code itself, also provide two files in MD format: README and RELEASE. The latter includes the version and the change history.

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 tool parses the manifests to determine which BSPs and applications are available to select. The Library Manager tool parses the manifests and allow you to see and select between various tags of these libraries. When selecting a particular library of a particular version, the *.mtb* file gets created in your project. These *.mtb* 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 *.mtb* files, and will deliver the latest code changes for the major version.

3.6.7 Dependencies Between Libraries

The following diagram shows the dependencies between libraries.



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

3.6.7.1 Dependencies via .mtb files

Dependencies for various libraries are specified in the manifest file. Only the top-level application will have .mtb files for the libraries it directly includes.

3.6.7.2 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 #error 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.

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:

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. There are two separate repos: core-make used by all recipes and a recipe-make-xxx that contains BSP/target specific details. These are the minimum required 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 application – 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 application – 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 application build. A library application is constructed by defining the LIBNAME variable in the application makefile.

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

```
DEPENDENT_LIB_PATHS=../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.

- When using the Project Creator to create an application, it provides the selected BSP and updates the makefile.
- Use the Library Manager to add, update, or remove a BSP from an application. You can also add a *.mtb* 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 *.mtb* files in the application directory and performs `git clone` operations on them. A *.mtb* file contains a git URL to a library repo, a specific tag for a version of the code, and a variable to specify the location to store the library.

The `getlibs` target finds and processes all *.mtb* files and uses the `git` command to clone or pull the code as appropriate. The target also calls the library-manager-cli tool to generate *.mtb* files for indirect dependencies. Then, it checks out the specific tag listed in the *.mtb* 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 *.mtb* 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`. See also KBA231252.

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 Project Creator and Library Manager tools run the `make getlibs` command and generate a *mtb.mk* file in the application's *libs* subdirectory. This file specifies the location of shared libraries included in the build.

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:

- *.mtb* 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 *.cyignore* file can contain make variables. For example, you can use the `SEARCH_` variable to exclude code from other libraries as shown for the "Test" directory in a library called <library-name>:

```
$(SEARCH_<library-name>/Test
```

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. ModusToolbox supports IAR, ARM, and GCC_ARM.

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.

4.5.1.7 Multi-project application with imported BSP

When you use an imported BSP to create a multi-project application, the system copies the BSP into an application root folder. For these types of applications, the Project Creator tool creates an importedbsp.mk file for each project with a SEARCH variable and relative path to the imported BSP. For example:

```
SEARCH+=<relative_path_to_BSP_folder>
```

If you do not use the Project Creator tool, you must create the files manually in each project directory.

In addition, when `make getlibs` is run, it updates the *mtb.mk* file with the following line:

```
-include ${CY_INTERNAL_APP_PATH}/importedbsp.mk
```

The "-" in front of "include" tells the make system to perform a conditional include. It only includes the file if it exists. If the file doesn't exist, the system does not issue a warning.

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.

Target	Description
`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.
`attach`	Starts a GDB client and attaches the debugger to the running target.
`clean`	Cleans the */build/<TARGET>* directory. The directory and all its contents are deleted from disk.
`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 and project files. 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. **Note** Project generation requires Python 3 to be installed and present in the PATH variable. **Note** To skip project creation and only create the launch configs, set `CY_MAKE_IDE=eclipse`.
`vscode`	Generates VS Code IDE json files. 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. This target requires to have `TOOLCHAIN=IAR` set . It 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 Python 3 to be installed and present in the PATH variable.
`uvision5`	Generates Keil µVision v5 IDE .cpdsc, .gpdsc, and .cprj files. This target requires to have `TOOLCHAIN=ARM` set. It 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 Python 3 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. Supported types are: `bt-configurator capsense-configurator capsense-tuner device-configurator dfuh-tool library-manager project-creator qspi-configurator seglcd-configurator smartio-configurator usbdev-configurator`.
`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.
`config_secure`	Runs the Secure Policy Configurator. This configurator is intended only for devices that support secure provisioning.

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-S2D44 ADDITIONAL_DEVICES_GEN=CYW4343WKUBG`
`update_bsp`	Change the device in a custom BSP generated by the `make bsp` command. This target changes the device set in a custom Board Support Package generated by the `make bsp` command. The `TARGET_GEN` variable will specify the Board Support Package to modify. The `DEVICE_GEN` variable will specify the new target device of the BSP. For example: `make update_bsp TARGET_GEN=NewBoard DEVICE_GEN=CY8C624ABZI-S2D44`
`lib2mtbx`	Convert *.lib* files to *.mtbx* files This will recursively look for *.lib* files in `CONVERSION_PATH` and create equivalent *.mtbx* files adjacent to them. The type of *.mtbx* file is determined by the `CONVERSION_TYPE` variable. This can be either [local] or [shared]. The default is [local].
`import_deps`	Import dependent *.mtbx* files of a given path into the application. This will recursively look for *.mtbx* files in `IMPORT_PATH`, copy them to the application's deps directory and rename them to *.mtb* files. Note that the import process is not applicable for applications using *.lib* files. These libraries must instead be situated in the application directory.
`get_app_info`	Prints the application info for the Eclipse IDE for ModusToolbox. 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"`

Target	Description
`get_env_info`	Prints the make, git, and, application repo info. This allows a quick printout of the current application 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).
`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.

4.9 Available Make Variables

The following variables customize various make targets. They can be defined in the application makefile or passed through the make invocation. The following sections group the variables for how they can be used.

4.9.1 Basic Configuration Make Variables

These variables define basic aspects of building an application. For example:

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

Variable	Description
`TARGET`	Specifies the target board/kit (that is, BSP). For example, CY8CPROTO-062-4343W. Example usage: `make build TARGET=CY8CPROTO-062-4343W`
`APPNAME`	Specifies the name of the application. For example, "AppV1" > AppV1.elf. Example usage: `make build APPNAME="AppV1"` This variable is used to set the name of the application artifact (programmable image). It also signifies that the application will build for a programmable image artifact that is intended for a target board. For applications that need to build to 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" > LibV1.a. Example Usage: `make build LIBNAME="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 `DEPENDENT_LIB_PATHS` variable.
`TOOLCHAIN`	Specifies the toolchain used to build the application. For example, GCC_ARM. Example Usage: `make build TOOLCHAIN=IAR CY_COMPILER_PATH="<path>/IAR Systems/Embedded Workbench 8.4/arm/bin"` Supported toolchains for this include GCC_ARM, IAR, and ARM.
`CONFIG`	Specifies the configuration option for the build [Debug Release]. Example Usage: `make build CONFIG=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]. Example Usage: `make build VERBOSE=true` Setting `VERBOSE` to true may help in debugging build errors/warnings. By default, it is set to false.

4.9.2 Advanced Configuration Make Variables

These variables define advanced aspects of building an application.

Variable	Description
`SOURCES`	Specifies C/C++ and assembly files outside of application directory. Example Usage (within makefile): `SOURCES+=path/to/file/Source1.c` This can be used to include files external to the application directory. The path can be both absolute or relative to the application directory.
`INCLUDES`	Specifies include paths outside of the application directory. Example Usage (within makefile): `INCLUDES+=path/to/headers` **Note** These MUST NOT have `-I` prepended. The path can be either absolute or relative to the application directory.
`DEFINES`	Specifies additional defines passed to the compiler. Example Usage (within makefile): `DEFINES+=EXAMPLE_DEFINE=0x01` **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. Example Usage (within makefile): `VFP_SELECT=hardfp`
`CFLAGS`	Prepends additional C compiler flags. Example Usage (within makefile): `CFLAGS+= -Werror -Wall -O2` **Note** If the entire C compiler flags list needs to be replaced, define the `CY_RECIPE_CFLAGS` make variable with the desired C flags. The values should be space separated.
`CXXFLAGS`	Prepends additional C++ compiler flags. Example Usage (within makefile): `CXXFLAGS+= -finline-functions` **Note** If the entire C++ compiler flags list needs to be replaced, define the `CY_RECIPE_CXXFLAGS` make variable with the desired C++ flags. Usage is similar to `CFLAGS`.
`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. Usage is similar to `CFLAGS`.
`LDFLAGS`	Prepends additional linker flags. Example Usage (within makefile): `LDFLAGS+= -nodefaultlibs` **Note** If the entire linker flags list needs to be replaced, define the `CY_RECIPE_LDFLAGS` make variable with the desired linker flags. Usage is similar to `CFLAGS`.
`LDLIBS`	Includes application-specific prebuilt libraries. Example Usage (within makefile): `LDLIBS+=../MyBinaryFolder/binary.o` **Note** If additional libraries need to be added using `-l` or `-L`, add to the `CY_RECIPE_EXTRA_LIBS` make variable. Usage is similar to `CFLAGS`.
`LINKER_SCRIPT`	Specifies a custom linker script location. Example Usage (within makefile): `LINKER_SCRIPT=path/to/file/Custom_Linker1.ld` 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. Example Usage (within makefile): `PREBUILD=python example_script.py` **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. Example Usage (within makefile): `POSTBUILD=python example_script.py` **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.

Variable	Description
COMPONENTS	Adds component-specific files to the build. Example Usage (within makefile): `COMPONENTS+=CUSTOM_CONFIGURATION` 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. Example Usage (within makefile): `DISABLE_COMPONENTS=BSP_DESIGN_MODUS` 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` as shown.

DEPENDENT_LIB_PATHS	List of dependent library application paths. For example, *../bspLib*.
	Note This variable replaces the `SEARCH_LIBS_AND_INCLUDES` variable.
	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`

DEPENDENT_APP_PATHS	List of dependent application paths. For example, *../cy_m0p_image*. The main application can have a dependency on other artifact producing applications (defined by setting `APPNAME`). This variable defines those dependencies for the main application. The artifacts of these dependent applications are translated to c-arrays and are brought into the main application as regular c-files and are compiled and linked. The main application also generates a "standalone" variant of the main application that does not include the dependent applications.
SEARCH	List of paths to include in auto-discovery. For example, *../mtb_shared/lib1*. When `getlibs` is run for applications that use *.mtb* files, a file is generated in *./libs/mtb.mk*. This file automatically populates the `SEARCH` variable with the locations of the libraries in the shared repo location (set by the `CY_GETLIBS_SEARCH_PATH` and `CY_GETLIBS_SHARED_NAME` variables). The `SEARCH` variable can also be used by the application to include other directories to auto-discovery.
IMPORT_PATH	Path to *.mtbx* dependency files to import into the application. This variable must be defined when calling `import_deps`. Any *.mtbx* dependency file found in this directory will be imported into the application. **Note** This is not applicable for applications using *.lib* files.
CONVERSION_PATH	Path to the *.lib* files to convert to *.mtbx* files. This variable must be defined when calling `lib2mtbx`. Any *.lib* file found in this directory will be converted.
CONVERSION_TYPE	(optional) Defines the type of *.mtbx* file to create. This variable can be set to [local] or [shared]. The default type is [local]. If [local], the library will be deposited in the application's `CY_GETLIBS_PATH` directory when performing `getlibs`. If [shared], the library will be deposited (when running `getlibs`) in the shared location as defined by `CY_GETLIBS_SHARED_PATH` and `CY_GETLIBS_SHARED_NAME`.
FORCE	Optional) Force overwrite existing files. When this variable is set [true], `lib2mtbx` overwrites any existing *.mtbx* files.

4.9.3 BSP Make Variables

These variables are used with the `make bsp` target.

Variable	Description
`DEVICE`	Device ID for the primary MCU on the target board/kit (set by *TARGET.mk*). The device identifier is mandatory for all board/kits.
`TARGET_GEN`	Name of the new target board/kit. Example Usage: `make bsp TARGET_GEN=MyBSP` 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. Example Usage: `make bsp TARGET_GEN=MyBSP DEVICE_GEN=CY8C624ABZI-S2D44` 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.

4.9.4 Getlibs Make Variables

These variables are used with the `make getlibs` target.

Variable	**Description**
`CY_GETLIBS_NO_CACHE`	Disables the cache when running `getlibs`. Example Usage: `make getlibs CY_GETLIBS_NO_CACHE=true` 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 *.mtb* files just like the Cypress libraries.
`CY_GETLIBS_OFFLINE`	Use the offline location as the library source. Example Usage: `make getlibs CY_GETLIBS_OFFLINE=true` 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. Example Usage: `make getlibs CY_GETLIBS_PATH="path/to/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 *.mtb* and *.lib* files. Usage is similar to `CY_GETLIBS_PATH`. Setting this path allows relocating the directory that the library-manager uses to store the *.mtb* / *.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.
`CY_GETLIBS_CACHE_PATH`	Absolute path to the cache directory. Usage is similar to `CY_GETLIBS_PATH`. 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. Usage is similar to `CY_GETLIBS_PATH`. 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. Usage is similar to `CY_GETLIBS_PATH`. 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.

Variable	Description
`CY_GETLIBS_SHARED_PATH`	Relative path to the shared repo location. All *.mtb* files have the format, `<URI><COMMIT><LOCATION>`. If the `<LOCATION>` field begins with `$$ASSET_REPO$$`, then the repo is deposited in the path specified by the `CY_GETLIBS_SHARED_PATH` variable. The default location is one directory level above the current application directory (Default: *../*). This is used with `CY_GETLIBS_SHARED_NAME` variable, which specifies the directory name.
`CY_GETLIBS_SHARED_NAME`	Directory name of the shared repo location. All *.mtb* files have the format, `<URI><COMMIT><LOCATION>`. If the `<LOCATION>` field begins with `$$ASSET_REPO$$`, then the repo is deposited in the directory specified by the `CY_GETLIBS_SHARED_NAME` variable. The default directory name is “mtb_shared”. This is used with `CY_GETLIBS_SHARED_PATH` variable, which specifies the directory path.

4.9.5 Path Make Variables

These variables are used to specify various paths.

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 application 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/recipe-make-cat1a* 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 recipe base library. For applications using *.mtb* files, the BSP's *TARGET.mk* file sets this variable and therefore the application does not need to.
`CY_BASELIB_CORE_PATH`	Relative path to the core base library. For example, *./libs/core-make* This directory must be relative to `CY_APP_PATH`. It defines the location of the library containing the core make files, where the expected directory structure is *<CY_BASELIB_CORE_PATH>/make*. All applications must set the location of the core base library. For applications using *.mtb* files, the BSP's *TARGET.mk* file sets this variable and therefore the application does not need to. This variable is not applicable for applications using the combined base library (such as recipe-make-cat1a).
`CY_EXTAPP_PATH`	Relative path to an external application 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 `DEPENDENT_LIB_PATHS` for tools and IDE support. **Note** The same functionality exists in the `SEARCH` variable. Using the `SEARCH` variable is preferred over `CY_EXTAPP_PATH`.
`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 8.4/arm/"`
`CY_TOOLS_DIR`	Absolute path to the tools root directory. Example Usage: `make build CY_TOOLS_DIR="path/to/ModusToolbox/tools_x.y"` 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 a specific Python executable. Example Usage: `CY_PYTHON_PATH="path/to/python/executable/python.exe"` For make targets that depend on Python, the build system looks for Python 3 in the user's PATH and uses that to invoke python. If you have a version of Python installed in a non-default location and do not have a path set for it, you can set `CY_PYTHON_PATH` as a System Variable. In Windows, you must use forward slashes in the path to the Python executable.
`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-application scenario.
`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

These are miscellaneous variables used for various make targets.

Variable	Description
`CY_IGNORE`	Adds to the directory and file ignore list. For example, *./file1.c ./inc1* Example Usage: `make build CY_IGNORE="path/to/file/ignore_file"` 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. Setting this to [true/1] allows the application to not include any recipe makefiles and only include the *start.mk* file from the tools install.
`CY_SKIP_CDB`	Skip creating *.cdb* files. Constant Database (CDB) files are generated during the build process. Setting this to [true] allows the build process to skip the *.cdb* files creation.
`CY_EXTRA_INCLUDES`	Specifies additional makefiles to add to the build. Example Usage: `make build CY_EXTRA_INCLUDES="./custom1.mk"` This variable provides a way of injecting additional make files into the core make files. It can be used when including the make file before or after *start.mk* in the application makefile is not possible.
`CY_LIBS_SEARCH_DEPTH`	Directory search depth for *.mtb* files (default: 5). Example Usage: `make getlibs CY_LIBS_SEARCH_DEPTH=7` This variable controls how deep the search mechanism in `getlibs` looks for *.mtb* files. **Note** Deeper searches take longer to process.
`CY_UTILS_SEARCH_DEPTH`	Directory search depth for *.cyignore* and *TARGET.mk* files (default: 5). Example Usage: `make getlibs CY_UTILS_SEARCH_DEPTH=7` 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. Example Usage: `make eclipse CY_IDE_PRJNAME="AppV1"` 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. Example Usage: `make get_app_info CY_CONFIG_FILE_EXT="modus cybt cyusbdev"` This variable accepts a space-separated list of configurator file extensions to search when running the `get_app_info` target.

Variable	Description
`CY_SUPPORTED_TOOL_TYPES`	Defines the supported tools for a BSP. Example Usage (bsp.mk): `CY_SUPPORTED_TOOL_TYPES+=seglcd-configurator` BSPs can define the supported tools that can be launched using the open target. The supported tool types are `bt-configurator, capsense-configurator, capsense-tuner, device-configurator, dfuh-tool, library-manager, project-creator, qspi-configurator, seglcd-configurator, smartio-configurator`, and `usbdev-configurator`. The BSP can make adjustments to the default recipe if needed.

5 Board Support Packages

5.1 Overview

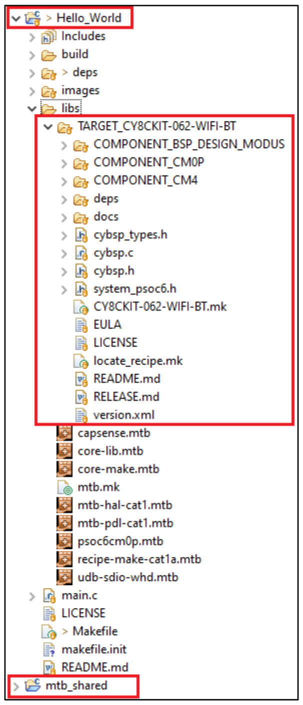
A BSP provides a standard interface to a board's features and capabilities. The API is consistent across Cypress kits. Other software (such as middleware or an application) can use the BSP to configure and control the hardware. BSPs do the following:

- 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.

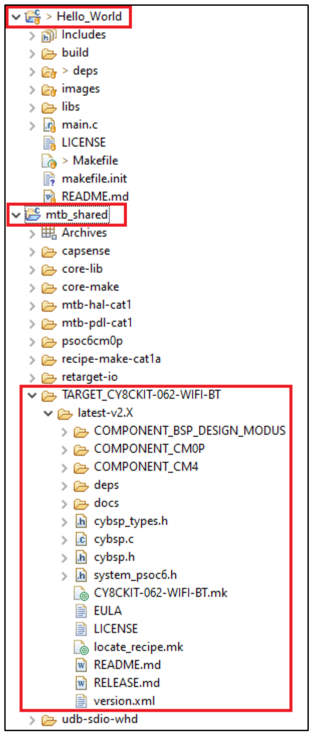
5.2 What’s in a BSP

This section presents an overview of the key resources that are part of a BSP. Using the MTB flow, applications can share BSPs and libraries. BSPs that are local to the application are located in the *libs* subdirectory, while shared BSPs are located in the *mtb_shared* directory adjacent to the application directory. For more details about library management, refer to the Library Manager User Guide.

The following shows a typical PSoC 6 BSP located in the application's *libs* subdirectory on the left. It also shows a shared BSP located in the *mtb_shared* directory on the right.

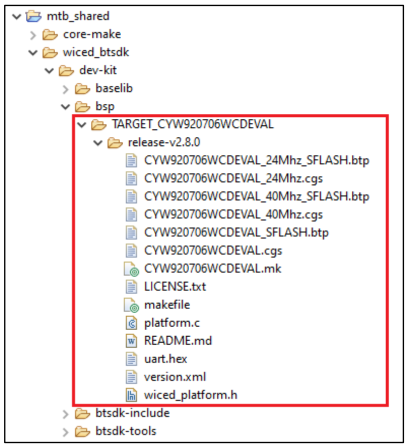


```
> Hello_World
  > Includes
  > build
  > > deps
  > images
  v libs
    v TARGET_CY8CKIT-062-WIFI-BT
      > COMPONENT_BSP_DESIGN_MODUS
      > COMPONENT_CM0P
      > COMPONENT_CM4
      > deps
      > docs
      > cybsp_types.h
      > cybsp.c
      > cybsp.h
      > system_psoc6.h
        CY8CKIT-062-WIFI-BT.mk
        EULA
        LICENSE
        locate_recipe.mk
        README.md
        RELEASE.md
        version.xml
      capsense.mtb
      core-lib.mtb
      core-make.mtb
      mtb.mk
      mtb-hal-cat1.mtb
      mtb-pdl-cat1.mtb
      psoc6cm0p.mtb
      recipe-make-cat1a.mtb
      udb-sdio-whd.mtb
  > main.c
    LICENSE
    > Makefile
    makefile.init
    README.md
> mtb_shared
```



```
v > Hello_World
  > Includes
  > build
  > > deps
  > images
  > libs
  > main.c
    LICENSE
    > Makefile
    makefile.init
    README.md
v mtb_shared
  > Archives
  > capsense
  > core-lib
  > core-make
  > mtb-hal-cat1
  > mtb-pdl-cat1
  > psoc6cm0p
  > recipe-make-cat1a
  > retarget-io
  v TARGET_CY8CKIT-062-WIFI-BT
    v latest-v2.X
      > COMPONENT_BSP_DESIGN_MODUS
      > COMPONENT_CM0P
      > COMPONENT_CM4
      > deps
      > docs
      > cybsp_types.h
      > cybsp.c
      > cybsp.h
      > system_psoc6.h
        CY8CKIT-062-WIFI-BT.mk
        EULA
        LICENSE
        locate_recipe.mk
        README.md
        RELEASE.md
        version.xml
  > udb-sdio-whd
```

Note For BTSDK v2.8 and later, shared BSPs and some shared libraries are located in subdirectories in the *mtb_shared* directory. For example:



```
v mtb_shared
  > core-make
  v wiced_btsdk
    v dev-kit
      > baselib
      v bsp
        v TARGET_CYW920706WCDEVAL
          v release-v2.8.0
              CYW920706WCDEVAL_24Mhz_SFLASH.btp
              CYW920706WCDEVAL_24Mhz.cgs
              CYW920706WCDEVAL_40Mhz_SFLASH.btp
              CYW920706WCDEVAL_40Mhz.cgs
              CYW920706WCDEVAL_SFLASH.btp
              CYW920706WCDEVAL.cgs
              CYW920706WCDEVAL.mk
              LICENSE.txt
              makefile
              platform.c
              README.md
              uart.hex
              version.xml
              wiced_platform.h
      > btsdk-include
      > btsdk-tools
```

For BTSDK v2.7 and earlier, shared BSPs and libraries can be found in the same structure, but without the leading *mtb_shared* directory as shown in the previous image.

The following sections describe the various files and directories in a typical BSP:

5.2.1 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 Modifying the BSP Configuration for a Single Application to learn how the application can override this component.

5.2.2 COMPONENT

Some applications may have additional "COMPONENT" subdirectories. These directories are conditional, based on what the BSP is being built for. For example, the PSoC 6 BSPs include COMPONENT directories to restrict which files are used when building for the Arm Cortex M4 or M0+ core.

5.2.3 deps Subdirectory

The *deps* subdirectory inside the BSP contains *.lib* files from earlier versions of ModusToolbox. This is not the same as the *deps* subdirectory inside the application that contains the *.mtb* files. See Typical Application Contents for more details.

5.2.4 docs Subdirectory

The docs subdirectory contains the documentation in HTML format for the selected BSP.

5.2.5 Support Files

Different BSPs will contain various files, such as the API interface to the board's resources. For example, a typical PSoC 6 BSP contains the following:

- *cybsp.c /.h* – 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.
- *cybsp_types.h* – This currently contains Doxygen comments. It is intended to contain the aliases (macro definitions) for all the board resources, as needed.
- *system_psoc6.h* – This file provides information about the chip initialization that is done pre- main().

5.2.6 <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.

5.2.7 locate_recipe.mk

This is a helper file for the BSP to specify the path to the core and recipe makefiles that are included as dependent libraries.

5.2.8 README/RELEASE.md

These are documentation files. The *README.md* file describes the BSP overall, while the *RELEASE.md* file covers changes made to version of the BSP.

5.2.9 BTSDK-Specific BSP files

BTSDK BSPs may optionally provide the following types of files:

- *wiced_platform.h* – Platform specific structures to define hardware information such as max number of GPIOs, LEDs or.user buttons available
- *makefile* – Provided to support LIB flow applications (BTSDK 2.7 and earlier). Not used in MTB flow BTSDK 2.8 or later applications.
- **.hex* – binary application image files that are used as part of the embedded application creation, program, and/or OTA (Over-The-Air) upgrade processes.
- *platform*.c/h* – Platform specific source and header files used by platform and application initialization functions.
- *<BSP_NAME>*.cgs* – Patch configuration records in text format, can be multiple copies supporting various board configurations.
- *<BSP_NAME>*.btp* – Configuration options related to building and programming the application image, can be multiple copies supporting various board configurations.

5.3 Creating your Own BSP

In most cases before you do any real design work on your application, you should create a BSP for your device and/or board. This allows you to configure the settings for your own custom hardware or for different linker options. Plus, you can save the BSP for use in future applications.

There are two basic methods to create a BSP; each involves creating an application. Using the first method, specify the closest-matching BSP to your intended BSP. In this case, you usually have to remove various settings and options that you don't need. For the second method, specify a "generic" BSP template when creating your application. In this case, your BSP is essentially built from scratch, and you need to add and configure settings and options for your needs.

Regardless of the method you choose, the basic process is the same for both:

1. Create a simple example application. Use a BSP that is close to your goal or select a "generic" BSP.

2. Navigate to the application directory, and run the `make bsp` target. Specify the new board name by passing the value to the `TARGET_GEN` variable. This this is the minimum required. For example, to create a BSP called MyBSP:

   ```
   make bsp TARGET_GEN=MyBSP
   ```

   Optionally, you may use `DEVICE_GEN` specify a new device if it is different than the one included with the original BSP. For example:

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

   The `make bsp` command creates a new BSP with the provided name at the top of the application project. It automatically copies the relevant startup and linker scripts into the newly created BSP, based on the device specified by the `DEVICE_GEN` option.

   It also creates *.mtbx* files for all the BSP's dependences. The Project Creator tool uses these files when you import your custom BSP into that tool. These files can also be used with the `make import_deps` command if you need to manually include the custom BSP's dependencies. Refer to the Library Manager User Guide for details about managing BSPs and libraries for custom BSPs.

   Note The BSP used as your starting point may have library references (for example, *capsense.lib* or *udb-sdio-whd.lib*) that are not needed by your custom BSP. You can delete these from the BSP's *deps* subdirectory. Be sure to remove the corresponding *.mtbx* files as well.

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

   ```
   make import_deps IMPORT_PATH=TARGET_MyBSP
   ```

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

4. Update the application's makefile `TARGET` variable to point to your new BSP. For example:

   ```
   TARGET=MyBSP
   ```

5. Optionally, update libraries in the application to make sure all the dependencies are aligned.

   ```
   make getlibs
   ```

6. Open the Device Configurator to customize settings in the new BSP's *design.modus* file for pin names, clocks, power supplies, and peripherals as required. Also, address any issues that arise.

7. Start writing code for your application.

If using an IDE, you need to generate/regenerate the configuration settings to reflect the new BSP. Use the appropriate command(s) for the IDE(s) that are being used. For example:

```
make vscode
```

Note 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 version control system such as Git. See the Manifest Files chapter for information on how to create a manifest to include your custom BSPs and their dependencies if you want them to show up as standard BSPs in the Project Creator and Library Manager.

5.4 Modifying the BSP Configuration for a Single Application

In cases where you don't want to create a BSP, you can modify the BSP configuration for a single application (such as different pin or peripheral settings). However, you should not typically modify the BSP directly since that results in changes to the BSP library. This would prevent you from updating the repository in the future, and it may affect other applications in the same workspace. Instead, use the following process to create a custom set of configuration files for a specific application:

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

   Hello_World/COMPONENT_CUSTOM_DESIGN_MODUS

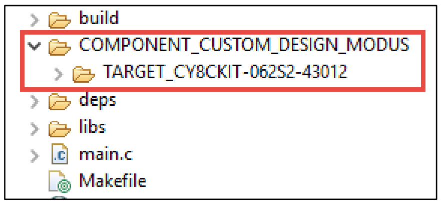
   This is a recommended best practice. In an upcoming step, you will modify the makefile to include files from that directory instead of the directory containing the default configuration files in the BSP.

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

   Hello_World/COMPONENT_CUSTOM_DESIGN_MODUS/TARGET_CY8CKIT-062S2-43012

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

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



3. Copy the *design.modus* file and other configuration files (that is, everything from inside the original BSP's *COMPONENT_BSP_DESIGN_MODUS* directory), and paste them into the new directory for the target.

4. In the application's makefile, add the following lines. For example:

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

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

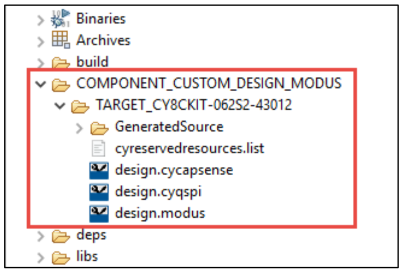
   The first line disables the configuration files from the original BSP since they are now in different directory.

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

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

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

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



```
> Binaries
> Archives
> build
v COMPONENT_CUSTOM_DESIGN_MODUS
  v TARGET_CY8CKIT-062S2-43012
    > GeneratedSource
      cyreservedresources.list
      design.cycapsense
      design.cyqspi
      design.modus
> deps
> libs
```

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

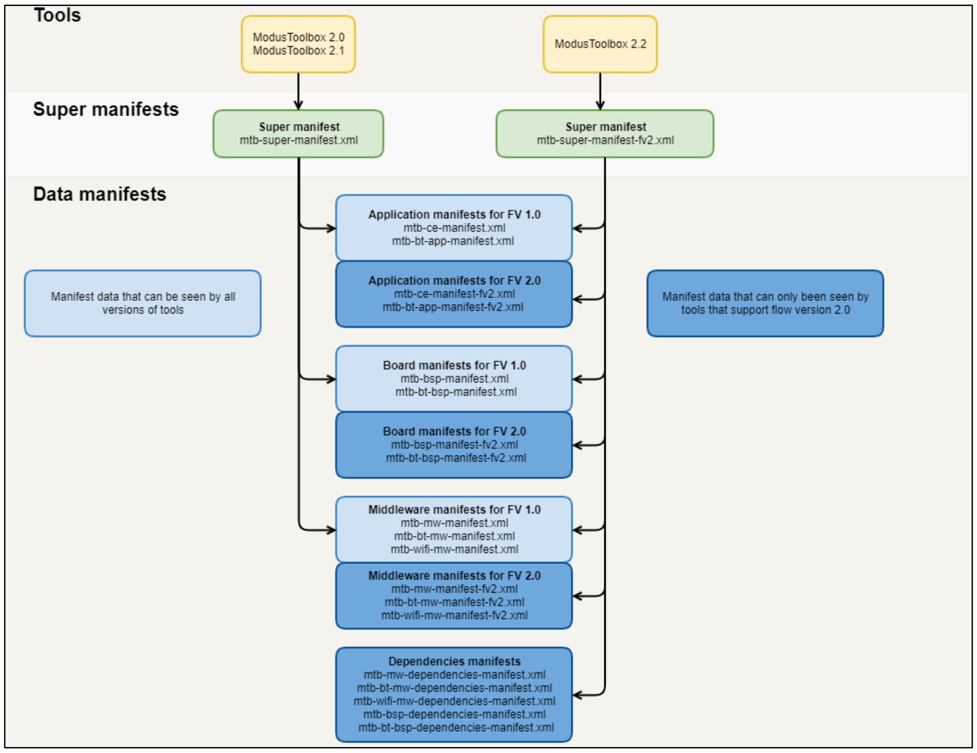
6 Manifest Files

6.1 Overview

Manifests are XML files that tell the Project Creator and Library Manager how to discover the list of available boards, example projects, libraries and library dependencies. There are several manifest files.

- The “super-manifest” file contains a list of URLs that software uses to find the board, code example, and middleware manifest files.
- The “app-manifest” file contains a list of all code examples that should be made available to the user.
- The “board-manifest” file contains a list of the boards that should be presented to the user in the new project creation tool as well as the list of BSP packages that are presented in the Library Manager tool. There is also a separate BSP dependencies manifest that lists the dependent libraries associated with each BSP.
- The “middleware-manifest” file contains a list of the available middleware (libraries). Each middleware item can have one or more versions of that middleware available. There is also a separate middleware dependencies manifest that lists the dependent libraries associated with each middleware library.

Beginning with ModusToolbox 2.2, there are two versions of manifest files: the existing ones for the LIB flow and earlier versions of ModusToolbox, and new ones for the MTB flow (aka "fv2"). The existing super-manifest file for use with ModusToolbox 2.1 and earlier contains only references to manifests that contain items that support the LIB flow. The new super-manifest file for use with ModusToolbox 2.2 and later contains references to all the manifest files.



Tools
ModusToolbox 2.0
ModusToolbox 2.1
ModusToolbox 2.2
Super manifests
Super manifest
mtb-super-manifest.xml
Super manifest
mtb-super-manifest-fv2.xml
Data manifests
Application manifests for FV 1.0
mtb-ce-manifest.xml
mtb-bt-app-manifest.xml
Application manifests for FV 2.0
mtb-ce-manifest-fv2.xml
mtb-bt-app-manifest-fv2.xml
Manifest data that can be seen by all versions of tools
Manifest data that can only been seen by tools that support flow version 2.0
Board manifests for FV 1.0
mtb-bsp-manifest.xml
mtb-bt-bsp-manifest.xml
Board manifests for FV 2.0
mtb-bsp-manifest-fv2.xml
mtb-bt-bsp-manifest-fv2.xml
Middleware manifests for FV 1.0
mtb-mw-manifest.xml
mtb-bt-mw-manifest.xml
mtb-wifi-mw-manifest.xml
Middleware manifests for FV 2.0
mtb-mw-manifest-fv2.xml
mtb-bt-mw-manifest-fv2.xml
mtb-wifi-mw-manifest-fv2.xml
Dependencies manifests
mtb-mw-dependencies-manifest.xml
mtb-bt-mw-dependencies-manifest.xml
mtb-wifi-mw-dependencies-manifest.xml
mtb-bsp-dependencies-manifest.xml
mtb-bt-bsp-dependencies-manifest.xml

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 of 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, you need to create manifest files for your content and a super-manifest that points to your manifest files. To see examples of the syntax of super-manifest and manifest files, you can look at the Cypress provided files on GitHub:

- Super Manifest: https://github.com/cypresssemiconductorco/mtb-super-manifest
- Code Example Manifest: https://github.com/cypresssemiconductorco/mtb-ce-manifest
- BSP Manifest (including dependencies): https://github.com/cypresssemiconductorco/mtb-bsp-manifest
- Middleware Manifest (including dependencies): https://github.com/cypresssemiconductorco/mtb-mw-manifest

Make sure you look at the "fv2" manifest files if you are using the MTB flow.

The manifest system is flexible, and there are multiple paths you can follow to customize the manifests.

- You can customize a super-manifest file and override the default file used by the tools.
- You can create secondary super-manifest files that identify additional content. The tools will merge your additional content with the default super-manifest.
- You can modify or replace any of the default manifest files (code example, BSP, etc.) with custom files, so long as your custom super-manifest file points to those rather than the default 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 the 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 Secondary Super-Manifest

In addition to the standard super-manifest file, you can specify additional super-manifest files. 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:

<user_home>/.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
```

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

file:///C:/MyManfests/my-super-manifest.xml

If the *manifest.loc* 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.

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.

IMPORTANT ModusToolbox libraries are updated frequently, and the offline content does not always have the latest versions available. We strongly recommend using the online content whenever possible. See https://community.cypress.com/docs/DOC-19903 for more details.

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).

   Note Make sure `CyRemoteManifestOverride` variable is not set when you use offline mode.

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 (repos) hosted on GitHub or any other online git server. To access private git repos, 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.

Note While you can host libraries on private repos, the custom content manifest must be accessible without authentication (that is, it cannot be hosted on a private git repo).

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=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx # 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.

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 described Getting Started chapter, the Project Creator tool includes a **Select Target IDE** option that generates files for the selected IDE. Also, as noted in the ModusToolbox Build System chapter, the make command includes various targets for the following IDEs:

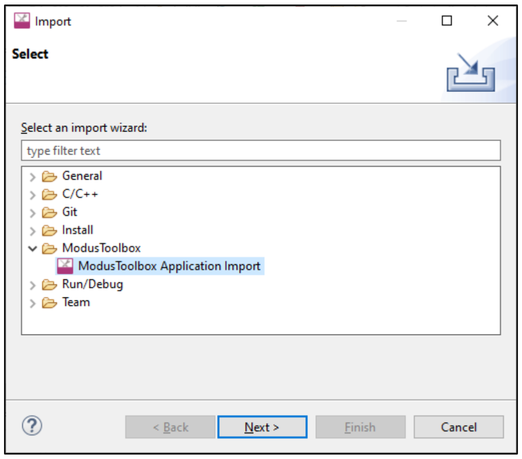
- Visual Studio (VS) Code – `make vscode`
- IAR Embedded Workbench – `make ewarm8 TOOLCHAIN=IAR`
- Keil µVision – `make uvision5 TOOLCHAIN=ARM`

7.2 Import to Eclipse

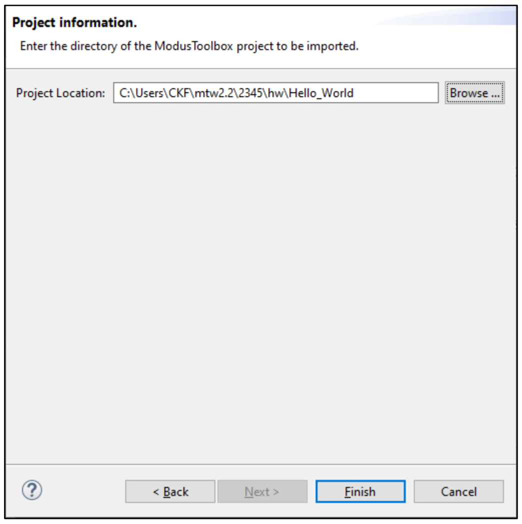
The easiest way to create a ModusToolbox application for Eclipse is to use the Eclipse IDE included with the ModusToolbox software. ModusToolbox includes an Eclipse plugin that provides links to launch the Project Creator tool and then import the application into Eclipse. For details, refer to the Eclipse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

If you already have a ModusToolbox application created some other way than through the included Eclipse IDE, you can import that application for use in Eclipse as follows:

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



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


Project information.

Enter the directory of the ModusToolbox project to be imported.

Project Location: C:\Users\CKF\mtw2.2\2345\hw\Hello_World

3. Click **Finish**.

   The application displays in the Eclipse IDE Project Explorer.

7.3 Export to VS Code

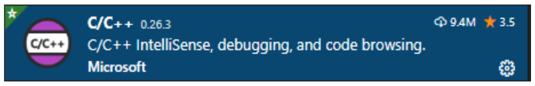
This section describes how to export a ModusToolbox application to VS Code.

7.3.1 Prerequisites

- ModusToolbox 2.3 software and application
- VS Code version 1.42.x or later
- VS Code extensions. Install the following.

  Note These versions change often; use the most current.

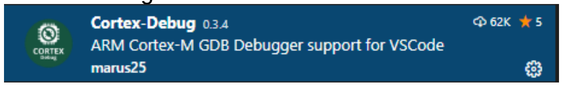
  - ☐ C/C++ tools

    
    C/C++ 0.26.3

    C/C++ IntelliSense, debugging, and code browsing.

    Microsoft

  - ☐ Cortex-Debug

    
    Cortex-Debug 0.3.4

    ARM Cortex-M GDB Debugger support for VSCode

    marus25

- For J-Link debugging, download and install J-Link software:

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

7.3.2 Process Example

1. Create a ModusToolbox application.

   a. If you use the Project Creator tool, choose "VS Code" from the **Target IDE** pull down menu.

   b. If you use the command line, open an appropriate shell program (see CLI Set-up Instructions), and navigate to the application directory, and run the following command:

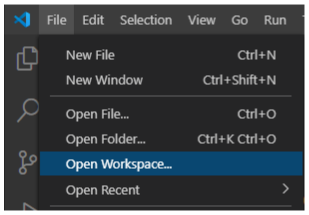
   ```
   make vscode
   ```

   Either process generates json files for debug/program launches, IntelliSense, and custom tasks.

   Note Any time you update/patch the tools for your application(s), that path information might change. So you will need to regenerate the needed support files by running the `make vscode` command or update them manually.

2. Open the VS Code tool.

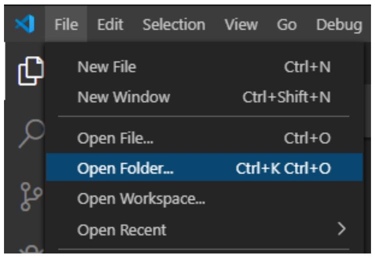
   a. To open the application and the *mtb_shared* directory in the same workspace, select **File > Open Workspace…**



   Navigate to the application directory and select the *<application_name>.code-workspace* file.

   If you have several applications in the workspace, you can add them using **Add workspace folder…**

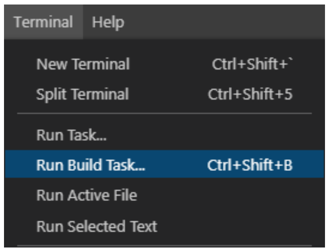
   b. To open just the application and select **File > Open Folder…**



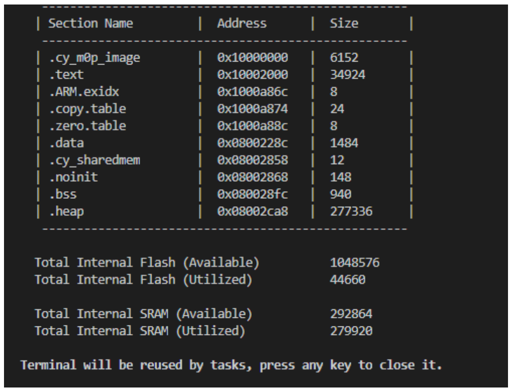
   Note On macOS, this command is **File > Open…**

   Navigate to and select the application directory, and then click **Select Directory**.

3. When your application opens in the VS Code IDE, select **Terminal > Run Build Task…**



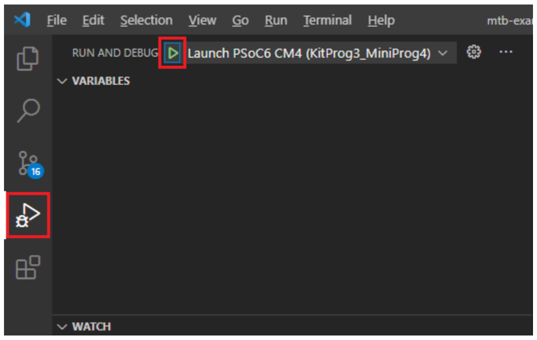
4. Then, select **Build: Build [Debug]**. After building, the VS Code terminal should display messages similar to the following:



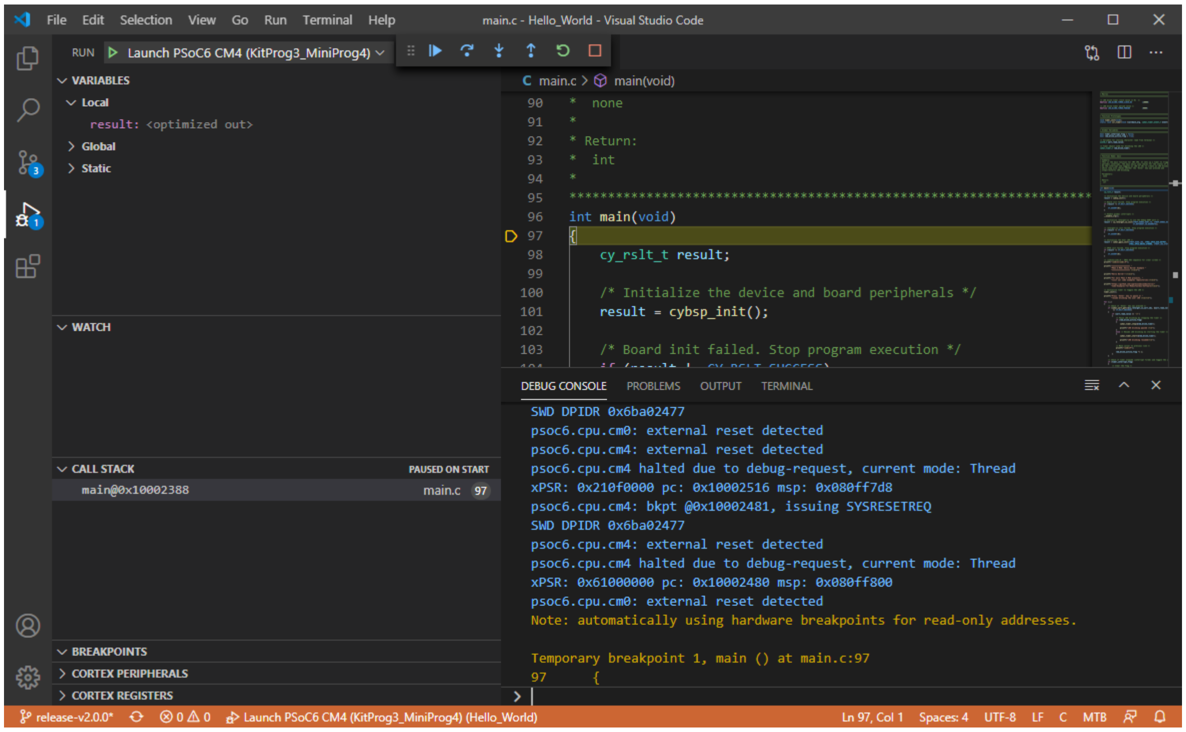
```
 ----------------------------------------------------------
 | Section Name          |  Address      |  Size       |
 ----------------------------------------------------------
 | .cy_m0p_image         |  0x10000000   |  6152       |
 | .text                 |  0x10002000   |  34924      |
 | .ARM.exidx            |  0x1000a86c   |  8          |
 | .copy.table           |  0x1000a874   |  24         |
 | .zero.table           |  0x1000a88c   |  8          |
 | .data                 |  0x0800228c   |  1484       |
 | .cy_sharedmem         |  0x08002858   |  12         |
 | .noinit               |  0x08002868   |  148        |
 | .bss                  |  0x080028fc   |  940        |
 | .heap                 |  0x08002ca8   |  277336     |
 ----------------------------------------------------------

 Total Internal Flash (Available)          1048576
 Total Internal Flash (Utilized)           44660

 Total Internal SRAM (Available)           292864
 Total Internal SRAM (Utilized)            279920

Terminal will be reused by tasks, press any key to close it.
```

7.3.2.1 *To Debug using KitProg3/MiniProg4*

Click the **Run and Debug** icon on the left and then click the **Play** button.


File Edit Selection View Go Run Terminal Help
mtb-exa
RUN AND DEBUG
Launch PSoC6 CM4 (KitProg3_MiniProg4)
VARIABLES
16
WATCH

The VS Code tool runs in debug mode.



```c
 *  none
 *
 * Return:
 *  int
 *
 *******************************************************************
int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init();

    /* Board init failed. Stop program execution */
```

```
SWD DPIDR 0x6ba02477
psoc6.cpu.cm0: external reset detected
psoc6.cpu.cm4: external reset detected
psoc6.cpu.cm4 halted due to debug-request, current mode: Thread
xPSR: 0x210f0000 pc: 0x10002516 msp: 0x080ff7d8
psoc6.cpu.cm4: bkpt @0x10002481, issuing SYSRESETREQ
SWD DPIDR 0x6ba02477
psoc6.cpu.cm4: external reset detected
psoc6.cpu.cm4 halted due to debug-request, current mode: Thread
xPSR: 0x61000000 pc: 0x10002480 msp: 0x080ff800
psoc6.cpu.cm0: external reset detected
Note: automatically using hardware breakpoints for read-only addresses.

Temporary breakpoint 1, main () at main.c:97
97      {
```

7.3.2.2 *To Debug using J-Link*

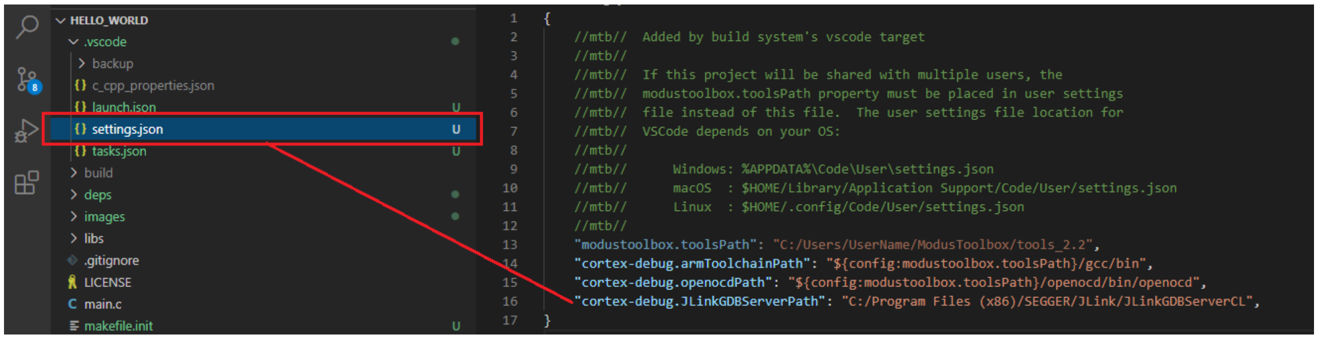
You can use a J-Link debugger probe to debug the application.

1. Navigate to and open the global *settings.json* file. If there is no such file, then create one. The file is located here by default:
   - ◻ Windows: *%APPDATA%/Code/User/settings.json*
   - ◻ macOS: *$HOME/Library/Application Support/Code/User/settings.json*
   - ◻ Linux: *$HOME/.config/Code/User/settings.json*
2. Add the path to the J-Link GDB server. For example:

   ```
   {"cortex-debug.JLinkGDBServerPath": "C:/Program Files (x86)/SEGGER/JLink/JLinkGDBServerCL"}
   ```

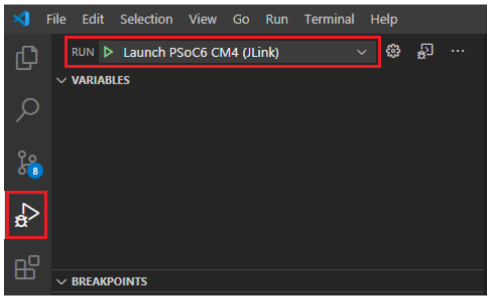
   - ◻ Windows: `"cortex-debug.JLinkGDBServerPath": "<JLinkInstallDir>/JLinkGDBServerCL"`
   - ◻ macOS/Linux: `"cortex-debug.JLinkGDBServerPath": "<JLinkInstallDir>/JLinkGDBServer"`

   Note The J-Link path can be configured in the local application's settings, if needed.



```
{
    //mtb//  Added by build system's vscode target
    //mtb//
    //mtb//  If this project will be shared with multiple users, the
    //mtb//  modustoolbox.toolsPath property must be placed in user settings
    //mtb//  file instead of this file.  The user settings file location for
    //mtb//  VSCode depends on your OS:
    //mtb//
    //mtb//      Windows: %APPDATA%\Code\User\settings.json
    //mtb//      macOS  : $HOME/Library/Application Support/Code/User/settings.json
    //mtb//      Linux  : $HOME/.config/Code/User/settings.json
    //mtb//
    "modustoolbox.toolsPath": "C:/Users/UserName/ModusToolbox/tools_2.2",
    "cortex-debug.armToolchainPath": "${config:modustoolbox.toolsPath}/gcc/bin",
    "cortex-debug.openocdPath": "${config:modustoolbox.toolsPath}/openocd/bin/openocd",
    "cortex-debug.JLinkGDBServerPath": "C:/Program Files (x86)/SEGGER/JLink/JLinkGDBServerCL",
}
```

3. Click the **Run and Debug** icon, select **Launch PSOC6 CM4 (JLink)** config, and click the **Play** button.



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 2.3 software and application
- Python 3.7 is installed in the *tools_2.3* directory, and the make build system has been configured to use it. You don't need to do anything if you use the *modus-shell/Cygwin.bat* file to run command line tools.

  However, if you plan to use your own version of Cygwin or some other type of bash, you will need to ensure your system is configured correctly to use Python 3.7. Use the `CY_PYTHON_PATH` as appropriate.
- 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 Example

1. Create a ModusToolbox application.

   a. If you use the Project Creator tool, choose "IAR" from the **Target IDE** pull down menu.

   b. If you use the command line, open an appropriate shell program (see CLI Set-up Instructions), navigate to the application directory, and run the following command:

   ```
   make ewarm8 TOOLCHAIN=IAR
   ```

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

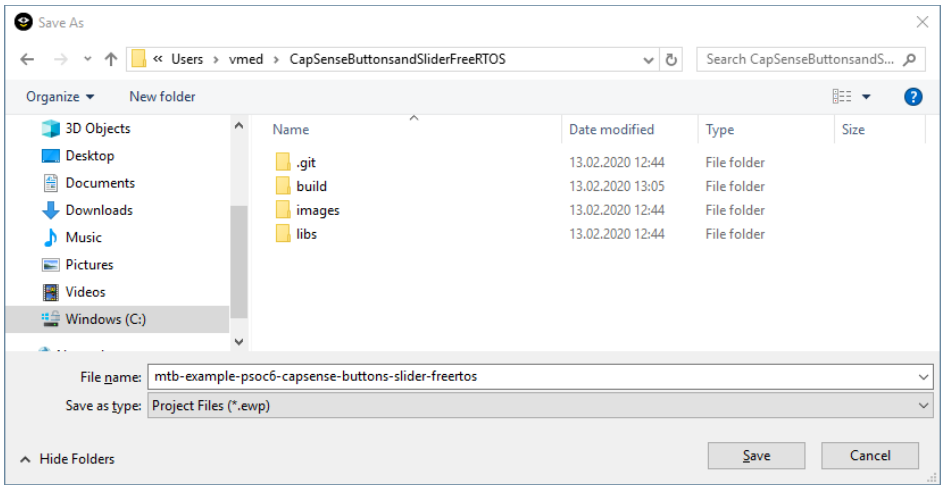
   An IAR connection file appears in the application directory. For example:

   mtb-example-psoc6-capsense-buttons-slider-freertos.ipcf

2. Start IAR Embedded Workbench.

3. On the main menu, select **Project > Create New Project > Empty project** and click **OK**.

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



Name	Date modified	Type	Size
.git	13.02.2020 12:44	File folder	
build	13.02.2020 13:05	File folder	
images	13.02.2020 12:44	File folder	
libs	13.02.2020 12:44	File folder	

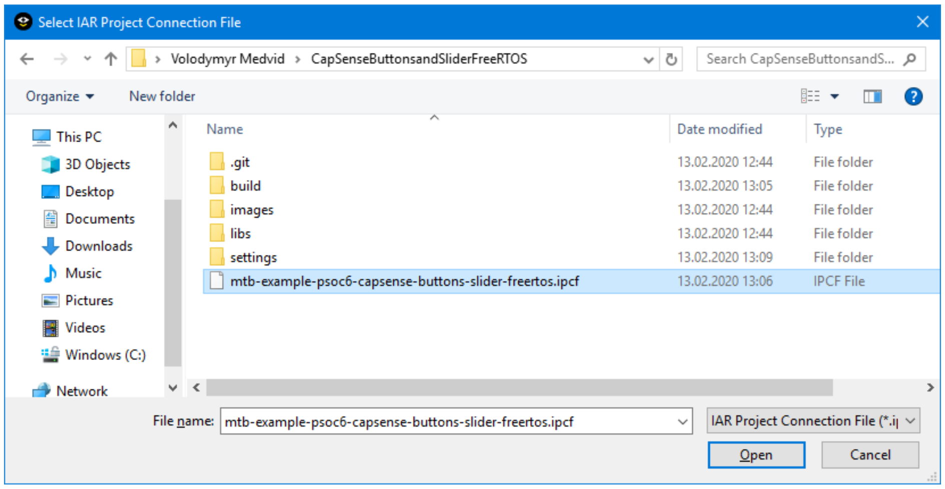
File name: mtb-example-psoc6-capsense-buttons-slider-freertos

Save as type: Project Files (*.ewp)

5. After the application is created, select **File > Save Workspace**, enter an arbitrary workspace name and click **Save**.

6. Select **Project > Add Project Connection** and click **OK**.

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


Select IAR Project Connection File
Volodymyr Medvid > CapSenseButtonsandSliderFreeRTOS
Search CapSenseButtonsandS...
Organize
New folder
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Windows (C:)
Network
Open
Cancel

Name	Date modified	Type
.git	13.02.2020 12:44	File folder
build	13.02.2020 13:05	File folder
images	13.02.2020 12:44	File folder
libs	13.02.2020 12:44	File folder
settings	13.02.2020 13:09	File folder
mtb-example-psoc6-capsense-buttons-slider-freertos.ipcf	13.02.2020 13:06	IPCF File

File name: mtb-example-psoc6-capsense-buttons-slider-freertos.ipcf

IAR Project Connection File (*.ip

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

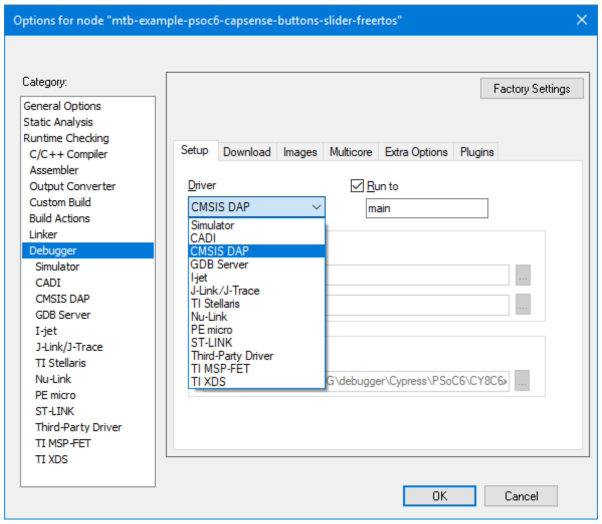
9. Connect the PSoC 6 kit to the host PC.

7.4.2.1 *To Use KitProg3/MiniProg4*

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

   <user_home>/ModusToolbox/tools_2.3/fw-loader/bin/

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

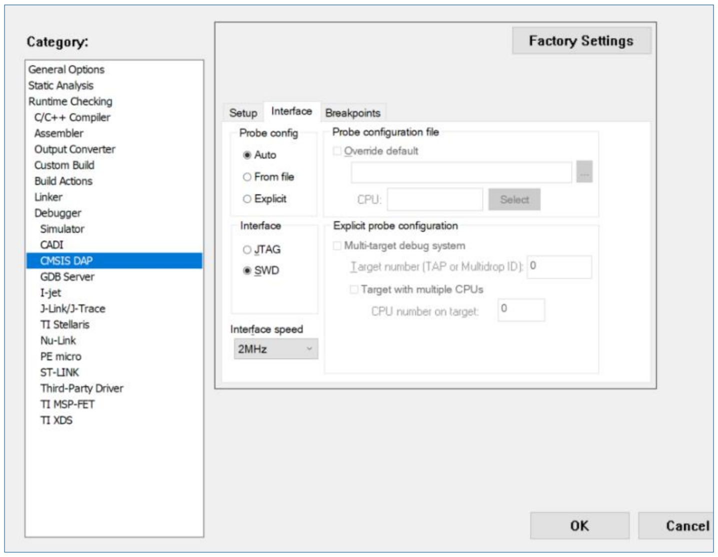


Driver: CMSIS DAP

☑ Run to: main

G\debugger\Cypress\PSoC6\CY8C6

3. Select the **CMSIS-DAP** node, switch the interface from **JTAG** to **SWD**, and set the Interface speed to **2MHZ**.



Probe config: [✓ Auto] From file Explicit

Probe configuration file: ☐ Override default

CPU:

Interface: JTAG [✓ SWD]

Interface speed: 2MHz

Explicit probe configuration: ☐ Multi-target debug system

Target number (TAP or Multidrop ID): 0

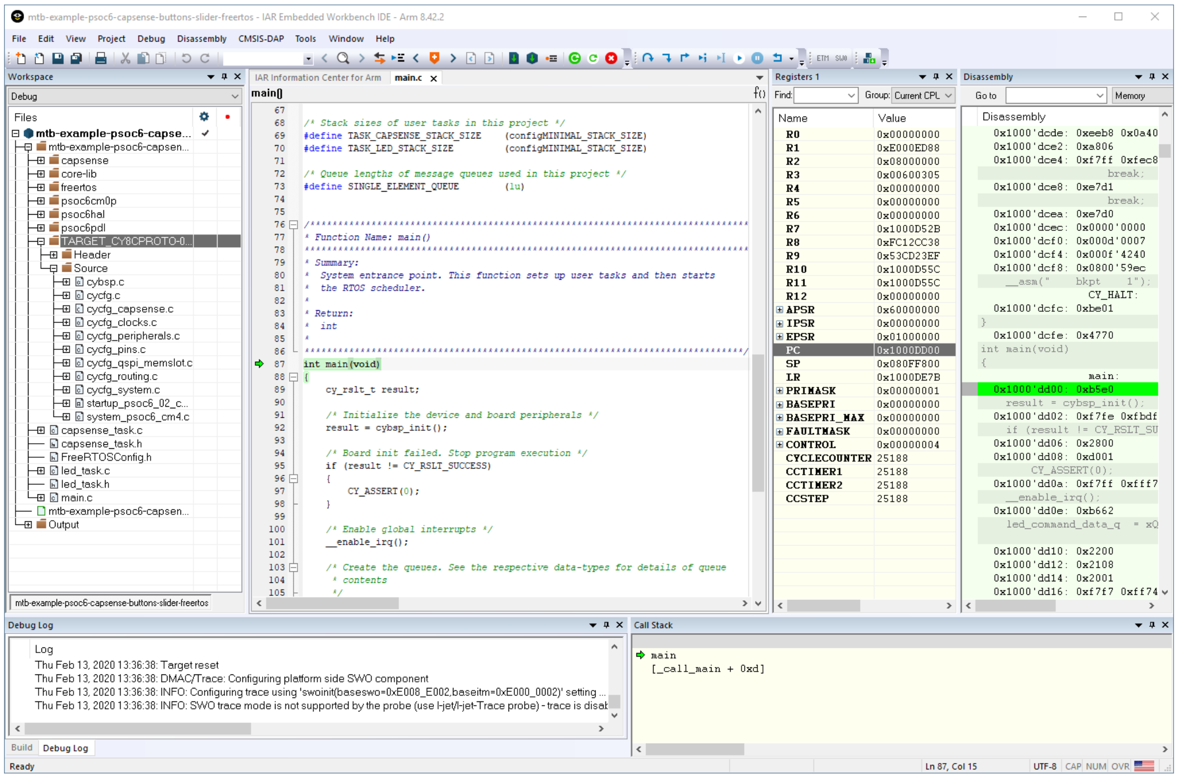
☐ Target with multiple CPUs

CPU number on target: 0

4. Click **OK**.

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

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



```

/* Stack sizes of user tasks in this project */
#define TASK_CAPSENSE_STACK_SIZE    (configMINIMAL_STACK_SIZE)
#define TASK_LED_STACK_SIZE         (configMINIMAL_STACK_SIZE)

/* Queue lengths of message queues used in this project */
#define SINGLE_ELEMENT_QUEUE        (1u)


/*******************************************************************************
* Function Name: main()
********************************************************************************
* Summary:
*  System entrance point. This function sets up user tasks and then starts
*  the RTOS scheduler.
*
* Return:
*  int
*
*******************************************************************************/
int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init();

    /* Board init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Enable global interrupts */
    __enable_irq();

    /* Create the queues. See the respective data-types for details of queue
     * contents
     */
```

```
Name          Value
R0            0x00000000
R1            0xE000ED88
R2            0x08000000
R3            0x00600305
R4            0x00000000
R5            0x00000000
R6            0x00000000
R7            0x1000D52B
R8            0xFC12CC38
R9            0x53CD23EF
R10           0x1000D55C
R11           0x1000D55C
R12           0x00000000
APSR          0x60000000
IPSR          0x00000000
EPSR          0x01000000
PC            0x1000DD00
SP            0x080FF800
LR            0x1000DE7B
PRIMASK       0x00000001
BASEPRI       0x00000000
BASEPRI_MAX   0x00000000
FAULTMASK     0x00000000
CONTROL       0x00000004
CYCLECOUNTER  25188
CCTIMER1      25188
CCTIMER2      25188
CCSTEP        25188
```

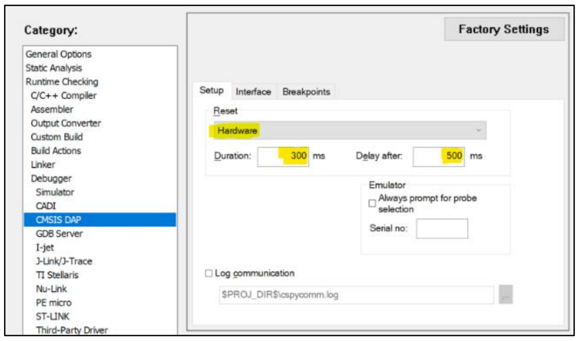
```
Disassembly
  0x1000'dcde: 0xeeb8 0x0a40
  0x1000'dce2: 0xa806
  0x1000'dce4: 0xf7ff 0xfec8
                    break;
  0x1000'dce8: 0xe7d1
                    break;
  0x1000'dcea: 0xe7d0
  0x1000'dcec: 0x0000'0000
  0x1000'dcf0: 0x000d'0007
  0x1000'dcf4: 0x000f'4240
  0x1000'dcf8: 0x0800'59ec
    __asm("    bkpt    1");
                 CY_HALT:
  0x1000'dcfc: 0xbe01
}
  0x1000'dcfe: 0x4770
int main(void)
{
                    main:
  0x1000'dd00: 0xb5e0
    result = cybsp_init();
  0x1000'dd02: 0xf7fe 0xfbdf
    if (result != CY_RSLT_SU
  0x1000'dd06: 0x2800
  0x1000'dd08: 0xd001
        CY_ASSERT(0);
  0x1000'dd0a: 0xf7ff 0xfff7
    __enable_irq();
  0x1000'dd0e: 0xb662
    led_command_data_q   = xQ

  0x1000'dd10: 0x2200
  0x1000'dd12: 0x2108
  0x1000'dd14: 0x2001
  0x1000'dd16: 0xf7f7 0xff74
```

```
Log
Thu Feb 13, 2020 13:36:38: Target reset
Thu Feb 13, 2020 13:36:38: DMAC/Trace: Configuring platform side SWO component
Thu Feb 13, 2020 13:36:38: INFO: Configuring trace using 'swoinit(baseswo=0xE008_E002,baseitm=0xE000_0002)' setting ...
Thu Feb 13, 2020 13:36:38: INFO: SWO trace mode is not supported by the probe (use I-jet/I-jet-Trace probe) - trace is disat
```

```
main
  [_call_main + 0xd]
```

7.4.2.2 *To use MiniProg4 with PSoC 6 Single Core and PSoC 6 256K*

For a single-core PSoC 6 MCU, you must specify a special type of reset, as follows:



Reset: Hardware

Duration: 300 ms   Delay after: 500 ms

☐ Always prompt for probe selection

Serial no:

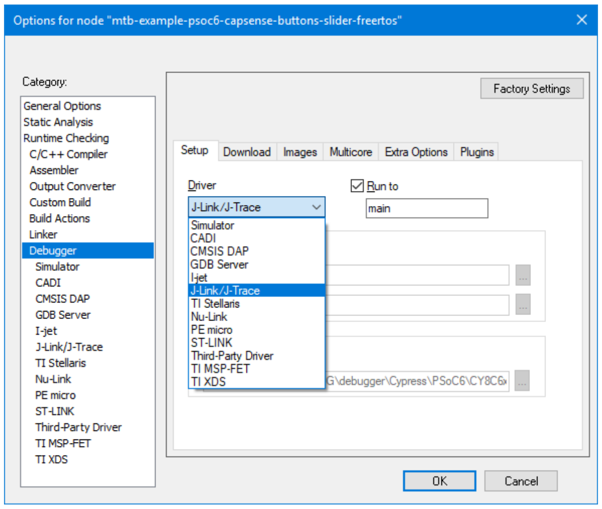
☐ Log communication

$PROJ_DIR$\cspycomm.log

7.4.2.3 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:



Options for node "mtb-example-psoc6-capsense-buttons-slider-freertos"

Category:
General Options
Static Analysis
Runtime Checking
C/C++ Compiler
Assembler
Output Converter
Custom Build
Build Actions
Linker
Debugger
Simulator
CADI
CMSIS DAP
GDB Server
I-jet
J-Link/J-Trace
TI Stellaris
Nu-Link
PE micro
ST-LINK
Third-Party Driver
TI MSP-FET
TI XDS

Factory Settings

Setup | Download | Images | Multicore | Extra Options | Plugins

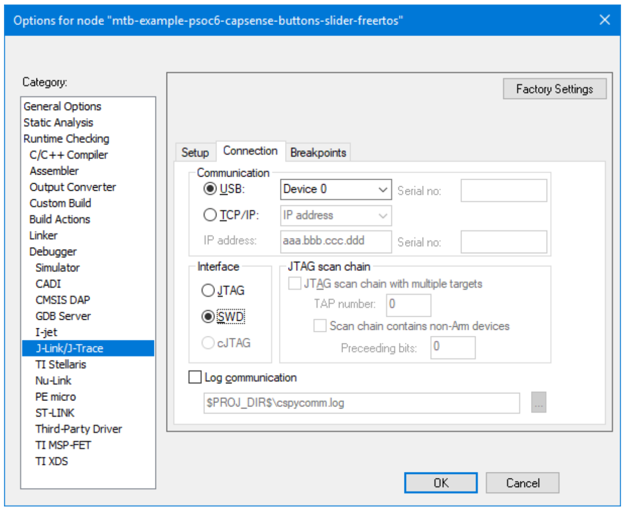
Driver: J-Link/J-Trace
Simulator, CADI, CMSIS DAP, GDB Server, I-jet, J-Link/J-Trace, TI Stellaris, Nu-Link, PE micro, ST-LINK, Third-Party Driver, TI MSP-FET, TI XDS

[✓] Run to: main

G\debugger\Cypress\PSoC6\CY8C6

OK | Cancel

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



Options for node "mtb-example-psoc6-capsense-buttons-slider-freertos"

Category: J-Link/J-Trace

Factory Settings

Setup | Connection | Breakpoints

Communication
[●] USB: Device 0 — Serial no:
[○] TCP/IP: IP address — IP address: aaa.bbb.ccc.ddd — Serial no:

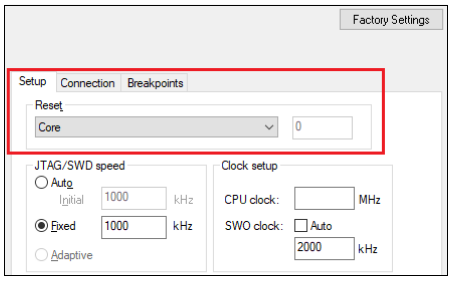
Interface
[○] JTAG
[●] SWD
[○] cJTAG

JTAG scan chain
[ ] JTAG scan chain with multiple targets
TAP number: 0
[ ] Scan chain contains non-Arm devices
Preceeding bits: 0

[ ] Log communication
$PROJ_DIR$\cspycomm.log

OK | Cancel

Note For PSoC 64 MCU, you must specify a special type of reset, as follows:



Factory Settings

Setup | Connection | Breakpoints

Reset: Core — 0

JTAG/SWD speed
[○] Auto — Initial: 1000 kHz
[●] Fixed: 1000 kHz
[○] Adaptive

Clock setup
CPU clock: MHz
SWO clock: [ ] Auto — 2000 kHz

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

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 2.3 software and application
- Python 3.7 is installed in the *tools_2.3* directory, and the make build system has been configured to use it. You don't need to do anything if you use the *modus-shell/Cygwin.bat* file to run command line tools.

  However, if you plan to use your own version of Cygwin or some other type of bash, you will need to ensure your system is configured correctly to use Python 3.7. Use the `CY_PYTHON_PATH` as appropriate.
- 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 Example

1. Create a ModusToolbox application.

   a. If you use the Project Creator tool, choose "ARM MDK" from the **Target IDE** pull down menu.

   b. If you use the command line, open an appropriate shell program (see CLI Set-up Instructions), navigate to the application directory, and Run the following command:

   ```
   make uvision5 TOOLCHAIN=ARM
   ```

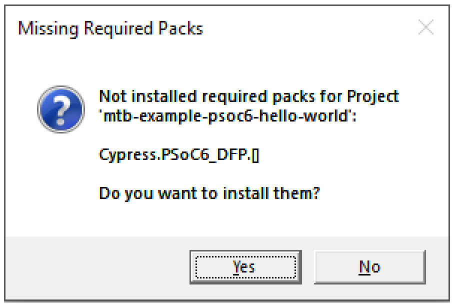
   Note Alternately, you can edit the application's makefile to specify a toolchain.

   This generates the following files in the application directory:

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

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

2. Double-click the *mtb-example-psoc6-hello-world* file (either **.cpdsc* or **.cprj*, depending on version). This launches the Keil µVision IDE. The first time you do this, the following dialog displays:



Missing Required Packs

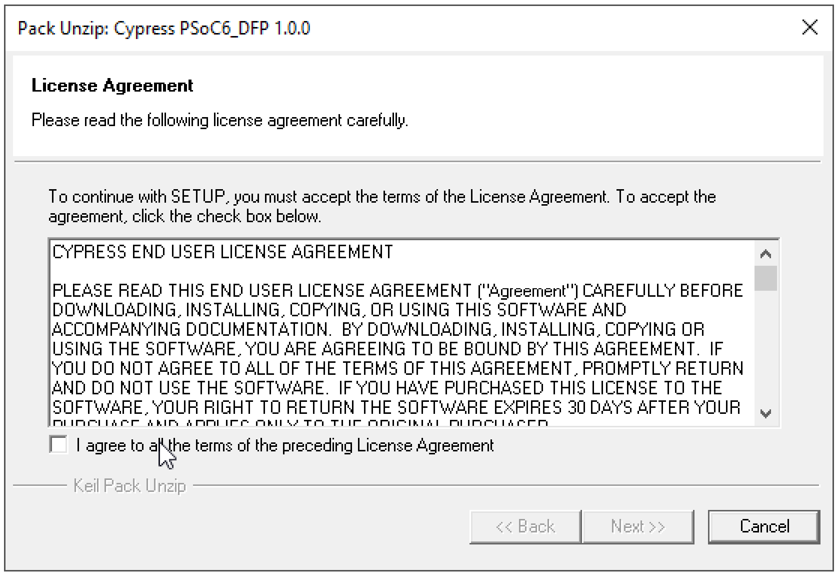
Not installed required packs for Project 'mtb-example-psoc6-hello-world':

Cypress.PSoC6_DFP.[]

Do you want to install them?

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

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


Pack Unzip: Cypress PSoC6_DFP 1.0.0

License Agreement

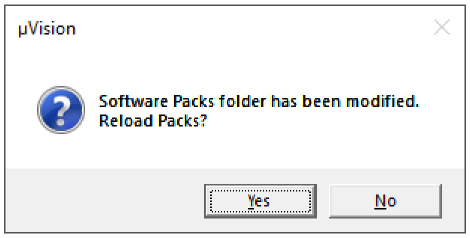
Please read the following license agreement carefully.

To continue with SETUP, you must accept the terms of the License Agreement. To accept the agreement, click the check box below.

CYPRESS END USER LICENSE AGREEMENT

PLEASE READ THIS END USER LICENSE AGREEMENT ("Agreement") CAREFULLY BEFORE DOWNLOADING, INSTALLING, COPYING, OR USING THIS SOFTWARE AND ACCOMPANYING DOCUMENTATION. BY DOWNLOADING, INSTALLING, COPYING OR USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL OF THE TERMS OF THIS AGREEMENT, PROMPTLY RETURN AND DO NOT USE THE SOFTWARE. IF YOU HAVE PURCHASED THIS LICENSE TO THE SOFTWARE, YOUR RIGHT TO RETURN THE SOFTWARE EXPIRES 30 DAYS AFTER YOUR

☐ I agree to all the terms of the preceding License Agreement


µVision

Software Packs folder has been modified.
Reload Packs?

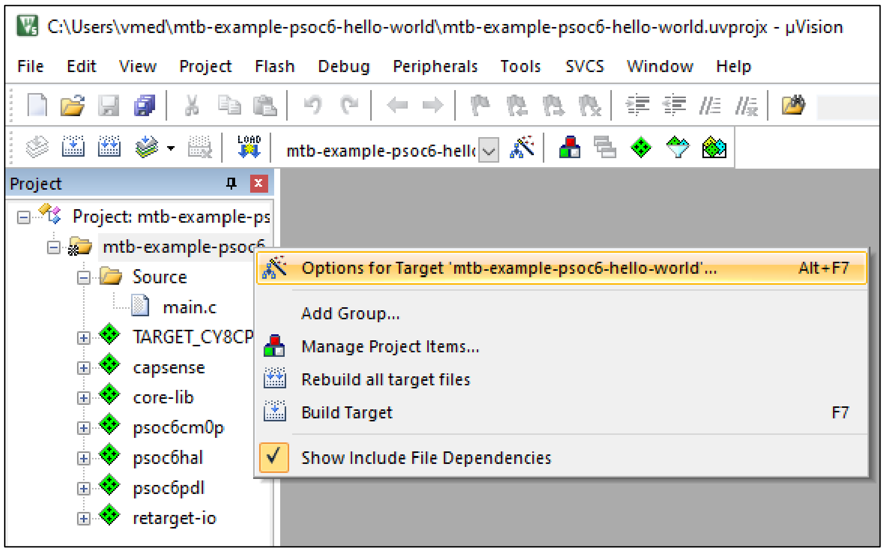
Note In some cases, you may see the following error message:

SSL caching disabled in Windows Internet settings. Switched to offline mode.

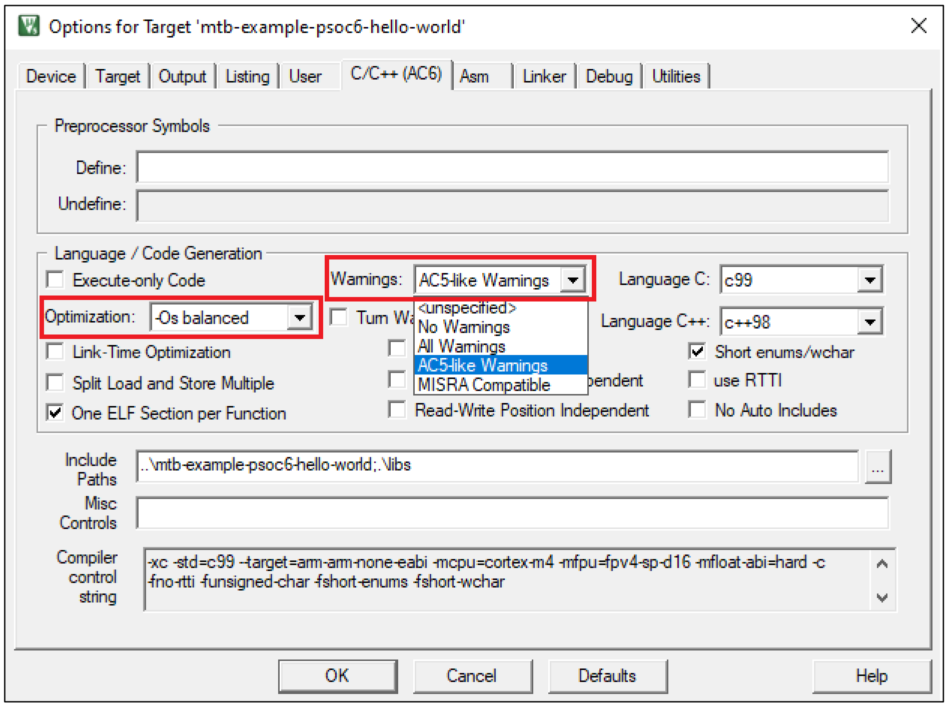
See this link for how to solve this problem: https://developer.arm.com/documentation/ka002253/latest

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

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


C:\Users\vmed\mtb-example-psoc6-hello-world\mtb-example-psoc6-hello-world.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
mtb-example-psoc6-hell
Project
Project: mtb-example-ps
mtb-example-psoc6
Source
main.c
TARGET_CY8CP
capsense
core-lib
psoc6cm0p
psoc6hal
psoc6pdl
retarget-io
Options for Target 'mtb-example-psoc6-hello-world'... Alt+F7
Add Group...
Manage Project Items...
Rebuild all target files
Build Target F7
Show Include File Dependencies

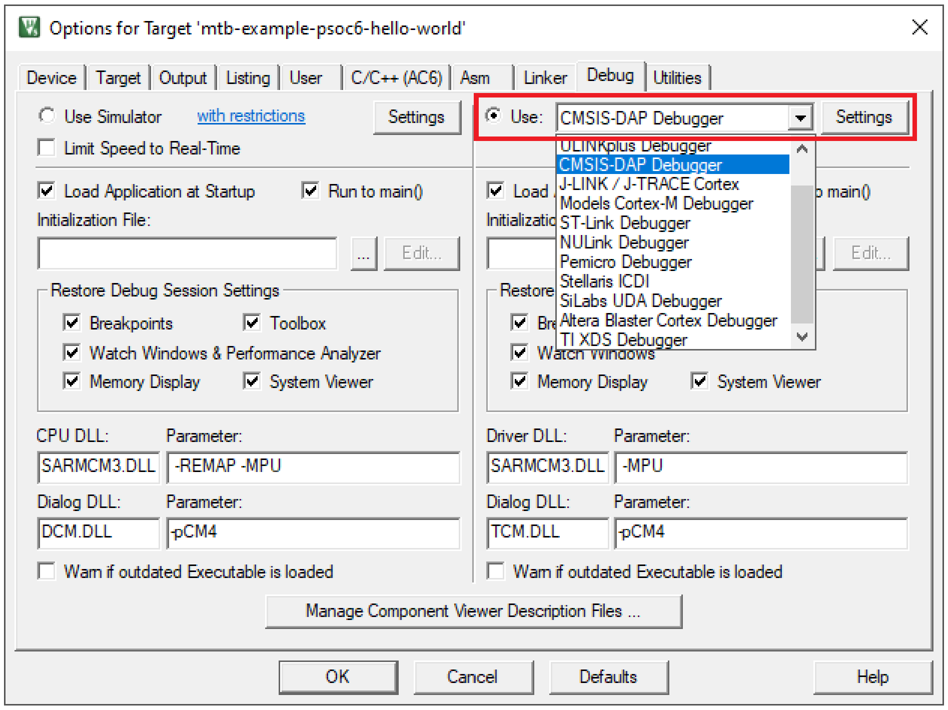
6. On the dialog, select the **C/C++ (AC6)** tab.
   - Check that the **Language C** version was automatically set to c99.
   - Select "AC5-like warnings" in the **Warnings** drop-down list.
   - Select "-Os balanced" in the **Optimization** drop-down list.


Options for Target 'mtb-example-psoc6-hello-world'
Device Target Output Listing User C/C++ (AC6) Asm Linker Debug Utilities
Preprocessor Symbols
Define:
Undefine:
Language / Code Generation
Execute-only Code
Turn Wa
Link-Time Optimization
Split Load and Store Multiple
pendent
Read-Write Position Independent
use RTTI
No Auto Includes
<unspecified>
No Warnings
All Warnings
AC5-like Warnings
MISRA Compatible
OK Cancel Defaults Help

Warnings: AC5-like Warnings
Optimization: -Os balanced
Language C: c99
Language C++: c++98
[✓] Short enums/wchar
[✓] One ELF Section per Function
Include Paths: ..\mtb-example-psoc6-hello-world;..\libs
Misc Controls:
Compiler control string:

```
-xc -std=c99 -target=arm-arm-none-eabi -mcpu=cortex-m4 -mfpu=fpv4-sp-d16 -mfloat-abi=hard -c
-fno-rtti -funsigned-char -fshort-enums -fshort-wchar
```

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



Options for Target 'mtb-example-psoc6-hello-world'

Device | Target | Output | Listing | User | C/C++ (AC6) | Asm | Linker | Debug | Utilities

○ Use Simulator with restrictions — Settings
☐ Limit Speed to Real-Time
☑ Load Application at Startup ☑ Run to main()
Initialization File:
Restore Debug Session Settings
☑ Breakpoints ☑ Toolbox
☑ Watch Windows & Performance Analyzer
☑ Memory Display ☑ System Viewer
CPU DLL: SARMCM3.DLL Parameter: -REMAP -MPU
Dialog DLL: DCM.DLL Parameter: -pCM4
☐ Warn if outdated Executable is loaded

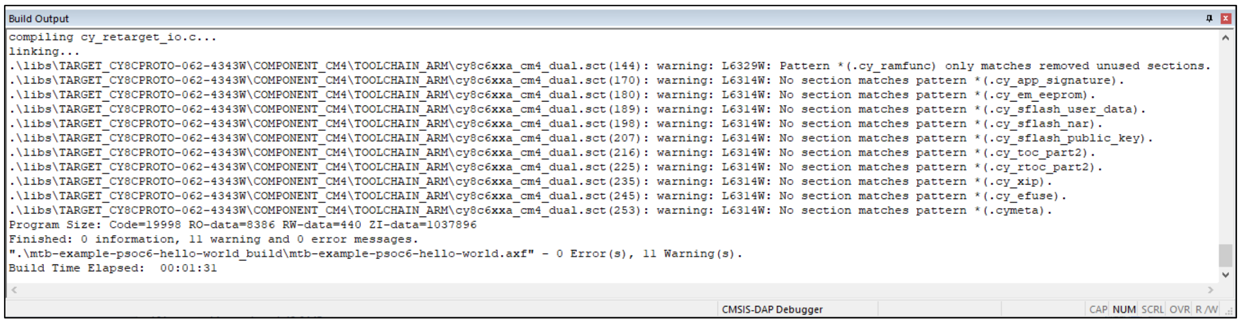
● Use: CMSIS-DAP Debugger — Settings
ULINKplus Debugger
CMSIS-DAP Debugger
J-LINK / J-TRACE Cortex
Models Cortex-M Debugger
ST-Link Debugger
NULink Debugger
Pemicro Debugger
Stellaris ICDI
SiLabs UDA Debugger
Altera Blaster Cortex Debugger
TI XDS Debugger
☑ Load ... main()
Initializatio...
Restore ...
☑ Bre...
☑ Watch Windows
☑ Memory Display ☑ System Viewer
Driver DLL: SARMCM3.DLL Parameter: -MPU
Dialog DLL: TCM.DLL Parameter: -pCM4
☐ Warn if outdated Executable is loaded

Manage Component Viewer Description Files ...

OK | Cancel | Defaults | Help

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

9. Select **Project > Build target**.



```
Build Output
compiling cy_retarget_io.c...
linking...
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(144): warning: L6329W: Pattern *(.cy_ramfunc) only matches removed unused sections.
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(170): warning: L6314W: No section matches pattern *(.cy_app_signature).
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(180): warning: L6314W: No section matches pattern *(.cy_em_eeprom).
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(189): warning: L6314W: No section matches pattern *(.cy_sflash_user_data).
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(198): warning: L6314W: No section matches pattern *(.cy_sflash_nar).
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(207): warning: L6314W: No section matches pattern *(.cy_sflash_public_key).
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(216): warning: L6314W: No section matches pattern *(.cy_toc_part2).
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(225): warning: L6314W: No section matches pattern *(.cy_rtoc_part2).
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(235): warning: L6314W: No section matches pattern *(.cy_xip).
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(245): warning: L6314W: No section matches pattern *(.cy_efuse).
.\libs\TARGET_CY8CPROTO-062-4343W\COMPONENT_CM4\TOOLCHAIN_ARM\cy8c6xxa_cm4_dual.sct(253): warning: L6314W: No section matches pattern *(.cymeta).
Program Size: Code=19998 RO-data=8386 RW-data=440 ZI-data=1037896
Finished: 0 information, 11 warning and 0 error messages.
".\mtb-example-psoc6-hello-world_build\mtb-example-psoc6-hello-world.axf" - 0 Error(s), 11 Warning(s).
Build Time Elapsed:  00:01:31
```

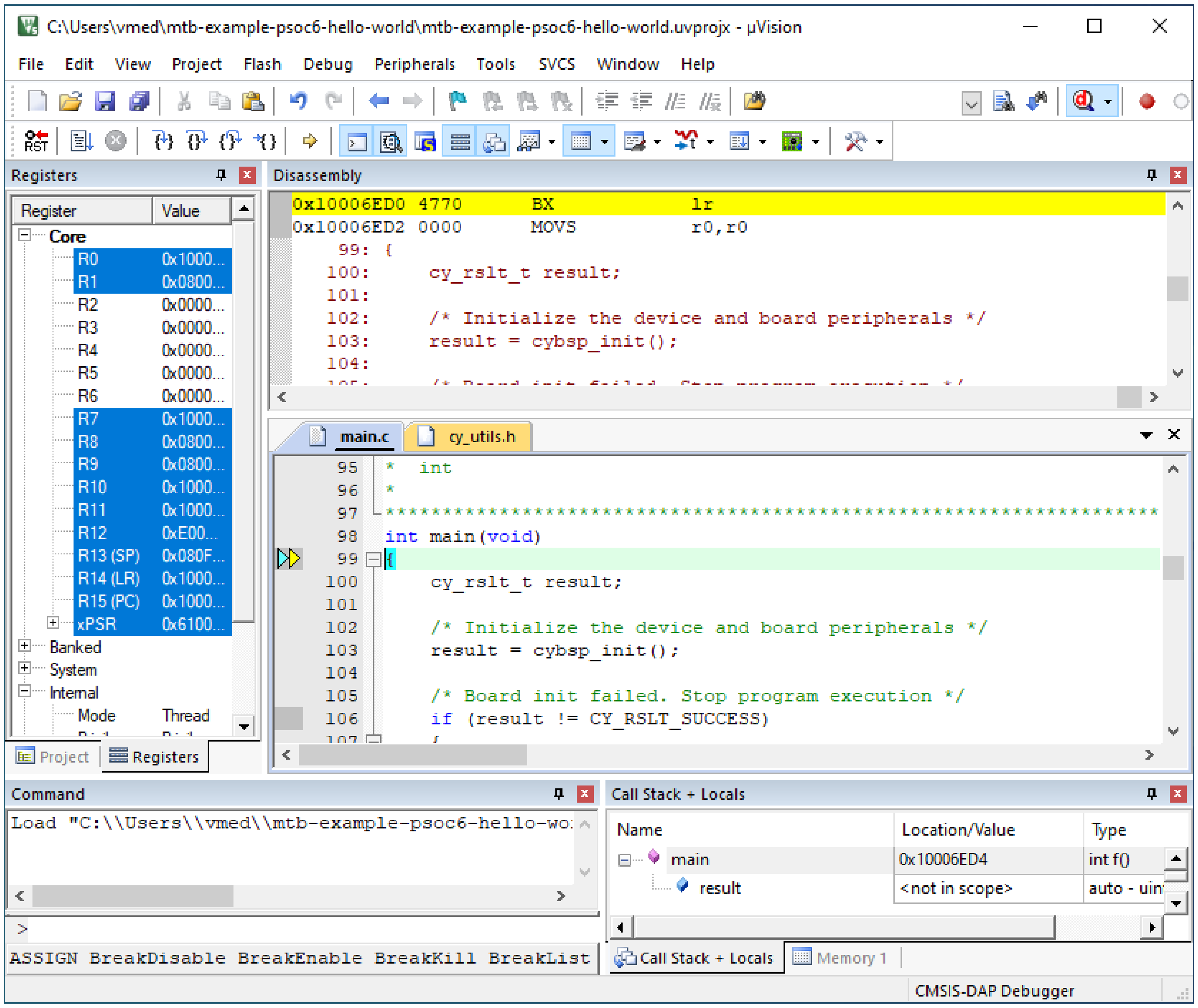
To suppress the linker warnings about unused sections defined in the linker scripts, add “6314,6329” to the **Disable Warnings** setting in the Project Linker Options.

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

11. 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_2.3/fw-loader/bin/

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


C:\Users\vmed\mtb-example-psoc6-hello-world\mtb-example-psoc6-hello-world.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Registers
Disassembly
main.c
cy_utils.h
Command
Call Stack + Locals
Project
Registers
Memory 1
CMSIS-DAP Debugger

Register	Value
Core	
R0	0x1000...
R1	0x0800...
R2	0x0000...
R3	0x0000...
R4	0x0000...
R5	0x0000...
R6	0x0000...
R7	0x1000...
R8	0x0800...
R9	0x0800...
R10	0x1000...
R11	0x1000...
R12	0xE00...
R13 (SP)	0x080F...
R14 (LR)	0x1000...
R15 (PC)	0x1000...
xPSR	0x6100...
Banked	
System	
Internal	
Mode	Thread

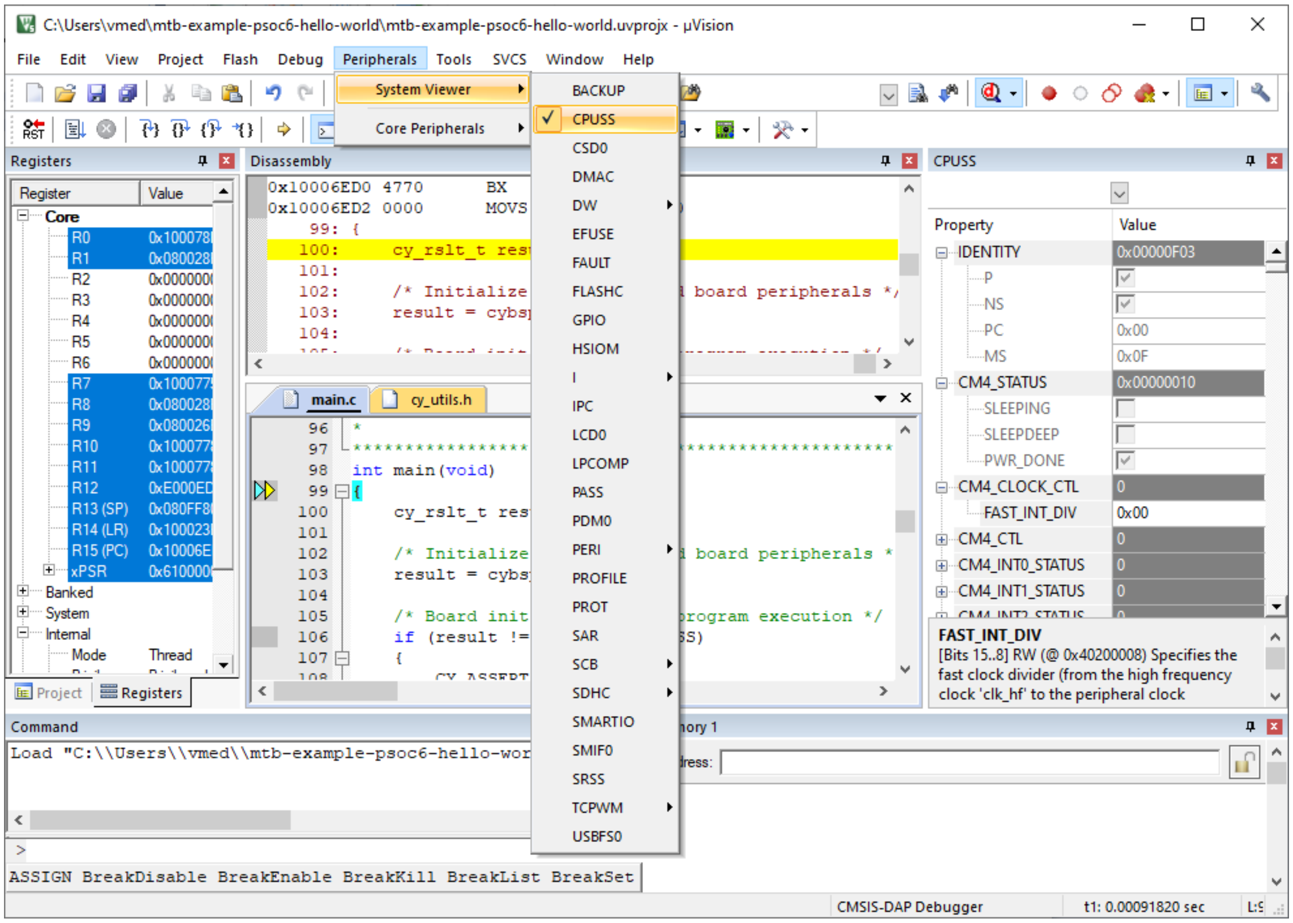
```
0x10006ED0 4770      BX         lr
0x10006ED2 0000      MOVS       r0,r0
    99: {
   100:     cy_rslt_t result;
   101:
   102:     /* Initialize the device and board peripherals */
   103:     result = cybsp_init();
   104:
```

```
 *  int
 *
*************************************************************************
int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init();

    /* Board init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
```

```
Load "C:\\Users\\vmed\\mtb-example-psoc6-hello-wo
>
ASSIGN BreakDisable BreakEnable BreakKill BreakList
```

Name	Location/Value	Type
main	0x10006ED4	int f()
result	<not in scope>	auto - uin

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



Register	Value
Core	
R0	0x100078
R1	0x080028
R2	0x000000
R3	0x000000
R4	0x000000
R5	0x000000
R6	0x000000
R7	0x10007
R8	0x080028
R9	0x080026
R10	0x10007
R11	0x10007
R12	0xE000ED
R13 (SP)	0x080FF8
R14 (LR)	0x100023
R15 (PC)	0x10006E
xPSR	0x610000
Banked	
System	
Internal	
Mode	Thread

```
0x10006ED0 4770      BX
0x10006ED2 0000      MOVS
    99: {
   100:         cy_rslt_t resu
   101:
   102:         /* Initialize                    board peripherals */
   103:         result = cybs
   104:
```

```
 *
 ***************************************************************
int main(void)
{
    cy_rslt_t res

    /* Initialize                    board peripherals */
    result = cybs

    /* Board init                    program execution */
    if (result !=                    SS)
    {
        CY_ASSERT
```

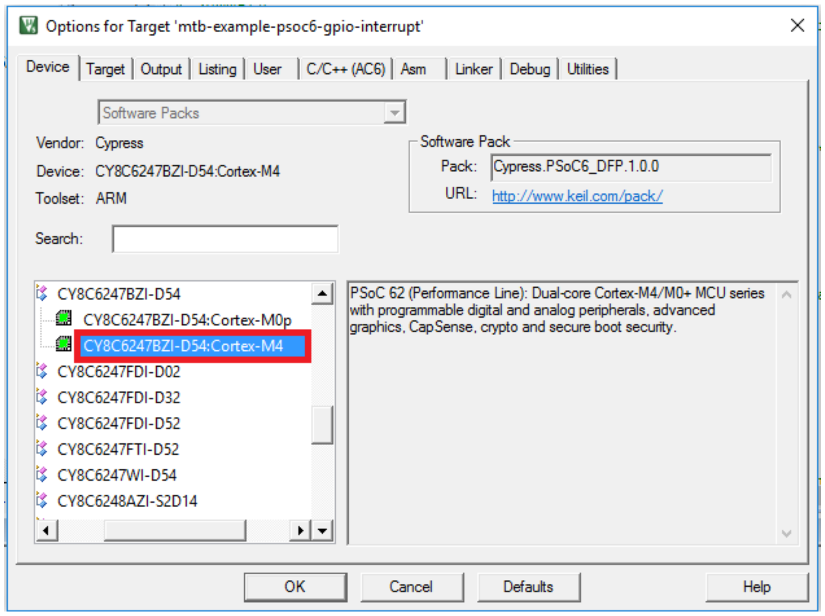
Property	Value
IDENTITY	0x00000F03
P	✓
NS	✓
PC	0x00
MS	0x0F
CM4_STATUS	0x00000010
SLEEPING	
SLEEPDEEP	
PWR_DONE	✓
CM4_CLOCK_CTL	0
FAST_INT_DIV	0x00
CM4_CTL	0
CM4_INT0_STATUS	0
CM4_INT1_STATUS	0
CM4_INT2_STATUS	0

FAST_INT_DIV
[Bits 15..8] RW (@ 0x40200008) Specifies the fast clock divider (from the high frequency clock 'clk_hf' to the peripheral clock

```
Load "C:\\Users\\vmed\\mtb-example-psoc6-hello-wor
>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet
```

7.5.2.1 *To Use KitProg3/MiniProg4, 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:



Options for Target 'mtb-example-psoc6-gpio-interrupt'

Vendor: Cypress
Device: CY8C6247BZI-D54:Cortex-M4
Toolset: ARM

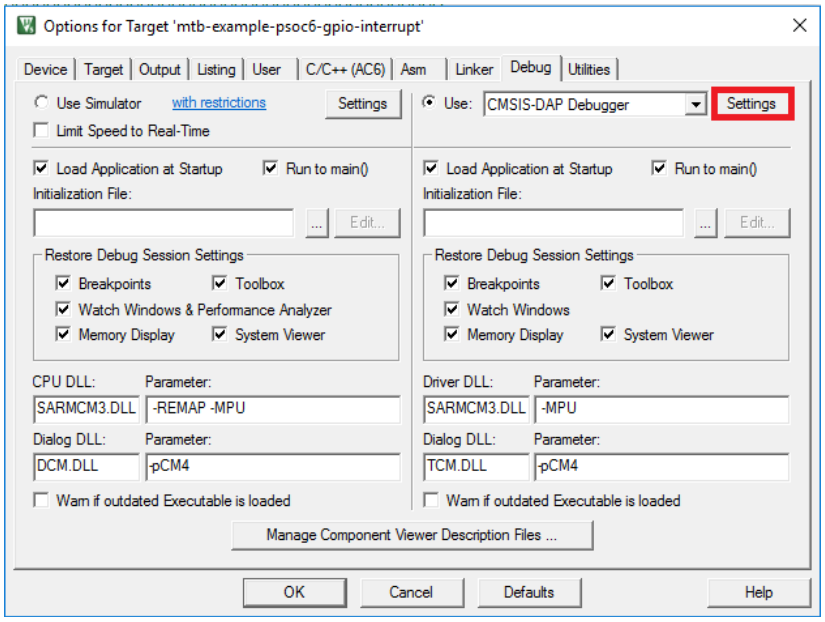
Software Pack
Pack: Cypress.PSoC6_DFP.1.0.0
URL: http://www.keil.com/pack/

- CY8C6247BZI-D54
  - CY8C6247BZI-D54:Cortex-M0p
  - CY8C6247BZI-D54:Cortex-M4
- CY8C6247FDI-D02
- CY8C6247FDI-D32
- CY8C6247FDI-D52
- CY8C6247FTI-D52
- CY8C6247WI-D54
- CY8C6248AZI-S2D14

PSoC 62 (Performance Line): Dual-core Cortex-M4/M0+ MCU series with programmable digital and analog peripherals, advanced graphics, CapSense, crypto and secure boot security.

Device | Target | Output | Listing | User | C/C++ (AC6) | Asm | Linker | Debug | Utilities
Software Packs
Search:
OK
Cancel
Defaults
Help

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



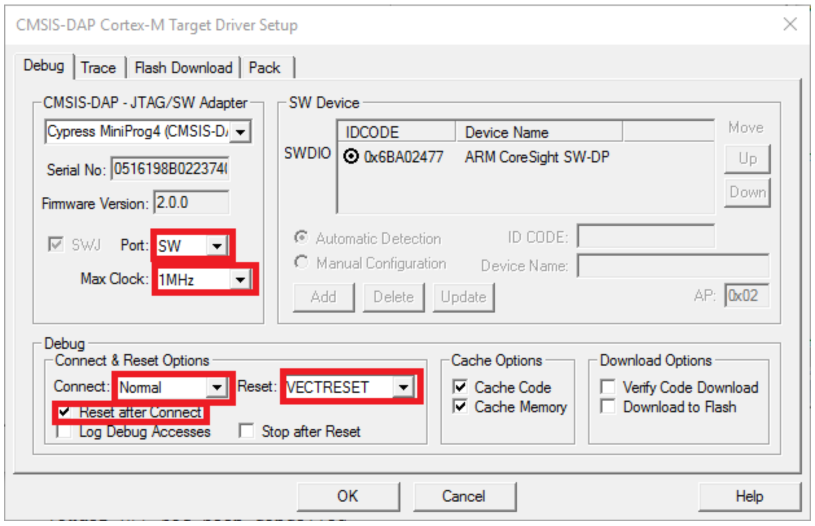
Options for Target 'mtb-example-psoc6-gpio-interrupt'

Use Simulator with restrictions — Settings
Limit Speed to Real-Time
Use: CMSIS-DAP Debugger — Settings

☑ Load Application at Startup ☑ Run to main()	☑ Load Application at Startup ☑ Run to main()
Initialization File:	Initialization File:
Restore Debug Session Settings: ☑ Breakpoints ☑ Toolbox ☑ Watch Windows & Performance Analyzer ☑ Memory Display ☑ System Viewer	Restore Debug Session Settings: ☑ Breakpoints ☑ Toolbox ☑ Watch Windows ☑ Memory Display ☑ System Viewer
CPU DLL: SARMCM3.DLL Parameter: -REMAP -MPU	Driver DLL: SARMCM3.DLL Parameter: -MPU
Dialog DLL: DCM.DLL Parameter: -pCM4	Dialog DLL: TCM.DLL Parameter: -pCM4
☐ Warn if outdated Executable is loaded	☐ Warn if outdated Executable is loaded

Device | Target | Output | Listing | User | C/C++ (AC6) | Asm | Linker | Debug | Utilities
Edit...
Manage Component Viewer Description Files ...
OK
Cancel
Defaults
Help

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


CMSIS-DAP Cortex-M Target Driver Setup
Debug | Trace | Flash Download | Pack
Move
Up
Down
Add
Delete
Update
OK
Cancel
Help

CMSIS-DAP - JTAG/SW Adapter: Cypress MiniProg4 (CMSIS-D/
Serial No: 0516198B022374(
Firmware Version: 2.0.0
✓ SWJ Port: SW
Max Clock: 1MHz

SW Device

	IDCODE	Device Name
SWDIO	● 0x6BA02477	ARM CoreSight SW-DP

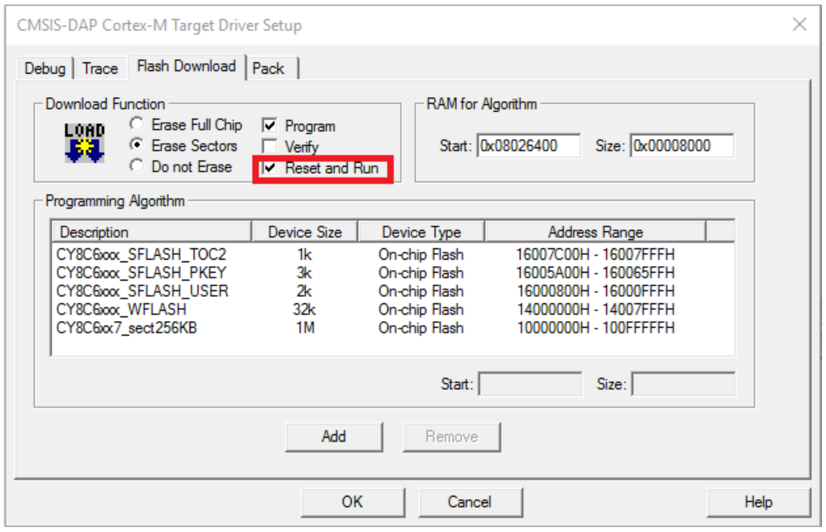
● Automatic Detection ID CODE:
○ Manual Configuration Device Name:
AP: 0x02

Debug
Connect & Reset Options
Connect: Normal Reset: VECTRESET
✓ Reset after Connect
☐ Log Debug Accesses ☐ Stop after Reset

Cache Options
✓ Cache Code
✓ Cache Memory

Download Options
☐ Verify Code Download
☐ Download to Flash

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


CMSIS-DAP Cortex-M Target Driver Setup
Debug | Trace | Flash Download | Pack
LOAD
Add
Remove
OK
Cancel
Help

Download Function
○ Erase Full Chip ✓ Program
● Erase Sectors ☐ Verify
○ Do not Erase ✓ Reset and Run

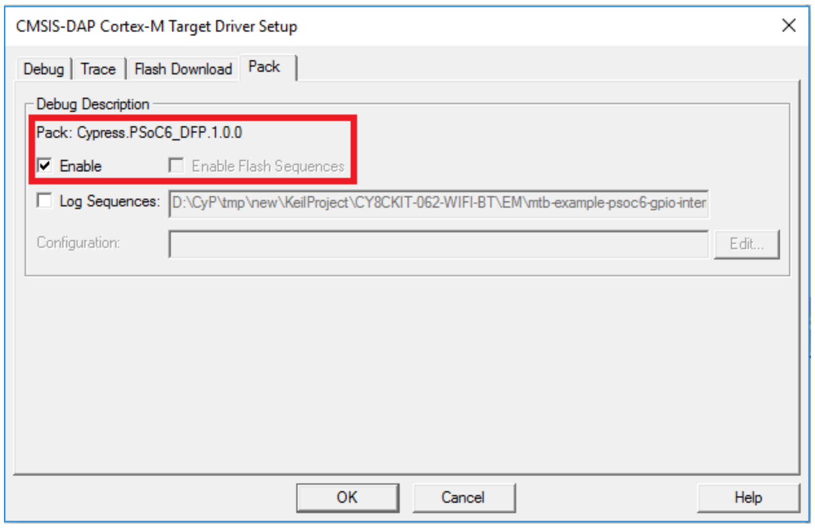
RAM for Algorithm
Start: 0x08026400 Size: 0x00008000

Programming Algorithm

Description	Device Size	Device Type	Address Range
CY8C6xxx_SFLASH_TOC2	1k	On-chip Flash	16007C00H - 16007FFFH
CY8C6xxx_SFLASH_PKEY	3k	On-chip Flash	16005A00H - 160065FFH
CY8C6xxx_SFLASH_USER	2k	On-chip Flash	16000800H - 16000FFFH
CY8C6xxx_WFLASH	32k	On-chip Flash	14000000H - 14007FFFH
CY8C6xx7_sect256KB	1M	On-chip Flash	10000000H - 100FFFFFH

Start: Size:

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


CMSIS-DAP Cortex-M Target Driver Setup

Debug | Trace | Flash Download | Pack

Debug Description

Pack: Cypress.PSoC6_DFP.1.0.0

[✓] Enable [ ] Enable Flash Sequences

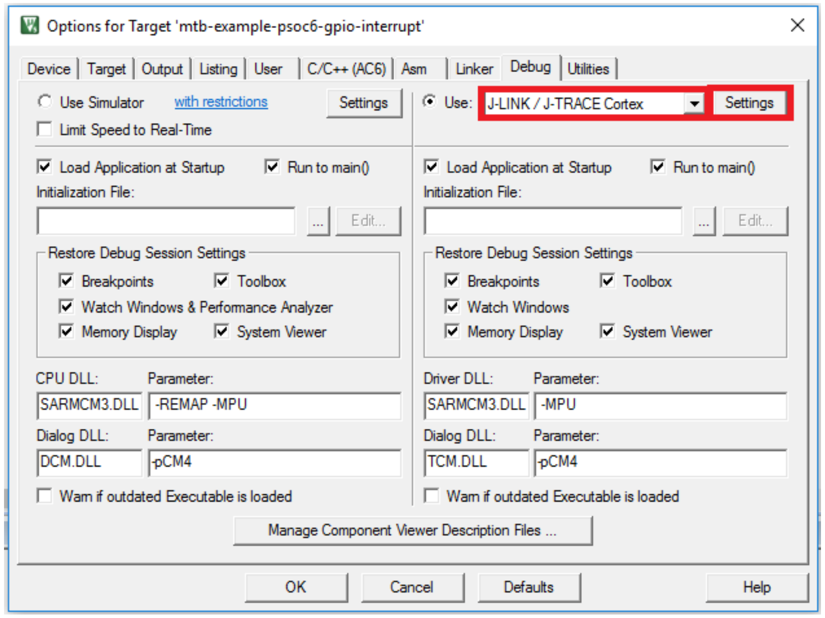
[ ] Log Sequences: D:\CyP\tmp\new\KeilProject\CY8CKIT-062-WIFI-BT\EM\mtb-example-psoc6-gpio-inter

Configuration:

OK Cancel Help

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”:


Options for Target 'mtb-example-psoc6-gpio-interrupt'

Device | Target | Output | Listing | User | C/C++ (AC6) | Asm | Linker | Debug | Utilities

( ) Use Simulator with restrictions Settings

[ ] Limit Speed to Real-Time

[✓] Load Application at Startup [✓] Run to main()

Initialization File:

Restore Debug Session Settings

[✓] Breakpoints [✓] Toolbox

[✓] Watch Windows & Performance Analyzer

[✓] Memory Display [✓] System Viewer

CPU DLL: SARMCM3.DLL Parameter: -REMAP -MPU

Dialog DLL: DCM.DLL Parameter: -pCM4

[ ] Warn if outdated Executable is loaded

(•) Use: J-LINK / J-TRACE Cortex Settings

[✓] Load Application at Startup [✓] Run to main()

Initialization File:

Restore Debug Session Settings

[✓] Breakpoints [✓] Toolbox

[✓] Watch Windows

[✓] Memory Display [✓] System Viewer

Driver DLL: SARMCM3.DLL Parameter: -MPU

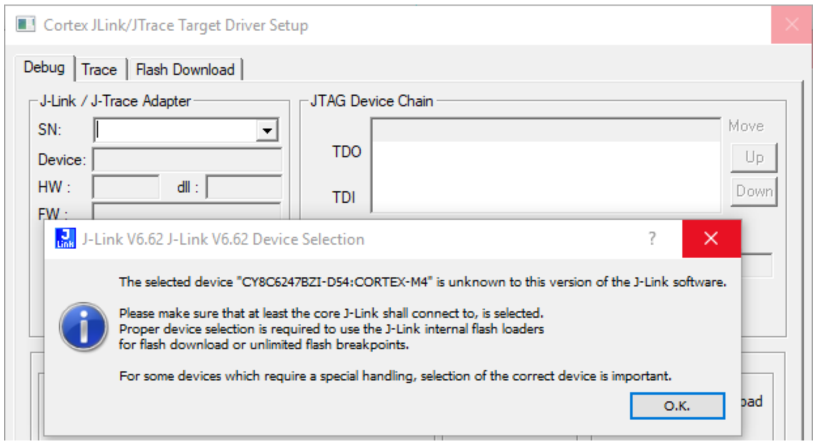
Dialog DLL: TCM.DLL Parameter: -pCM4

[ ] Warn if outdated Executable is loaded

Manage Component Viewer Description Files ...

OK Cancel Defaults Help

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

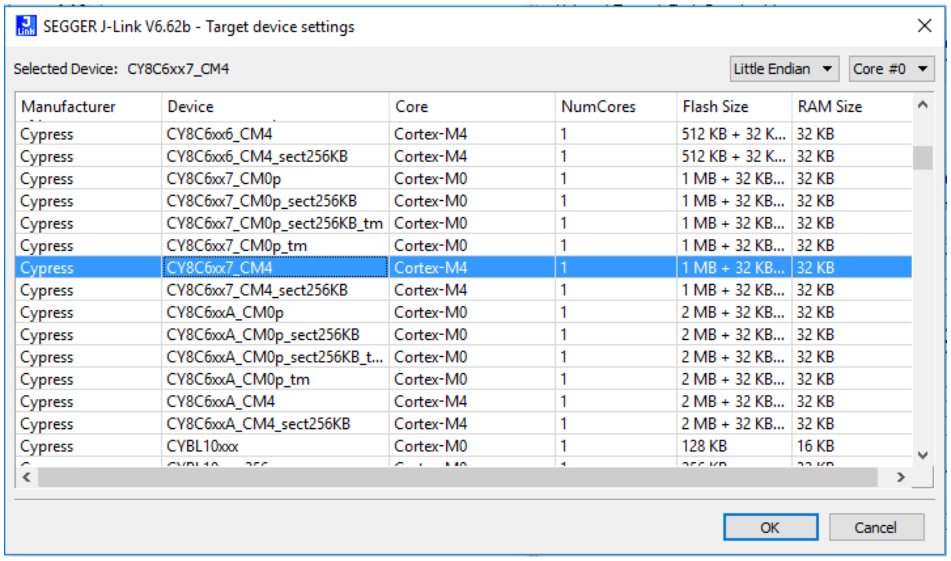


The selected device "CY8C6247BZI-D54:CORTEX-M4" is unknown to this version of the J-Link software.

Please make sure that at least the core J-Link shall connect to, is selected.
Proper device selection is required to use the J-Link internal flash loaders
for flash download or unlimited flash breakpoints.

For some devices which require a special handling, selection of the correct device is important.

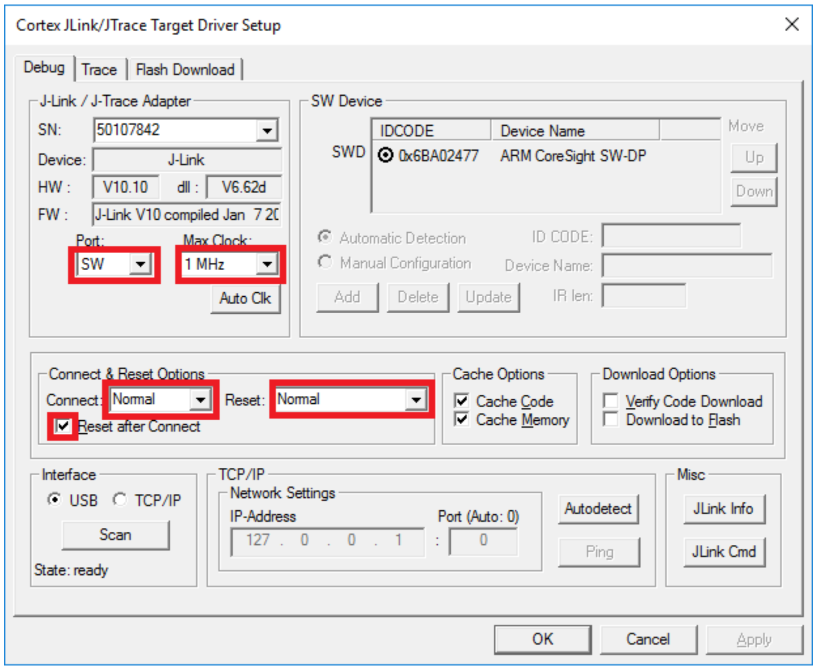
4. Select appropriate target in Wizard:



Selected Device: CY8C6xx7_CM4

Manufacturer	Device	Core	NumCores	Flash Size	RAM Size
Cypress	CY8C6xx6_CM4	Cortex-M4	1	512 KB + 32 K...	32 KB
Cypress	CY8C6xx6_CM4_sect256KB	Cortex-M4	1	512 KB + 32 K...	32 KB
Cypress	CY8C6xx7_CM0p	Cortex-M0	1	1 MB + 32 KB...	32 KB
Cypress	CY8C6xx7_CM0p_sect256KB	Cortex-M0	1	1 MB + 32 KB...	32 KB
Cypress	CY8C6xx7_CM0p_sect256KB_tm	Cortex-M0	1	1 MB + 32 KB...	32 KB
Cypress	CY8C6xx7_CM0p_tm	Cortex-M0	1	1 MB + 32 KB...	32 KB
Cypress	CY8C6xx7_CM4	Cortex-M4	1	1 MB + 32 KB...	32 KB
Cypress	CY8C6xx7_CM4_sect256KB	Cortex-M4	1	1 MB + 32 KB...	32 KB
Cypress	CY8C6xxA_CM0p	Cortex-M0	1	2 MB + 32 KB...	32 KB
Cypress	CY8C6xxA_CM0p_sect256KB	Cortex-M0	1	2 MB + 32 KB...	32 KB
Cypress	CY8C6xxA_CM0p_sect256KB_t...	Cortex-M0	1	2 MB + 32 KB...	32 KB
Cypress	CY8C6xxA_CM0p_tm	Cortex-M0	1	2 MB + 32 KB...	32 KB
Cypress	CY8C6xxA_CM4	Cortex-M4	1	2 MB + 32 KB...	32 KB
Cypress	CY8C6xxA_CM4_sect256KB	Cortex-M4	1	2 MB + 32 KB...	32 KB
Cypress	CYBL10xxx	Cortex-M0	1	128 KB	16 KB

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


Cortex JLink/JTrace Target Driver Setup

Debug | Trace | Flash Download

J-Link / J-Trace Adapter
SN: 50107842
Device: J-Link
HW: V10.10 dll: V6.62d
FW: J-Link V10 compiled Jan 7 20
Port: SW
Max Clock: 1 MHz
Auto Clk

SW Device

	IDCODE	Device Name
SWD	0x6BA02477	ARM CoreSight SW-DP

Move Up Down
Automatic Detection ID CODE:
Manual Configuration Device Name:
Add Delete Update IR len:

Connect & Reset Options
Connect: Normal Reset: Normal
☑ Reset after Connect

Cache Options
☑ Cache Code
☑ Cache Memory

Download Options
☐ Verify Code Download
☐ Download to Flash

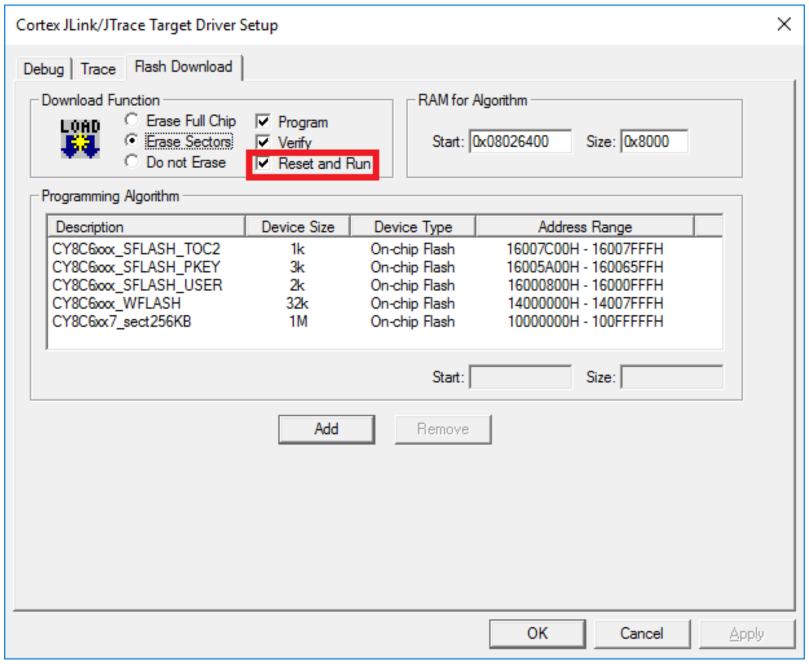
Interface
● USB ○ TCP/IP
Scan
State: ready

TCP/IP
Network Settings
IP-Address: 127 . 0 . 0 . 1 Port (Auto: 0): 0
Autodetect
Ping

Misc
JLink Info
JLink Cmd

OK Cancel Apply

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


Cortex JLink/JTrace Target Driver Setup

Debug | Trace | Flash Download

Download Function
○ Erase Full Chip
● Erase Sectors
○ Do not Erase
☑ Program
☑ Verify
☑ Reset and Run

RAM for Algorithm
Start: 0x08026400 Size: 0x8000

Programming Algorithm

Description	Device Size	Device Type	Address Range
CY8C6xxx_SFLASH_TOC2	1k	On-chip Flash	16007C00H - 16007FFFH
CY8C6xxx_SFLASH_PKEY	3k	On-chip Flash	16005A00H - 160065FFH
CY8C6xxx_SFLASH_USER	2k	On-chip Flash	16000800H - 16000FFFH
CY8C6xxx_WFLASH	32k	On-chip Flash	14000000H - 14007FFFH
CY8C6xx7_sect256KB	1M	On-chip Flash	10000000H - 100FFFFFH

Start: Size:
Add Remove

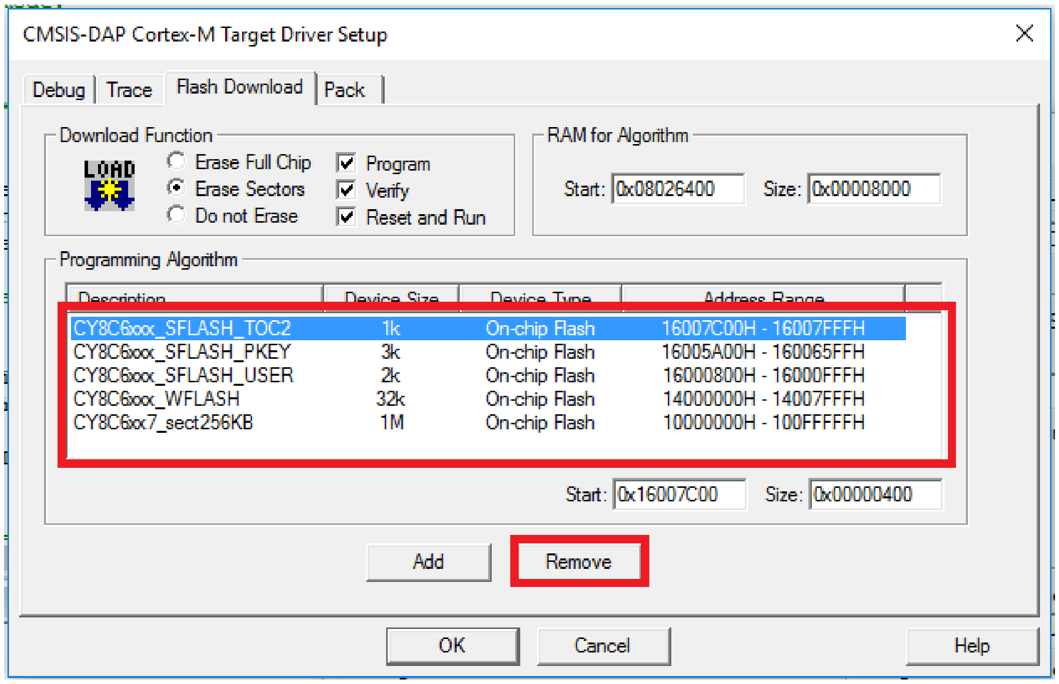
OK Cancel Apply

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:



CMSIS-DAP Cortex-M Target Driver Setup

Debug | Trace | Flash Download | Pack

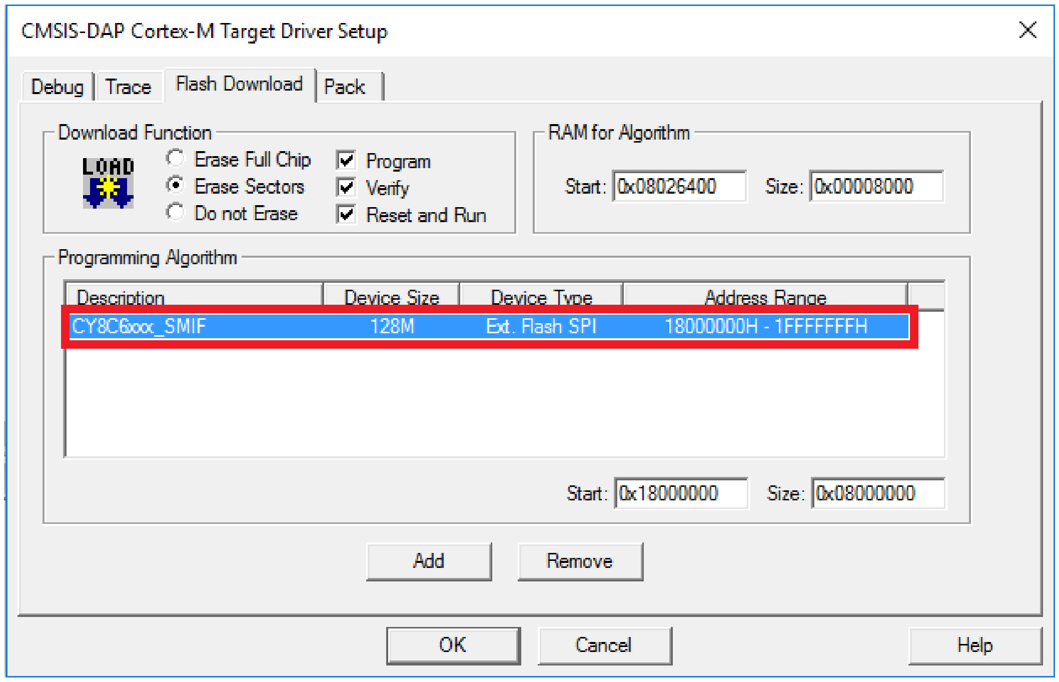
Download Function: ( ) Erase Full Chip, [✓ (•) Erase Sectors], ( ) Do not Erase; [✓] Program, [✓] Verify, [✓] Reset and Run

RAM for Algorithm: Start: 0x08026400 Size: 0x00008000

Programming Algorithm

Description	Device Size	Device Type	Address Range
CY8C6xxx_SFLASH_TOC2	1k	On-chip Flash	16007C00H - 16007FFFH
CY8C6xxx_SFLASH_PKEY	3k	On-chip Flash	16005A00H - 160065FFH
CY8C6xxx_SFLASH_USER	2k	On-chip Flash	16000800H - 16000FFFH
CY8C6xxx_WFLASH	32k	On-chip Flash	14000000H - 14007FFFH
CY8C6xx7_sect256KB	1M	On-chip Flash	10000000H - 100FFFFFH

Start: 0x16007C00 Size: 0x00000400



CMSIS-DAP Cortex-M Target Driver Setup

Debug | Trace | Flash Download | Pack

Download Function: ( ) Erase Full Chip, [✓ (•) Erase Sectors], ( ) Do not Erase; [✓] Program, [✓] Verify, [✓] Reset and Run

RAM for Algorithm: Start: 0x08026400 Size: 0x00008000

Programming Algorithm

Description	Device Size	Device Type	Address Range
CY8C6xxx_SMIF	128M	Ext. Flash SPI	18000000H - 1FFFFFFFH

Start: 0x18000000 Size: 0x08000000

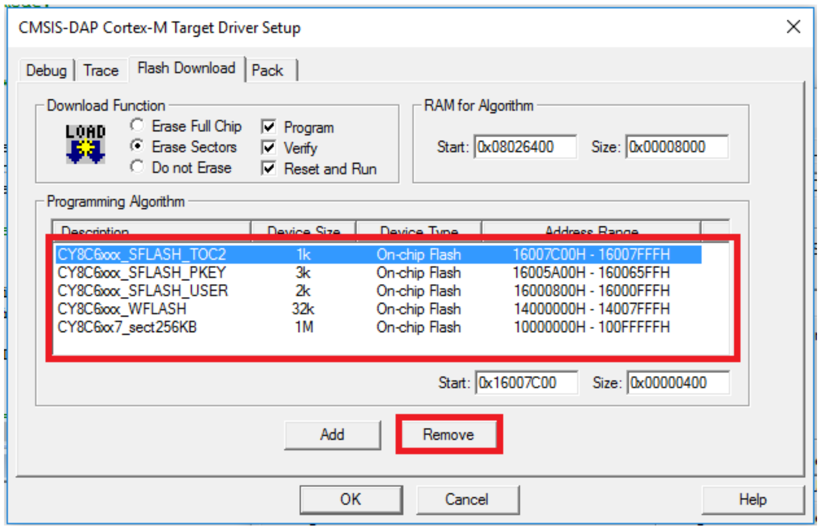
3. Download flash.

   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:



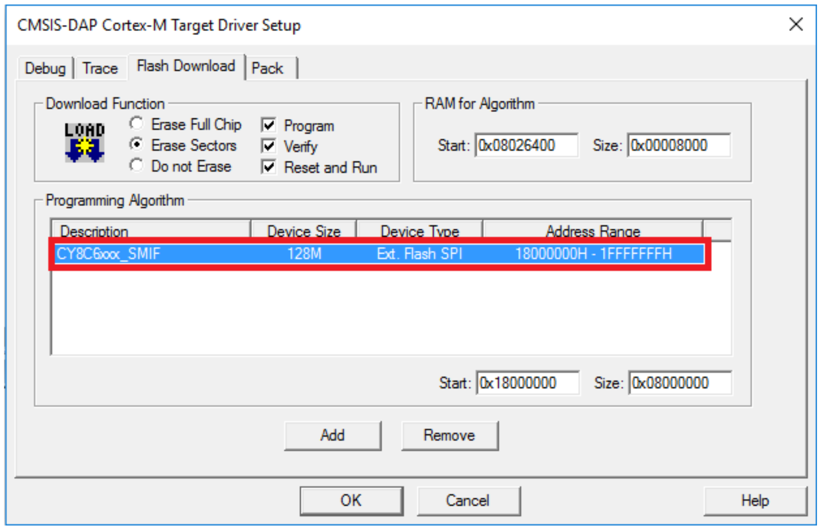
Download Function: ○ Erase Full Chip, ● Erase Sectors, ○ Do not Erase; ☑ Program, ☑ Verify, ☑ Reset and Run

RAM for Algorithm: Start: 0x08026400 Size: 0x00008000

Programming Algorithm

Description	Device Size	Device Type	Address Range
CY8C6xxx_SFLASH_TOC2	1k	On-chip Flash	16007C00H - 16007FFFH
CY8C6xxx_SFLASH_PKEY	3k	On-chip Flash	16005A00H - 160065FFH
CY8C6xxx_SFLASH_USER	2k	On-chip Flash	16000800H - 16000FFFH
CY8C6xxx_WFLASH	32k	On-chip Flash	14000000H - 14007FFFH
CY8C6xx7_sect256KB	1M	On-chip Flash	10000000H - 100FFFFFH

Start: 0x16007C00 Size: 0x00000400



Download Function: ○ Erase Full Chip, ● Erase Sectors, ○ Do not Erase; ☑ Program, ☑ Verify, ☑ Reset and Run

RAM for Algorithm: Start: 0x08026400 Size: 0x00008000

Programming Algorithm

Description	Device Size	Device Type	Address Range
CY8C6xxx_SMIF	128M	Ext. Flash SPI	18000000H - 1FFFFFFFH

Start: 0x18000000 Size: 0x08000000

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

Document Revision History

Document Title: ModusToolbox™ 2.3 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 associated text.
*B	4/1/2020	Fix broken links.
*C	4/29/2020	Fix incorrect link.
*D	8/28/2020	Updates for ModusToolbox 2.2.
*E	9/23/2020	Corrections to Build System and Board Support Packages chapters.
*F	9/29/2020	Added links to KBAs; updated text for cyignore.
*G	10/2/2020	Added details for BTSDK v2.8 BSPs/libraries.
*H	1/14/2021	Updated Manifest chapter and fixed broken links.
*I	3/23/2021	Updates for ModusToolbox 2.3.
*J	5/24/2021	Updated information for creating a custom BSP.