ModusToolbox™ 3.x tools package user guide

ModusToolbox™ tools package version 3.9.0

About this document

A newer version of this document may be available on the web here

Scope and purpose

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

- Chapter 1 describes the various aspects of ModusToolbox™ software.
- Chapter 2 provides instructions for getting started using the ModusToolbox™ tools.
- Chapter 3 provides instructions for working with an application after it has been created.
- Chapter 4 discusses using version control systems and methods to share an application.
- Chapter 5 describes the ModusToolbox™ build system.
- Chapter 6 covers different aspects of the ModusToolbox™ board support packages (BSPs).
- Chapter 7 explains the ModusToolbox™ manifest files and how to use them with BSPs, libraries, and code examples.
- Chapter 8 provides instructions for using a ModusToolbox™ application with various third-party tools.

Document conventions

Convention	Explanation
Bold	Emphasizes heading levels, column headings, menus and sub-menus
Italics	Denotes file names and paths.
`Monospace`	Denotes APIs, functions, interrupt handlers, events, data types, error handlers, file/folder names, directories, command line inputs, code snippets
File > New	Indicates that a cascading sub-menu opens when you select a menu item

Abbreviations and definitions

This section provides a list of various terms and definitions used in the ModusToolbox™ ecosystem.

Term	Definition
Application	A ModusToolbox™ Application is required to have a *Makefile* that supports the standard make targets (for example, `get_app_info`, `build`, `getlibs`). There is generally one directory that contains one or more Projects that are meant to build and work together. The Application Makefile forwards the make targets along to the Projects in the appropriate order.
Board (e.g.,: EVK/DVK/kit)	This is a physical collection of electronics that have been grouped together to show of capabilities of a chip or an end product created by a customer that incorporates Infineon silicon. Boards may be developed by Infineon, 3rd party partners such as Future, or the customer themselves. This is a Target for which a ModusToolbox™ Application can be built.
Board Support Package (BSP)	A library that you may customize, which adds support to build and boot a Target board and related devices. This includes source code, headers, resources, linker scripts, startup code, and they define common aliases for resources like GPIOs to aid in portability. All ModusToolbox™ Applications require at least one BSP.

About this document

Term	Definition
BSP Configurator	A specific type of configurator that is used to configure aspects of the system that are specific to the BSP. Since the BSP contains the base microcontroller, the Device Configurator, as an example, would be a BSP configurator.
Chip/device	This is a specific piece of silicon that Infineon sells.
Code Example	This is the starting point / template for an Application. It is typically a Git repo, but it can also be another directory.
Configurator	GUI and/or CLI tools that simplify the configuration of a run-time asset (chip, board, or middleware). There are various types of configurators, and they generally create code or data structures that become part of the target application.
Device Configurator	A BSP Configurator that is used to configure hardware on the main Target device. It allows configuration of device GPIOs, peripherals, clocks, DMA, etc. This is generally the starting configurator that can also be used to open other BSP Configurators.
Device database (device-db)	Contains a set of devices that can be used by various tools for Application and BSP creation.
Driver	Low-level firmware that enables a specific hardware peripheral.
Firmware	Program code that runs on a Chip as a result of a building the Application and Programming (flash).
Hardware Abstraction Layer (HAL)	A set of firmware Drivers that provide an interface to low-level hardware that is consistent across most Infineon devices.
Kit	See Board.
Lastest Locking	A mechanism that tracks and locks versions of libraries to specific versions for a given Application when that Application is created. Older versions of the tools package used a file named *locking-commit.log*. Version 3.3 and later use a file named *assetlocks.json*.
Library Configurator	A specific type of Configurator that is used to configure aspects of the system that are specific to the project, but not specific to the BSP. The Bluetooth® Configurator that is used to configure items such as advertising settings, services and characteristics that are part of a project is an example of a Library Configurator. This type of configurator is generally tightly coupled to a middleware library (i.e. Bluetooth®, etc.).
Manifests	This is a set of XML files that describe the downloadable content that is available. This includes code examples, BSPs, and middleware libraries.
Middleware	A set of related firmware, distributed as a library of source code, headers, pre-compiled code, and resources that is designed to be used by a variety of applications.
Module	A piece of silicon or PCB developed by Infineon or a 3rd party partner that contains one or more Infineon chips that have been qualified to work together.
ModusToolbox™ packs	An additional set of tools, manifests, and firmware assets that are installed as a group through Infineon Developer Center to extend the capabilities of the basic ModusToolbox™ tools package. There are Technology Packs that usually add some sort of functionality and Early Access Packs that enable early access to new devices. Early Access Packs are usually restricted to select customers.
ModusToolbox™ Setup program	A stand-alone tool used to install ModusToolbox™ packages.

About this document

Term	Definition
ModusToolbox™ tools package	The basic set of tools, build system, and IDE support installed from a package for Windows, macOS, and Linux.
mtb-shared directory	Provides the default and added library code used for the Applications within a given Workspace.
Peripheral Driver Library (PDL)	A set of low level, device specific, firmware Drivers that provide access to hardware peripherals
Program	This term can mean different things. When you download an Application onto a device, it is often referred to as Programming the device. In the context of a Tool, program can refer to that Tool's executable file and that is described by the *props.json* file found in that Tool's directory.
Programming/debugger probes	Hardware connected to the kit/device that provides protocols to communicate between the device and the software. These include KitProg3, MiniProg4, I-Jet, J-Link, etc.
Project	For a ModusToolbox™ single-core Application, there is only one Project and so the terms Application and Project are combined into a single entity. A ModusToolbox™ Project contains a makefile that uses the "core-make" asset to build a set of source files into a linked library or ELF image. For a ModusToolbox™ multi-core application, there are Projects for each core. So, a Project is a sub-unit of the Application, and each project is associated with a particular core of the targeted Board. The term "project" has slightly different meanings in the context of 3rd party tools. It will be disambiguated where necessary (for example, "Eclipse project").
props.json	A file that contains properties for the given Tool.
Target	The BSP for which a project is currently building.
Tool	This is any program that is installed as part of the ModusToolbox™ tools package or through the ModusToolbox™ Setup program to provide build or configuration functionality. Specifically, a tool is a directory in the *tools_x.y* installation directory or under the tools subdirectory of a ModusToolbox™ Pack and that contains a valid *props.json* file. Examples of tools include: Dashboard Project Creator Library Manager BSP Assistant Configurators
Workspace	A collection of files and directories that contain one or more Applications and a common "mtb_shared" directory for shared libraries. These are defined differently by different IDEs.

Table of contents

About this document . . . 1
Table of contents . . . 4
1 Introduction . . . 6
1.1 What is ModusToolbox™ software? . . . 6
1.2 Tools package . . . 6
1.3 Run-time software . . . 12
1.4 Product versioning . . . 14
2 Getting started . . . 19
2.1 Install and configure software . . . 19
2.2 Launch Dashboard . . . 20
2.3 Create application from template . . . 20
2.4 Understand application structures . . . 22
2.5 Build and program . . . 26
3 Updating the example application . . . 28
3.1 Update libraries . . . 28
3.2 Create/edit BSPs . . . 29
3.3 Configure settings for devices, peripherals, and libraries . . . 29
3.4 Write application code . . . 31
3.5 Debug the application . . . 31
4 Version Control and sharing applications . . . 33
4.1 Files to include/exclude . . . 33
4.2 Basic sharing steps . . . 33
5 ModusToolbox™ build system . . . 35
5.1 Overview . . . 35
5.2 make help . . . 35
5.3 make getlibs . . . 36
5.4 BSPs . . . 36
5.5 Environment variables . . . 36
5.6 Adding source files . . . 37
5.7 Pre-builds and post-builds . . . 39
5.8 Available make targets . . . 39
5.9 Available make variables . . . 42
5.10 Tools targets . . . 48
6 Board support packages . . . 49
6.1 Overview . . . 49
6.2 What's in a BSP . . . 49
6.3 Creating your own BSP . . . 51

Table of contents

7 Manifest files 53
7.1 Overview 53
7.2 Create your own manifest 54
7.3 Local content storage 56

8 Using applications with third-party tools 57
8.1 Using supported IDEs 57
8.2 Multi-core debugging 58
8.3 Generating files for XMC™ Simulator tool 59

Disclaimer 60

1 Introduction

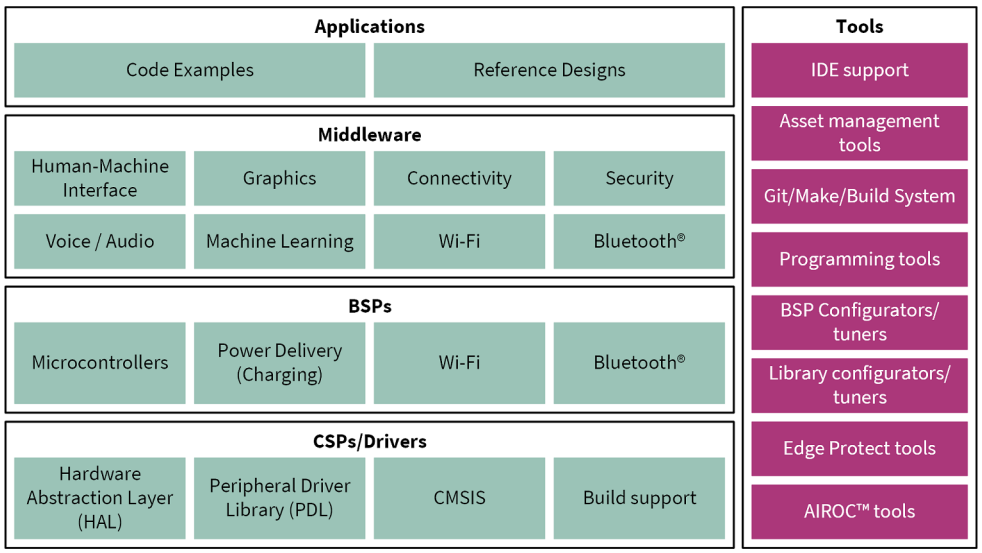
This chapter provides an overview of the ModusToolbox™ software environment, which provides support for many types of devices and ecosystems. The major headings in this chapter include:

- What is ModusToolbox™ software?
- Tools package
- Run-time software
- Product versioning

1.1 What is ModusToolbox™ software?

ModusToolbox™ software 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.

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



ModusToolbox™ software 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™ Wi-Fi and Bluetooth®, security, and various other features.

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.

1.2 Tools package

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™ software is not a monolithic, proprietary software tool that dictates the use of any particular IDE.

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

- Base tools package directory structure
- Tool descriptions

The installer does not include Code examples or Libraries (middleware), but it does provide the tools to access them.

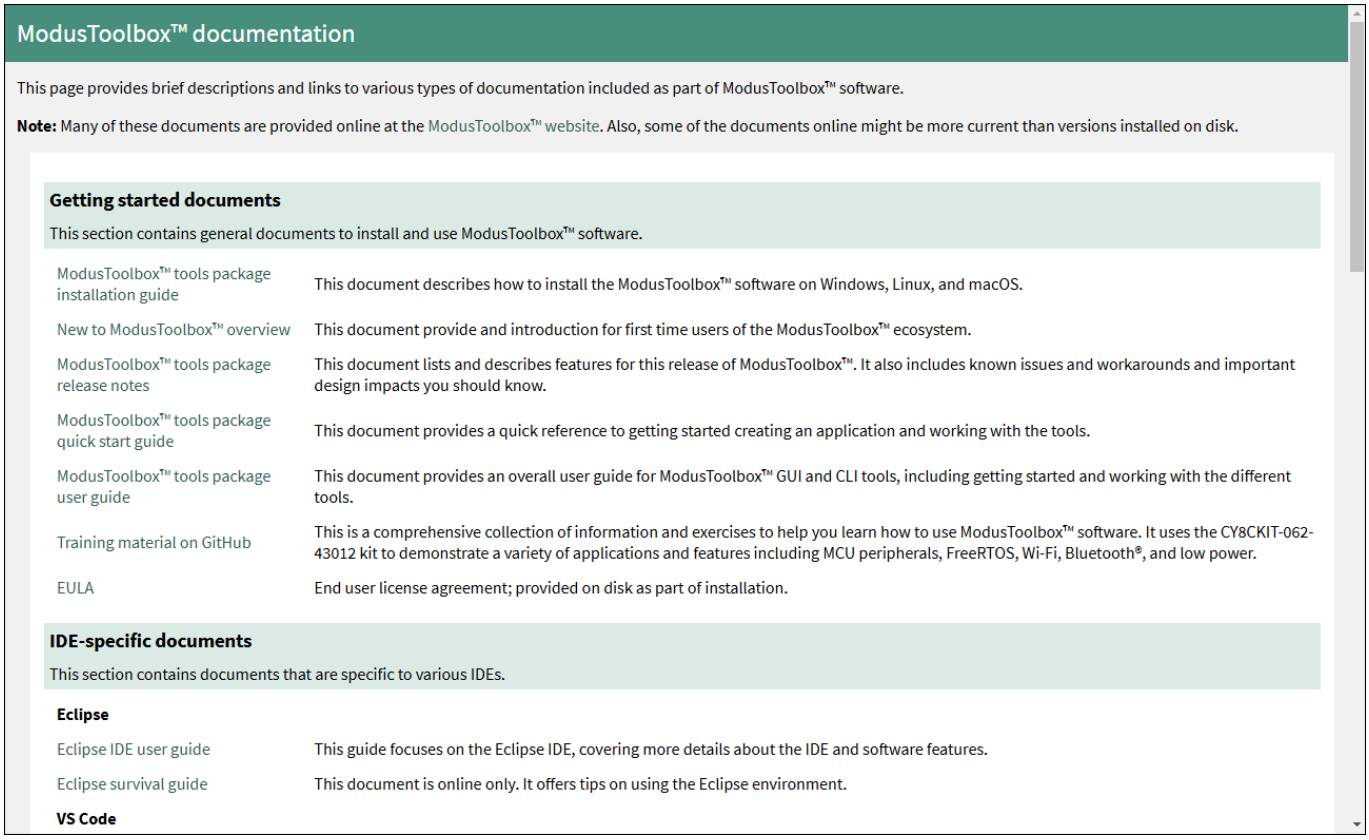
1.2.1 Base tools package directory structure

Refer to the ModusToolbox™ tools package installation guide for information about installing ModusToolbox™ software. Once it is installed, the various ModusToolbox™ top-level subdirectories are organized as follows in the "*ModusToolbox*" install directory :

***Note**: Your installation may include more than one ModusToolbox™ version. Refer to the Product versioning section for more details.*

1.2.1.1 docs_<version>

This is the top-level documentation directory. It contains various top-level documents and the *doc_landing.html* file with links to documents provided as part of ModusToolbox™ software. This file is also available from the Dashboard.

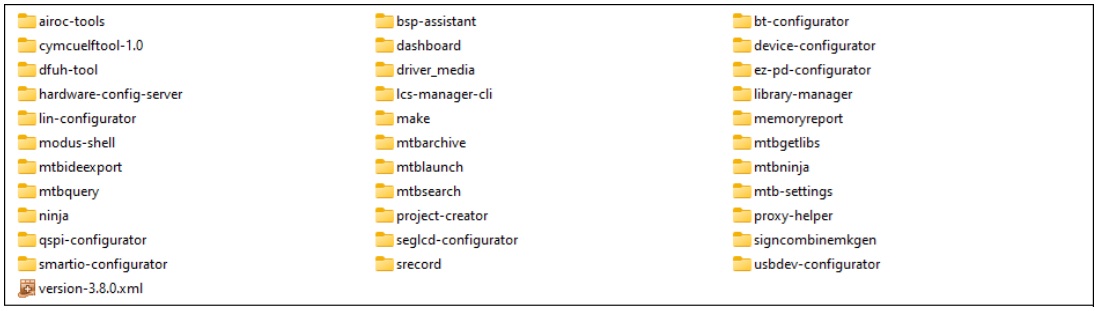


1.2.1.2 resources_<version>

This directory contains a *.json* file with details about the version of the tools package. There are also a couple helper files for the icon and launch-tool.

1.2.1.3 tools_<version>

This contains all the various tools and scripts installed as part of ModusToolbox™ tools package. See Tool descriptions for more information.



1.2.1.4 Packs

Packs can be installed separately from a tools package release. These packs include additional software and tools for specific features, such as machine learning. If you install a pack, it will create a "packs" subdirectory in the root "ModusToolbox" installation directory. Refer to the pack documentation for specific details about a pack.

To install a pack, use the Setup program or go to https://softwaretools.infineon.com/tools. There will be links to either install the pack directly or download it to install manually. The pack documentation will provide additional instructions and requirements, as needed.

1.2.2 Tool descriptions

The *tools_<version>* directory includes the following configurators, tools, and utilities:

1.2.2.1 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 and/or a 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:

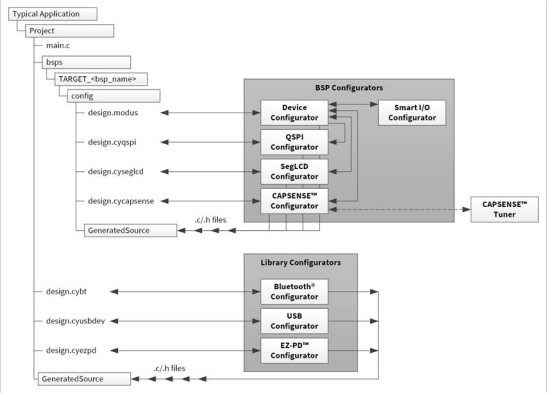
- Displaying a user interface for editing parameters
- Setting up connections such as pins and clocks for a peripheral
- Generating code to configure middleware

***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 that could be used in a typical application.

1 Introduction



BSP configurators

BSP configurators configure the hardware on a specific device. This can be a board provided by us, a 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. Refer to the Device Configurator user guide for more details.
- **CAPSENSE™ Configurator**: (installed separately) Configure CAPSENSE™ hardware, and generate the required firmware. This includes tasks such as mapping pins to sensors and how the sensors are scanned. Refer to the CAPSENSE™ Configurator user 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. Refer to the CAPSENSE™ Tuner user 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. Refer to the QSPI Configurator user 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. Refer to the Smart I/O Configurator user 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. Refer to the SegLCD Configurator user guide for more details.

Library configurators

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, etc.). Any source code generated by the configurator is stored in a GeneratedSource directory in the same directory as the XML file. The Library configurators include:

- **Bluetooth® Configurator**: Configure Bluetooth® settings. These include 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 AIROC™ Bluetooth® applications. Refer to the Bluetooth® Configurator user guide for more details.
- **USB Configurator**: Configure USB settings and generate the required firmware. This includes options for defining the Device Descriptor and Settings. Refer to the USB Configurator user guide for more details.

- **LIN Configurator** (installed separately): Configure various LIN settings, such as frames and signals, and generate the required firmware. Refer to the LIN Configurator user guide for more details.
- **EZ-PD™ Configurator**: Configure the features and parameters of the PDStack middleware for PMG1 family of devices. Refer to the EZ-PD™ Configurator user guide for more details.

1.2.2.2 Base tools

ModusToolbox™ software includes base tools that provide support for application creation, device firmware updates, and so on. All tools are installed by the ModusToolbox™ tools package 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
dashboard	Top-level starting point to create applications and BSPs.	user guide
project-creator	Create a new application.	user guide
library-manager	Add and remove libraries and BSPs used in an application; edits the *Makefile*.	user guide
bsp-assistant	Create and update BSPs.	user guide
mtb-settings	ModusToolbox™ Settings tool provides one place to configure various settings for all tools, such as proxy settings, content mode, manifest DB location, etc.	user guide
cymcuelftool	Older tool used to merge 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	Communicate with a device 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
mtbarchive	GUI and command-line tool to create a zip file archive of a project to share with colleagues.	The tool creates a readme file.

1.2.2.3 Utilities

ModusToolbox™ software includes some additional utilities that are often necessary for application development. In general, you do not use these utilities actively; they are triggered by various user actions.

Utility	Description
airoc-tools	Assorted utilities used for AIROC™ projects.
hardware-config-server	Background utility that brokers transactions between (potentially) multiple user-facing applications and a common design file.
make	Build system tools.
memoryreport	This tool generates memreport.txt and memreport.json files in the build directory that can be used with the Memory Configurator.
mtbideexport	Background utility used to provide various IDE support.
mtblaunch	Background utility used to launch various tools.
mtbninja/ninja	Background utility to speed up builds.

Utility	Description
mtbquery	Background utility used to query databases.
mtbsearch	Background utility used to search projects.
signcombinemkgen	Signer/combiner make generation tool.
SRecord	Collection of tools for manipulating EPROM load files. This is used to merge multi-core application images into a combined programmable HEX image.

1.2.2.4 Build system infrastructure

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

- create an executable
- provide debug capabilities

A *Makefile* defines everything required for your application, including:

- target hardware (board/BSP 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 `SEARCH` variable. You can also explicitly list files in the `SOURCES` and `INCLUDES` make variables.

See ModusToolbox™ build system for detailed documentation on the build system infrastructure.

1.2.2.5 Program and debug support

For programming and debugging, the ModusToolbox™ Programming tools package is available separately, and it is installed in addition to the base tools package. You can use various IDEs to program devices and establish a debug session (see Using supported IDEs).

The following tools are part of the Programming tools package; they are not installed in the base package "*ModusToolbox*" tools directory:

Tool	Description	Documentation
ModusToolbox™ Programmer	ModusToolbox™ Programmer functionality is built into ModusToolbox™ Software. ModusToolbox™ is also available as a stand-alone tool.	user guide
fw-loader	A simple command line tool to identify which version of KitProg is on a 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 a low-level communication/debug firmware. Use fw-loader to upgrade your kit to KitProg3, if needed.	user guide
OpenOCD	Our specific implementation of OpenOCD is installed with ModusToolbox™ software.	developer's guide

ModusToolbox™ software supports the Open On-Chip Debugger (OpenOCD) using a GDB server, and supports the J-Link debug probe. See Switch to J-Link for details.

ModusToolbox™ Programmer uses 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™ Programming tools package. The latest version of the tool is also available separately in a GitHub repository.

1.3 Run-time software

ModusToolbox™ software also includes an extensive collection of GitHub-hosted repos comprising Code Examples, BSPs, 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 example templates 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. Refer to https://github.com/Infineon/modustoolbox-software#libraries for more details.

1.3.1 Code examples

ModusToolbox™ code examples applicable to your selected BSP can be found using the Project Creator tool. For most code examples, you can use the Create application from template to create an application and use it directly with ModusToolbox™ tools. For some examples, you will need to follow the directions in the code example README file. Instructions vary based on the nature of the application and the targeted ecosystem.

1.3.2 Libraries (middleware)

In addition to the 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 installs all the libraries your application needs.

A ModusToolbox™ application can use different libraries based on the BSP. In general, there are several categories of libraries. Each library is delivered in its own folder, complete with documentation.

1.3.2.1 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 display support or BTSpy.
- MCU Middleware – These include MCU-specific libraries such as freeRTOS or Clib support.

1.3.2.2 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

1.3.2.3 BSP-specific base libraries:

BSP-specific libraries include the following:

- PSOC™ Edge (mtb-dsl-pse8xxgp)
- PSOC™ Control, (mtb-hal-cat1, mtb-pdl-cat1, recipe-make-cat1a, cat1b, etc.)
- PSOC™ 4 (mtb-pdl-cat2, recipe-make-cat2)

1.3.2.4 Device database (device-db)

The device-db consists of a set of devices, known as marketing part numbers (MPNs), distributed in a set of hierarchical directories. Each MPN contains a base view describing the basic characteristics of the device and additional views as defined by applications that use the device-db.

1.3.3 BSPs

The 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*)
- start-up 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.

For more details about BSPs, refer to the Board support packages chapter later in this document.

1.3.3.1 Supported devices

ModusToolbox™ software supports development on the following Arm® Cortex-M MCUs.

- PSOC™ Edge
- PSOC™ Control
- PSOC™ 4 (See AN79953: Getting Started with PSOC™ 4 for the supported PSOC™ 4 devices.)
- PSOC™ 6
- AIROC™ Wi-Fi and Bluetooth® chips
- TRAVEO™
- USB device controller
- PMG1 USB-C Power Delivery
- Wireless Charging devices
- XMC™ Industrial

1.3.3.2 BSP releases

We release BSPs independently of ModusToolbox™ software as a whole. This search link finds all currently available BSPs on our GitHub site.

The search results include links to each repository, named TARGET_kitnumber. For example, you will find links to repositories like TARGET_KIT_PSC3M5_CC2. 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.

1.4 Product versioning

ModusToolbox™ products include tools and firmware that can be used individually, or as a group, to develop connected applications for our devices. We understand 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.

1.4.1 General philosophy

ModusToolbox™ software 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™ tools package and the ModusToolbox™ Setup program.

All ModusToolbox™ products developed by us 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. From the 2.3 release and newer, 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. From the 2.3 release and newer using the MTB flow, these dependencies are created using "release-vX.Y.Z" style tags. The tool also creates and updates a file named *assetlocks.json* 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.

1.4.2 Tools package versioning

The ModusToolbox™ tools installation package is versioned as MAJOR.MINOR.PATCH. The file located at *<install_path>/ModusToolbox/tools_<version>/version-x.y.z.xml* also indicates the build number.

Every MAJOR.MINOR version of a ModusToolbox™ product is installed by default into *<install_path>/ModusToolbox*. So, if you have multiple versions of ModusToolbox™ software installed, they are all installed in parallel in the same "*ModusToolbox*" directory.

1.4.3 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.

```
INFO: Multiple tools versions were found in
"/cygdrive/c/Users/XYZ/ModusToolbox/tools_a.b
/cygdrive/c/Users/XYZ/ModusToolbox/tools_x.y
C:/Users/XYZ/ModusToolbox/tools_x.y C:/Users/XYZ/ModusToolbox/tools_x.y".
This build is currently using "C:/Users/XYZ/ModusToolbox/tools_x.y".
Check that this is the correct version that should be used in this build.
To stop seeing this message, 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.
```

1.4.4 Specifying alternate tools version

By default, the ModusToolbox™ software uses the most current version of the *tools_<version>* directory installed. That is, if you have ModusToolbox™ versions 3.8 and 3.7 installed, and if you launch the Eclipse IDE, the IDE will use the tools from the *tools_3.8* directory to launch configurators and build an application. This section describes how to specify the path to the desired version.

Environment 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/User Variable, for example:

```
CY_TOOLS_PATHS=[ModusToolbox-install-path]/tools_[version]
```

Note: *Use a Windows style path, (that is, not like /cygdrive/c/). Also, use forward slashes. For example: C:/Users/XYZ/ModusToolbox/tools_3.x/*

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

Specific project Makefile

If you want to use a specific "tools" path for your current 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_3.x
```

This setting in the *Makefile* takes precedence over the environment variable .

1.4.4.1 Environment 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/User Variable, for example:

```
CY_TOOLS_PATHS=[ModusToolbox-install-path]/tools_[version]
```

***Note**: Use a Windows style path, (that is, not like /cygdrive/c/). Also, use forward slashes. For example: C:/Users/XYZ/ModusToolbox/tools_3.x/*

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

1.4.4.2 Specific project Makefile

If you want to use a specific "tools" path for your current 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_3.x
```

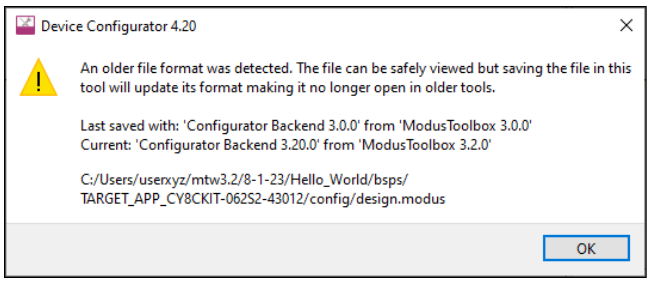
This setting in the *Makefile* takes precedence over the environment variable .

1.4.5 Tools and configurators versioning

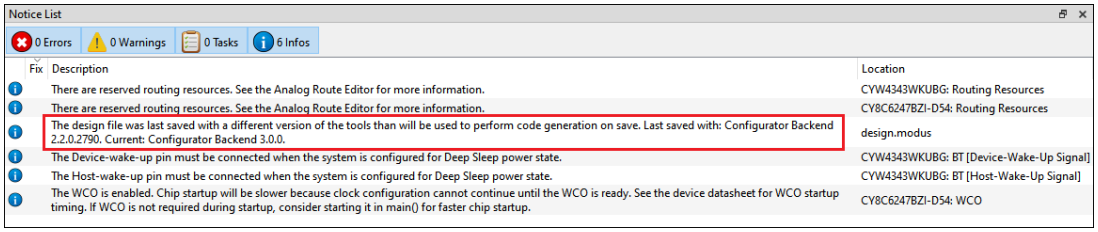
Every tool and configurator follow the standard versioning scheme and include a version.xml file that also contains a build number.

1.4.5.1 Configurator messages

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



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



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

1.4.6 GitHub libraries versioning

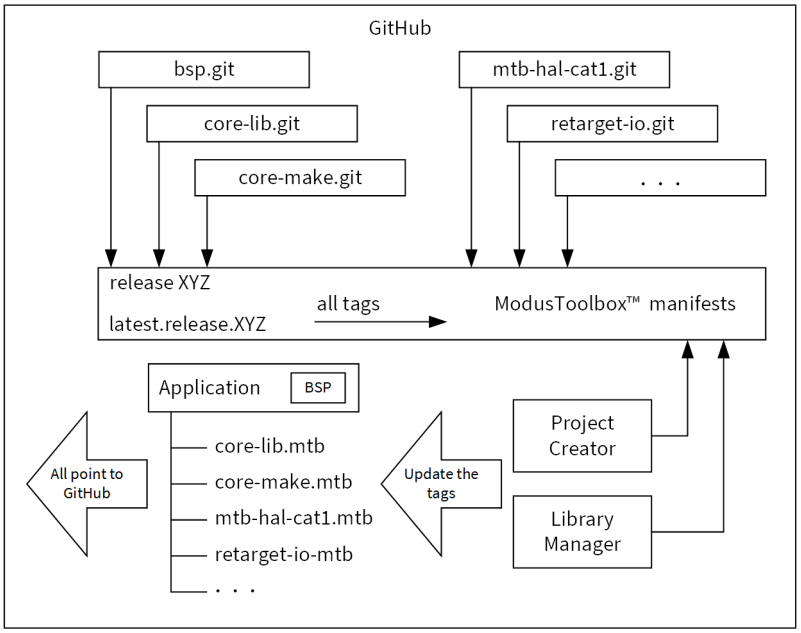
GitHub libraries follow the same versioning scheme: MAJOR.MINOR.PATCH. The GitHub libraries, besides the code itself, also provide two files in MD format: README and RELEASE. The latter includes the version and the change history.

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, if using the `git clone` method to create the application instead of the Project Creator tool, we recommend 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.

1.4.7 Dependencies between libraries

The following diagram shows the dependencies between libraries.



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

1.4.7.1 Git repo 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.

1.4.7.2 Regular C dependencies via #include

Our 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™ software, which has a fragmented distribution model. You are allowed and empowered to update libraries individually.

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 Visual Studio Code (VS Code) or the Eclipse-based IDE, if installed, which provides an integrated flow with all the ModusToolbox™ tools. Or you can use other IDEs, such as IAR Embedded Workbench, 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 getting started with ModusToolbox™ software includes these tasks:

- Install and configure software
- Launch Dashboard
- Create application from template
- Understand application structures
- Build and program

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™ Setup program is located on our website here: https://softwaretools.infineon.com/tools/com.ifx.tb.tool.modustoolboxsetup

The Setup program is used to install base packages and additional packages on Windows, Linux, and macOS. Select the latest tools package and any other packages you want, such as the Eclipse IDE for ModusToolbox™. The Setup program will also select dependency package, such as Programming tools. Refer to the ModusToolbox™ software installation guide for specific instructions.

2.1.1 GUI set-up instructions

In general, the GUI-based tools included as part of the ModusToolbox™ tools package work out of the box without any changes required. The same is true of packages installed through the Setup program in addition to the tools package. 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. To check your installation, open the appropriate command-line terminal for your operating system.

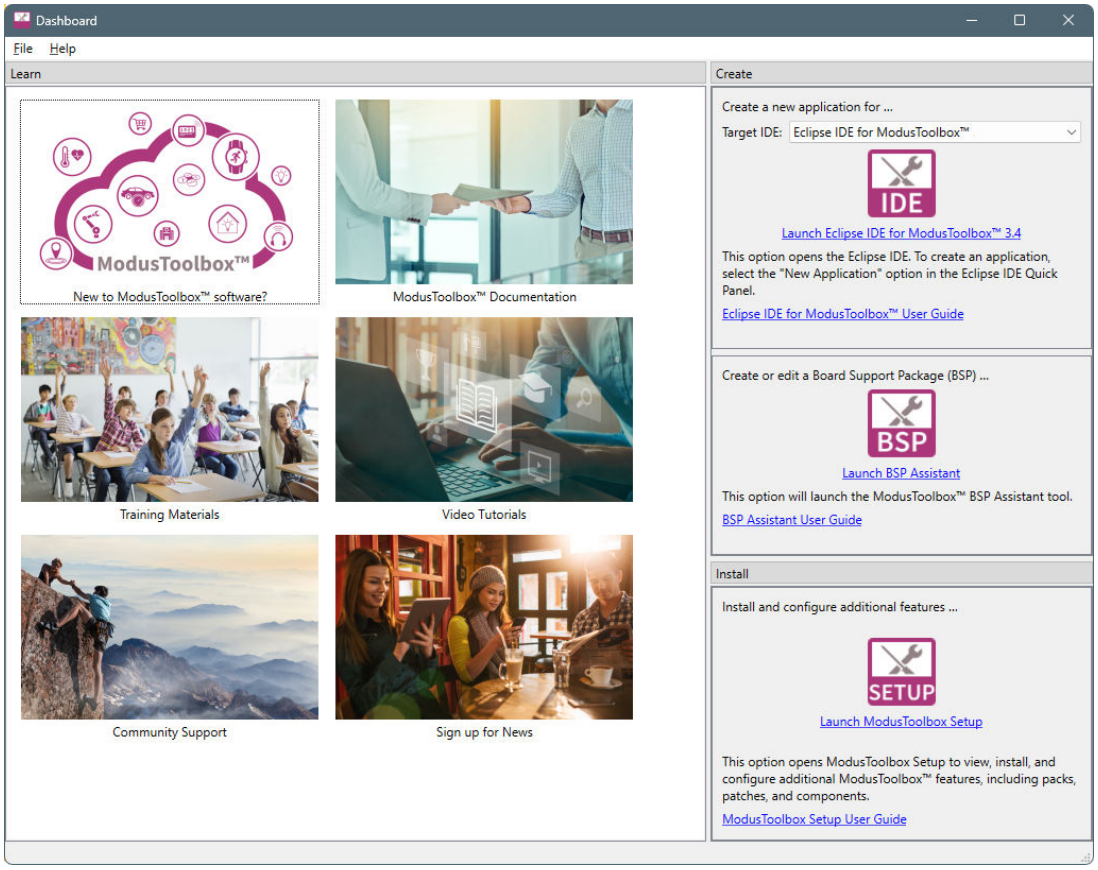
***Note**: For Windows, the tools package provides a command-line utility called "modus-shell." You can run this by typing "modus-shell" in the Windows search box.*

- 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 Launch Dashboard

The ModusToolbox™ tools package includes a tool called the Dashboard. You can launch this optional tool from the last step in the Setup program. You can also launch it manually as applicable for your operating system.



The Dashboard provides links to various sources of documentation and training materials. It also contains starting points: create a new application, create/edit a BSP, install or launch the ModusToolbox™ Setup program.

For more details about this tool, refer to the Dashboard user guide.

2.3 Create application from template

ModusToolbox™ software includes the Project Creator as both a GUI tool and a command line tool to easily create ModusToolbox™ applications. The Project Creator tool clones the selected BSP and code example template(s), and then creates the directory structure at the specified location with the specified name. The Project Creator tools also run the required processes to download and import all the necessary libraries and dependencies.

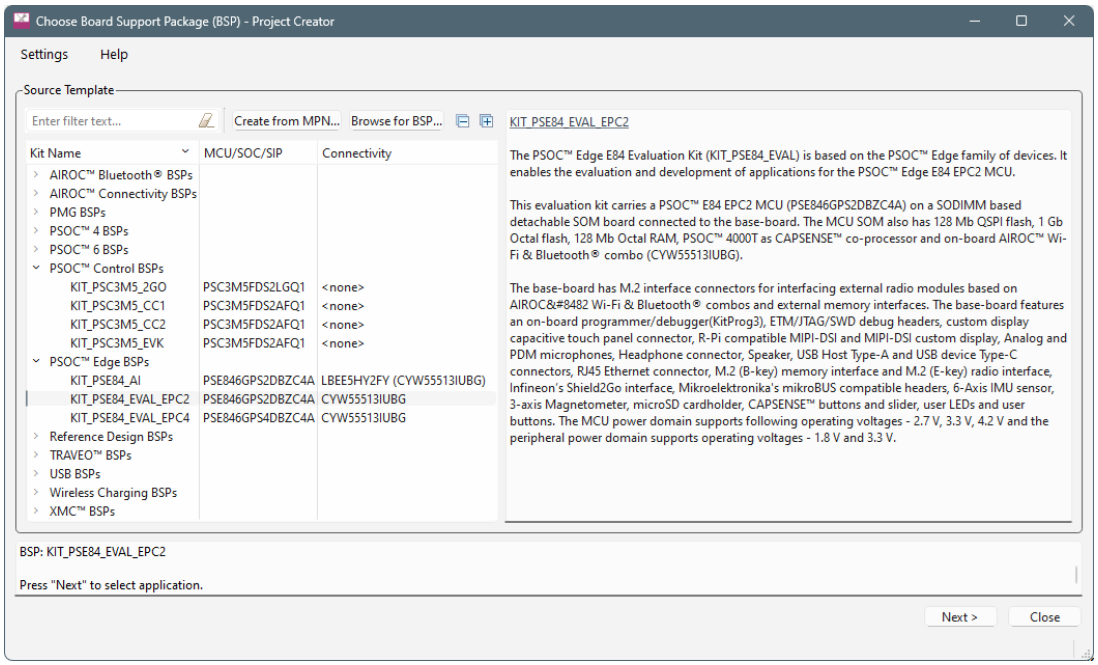
***Note**: This section describes creating a new application from a template. The process to import or share an existing application is covered in the Using applications with third-party tools chapter.*

2.3.1 Project Creator GUI

The Project Creator GUI tool provides a series of screens to select a BSP and code example template(s), specify the application name and location, as well as select the target IDE. The tool displays various messages during the application creation process.

Open the Project Creator GUI tool from the Dashboard or as applicable for your operating system. The executable file is installed in the following directory, by default:

<install_path>/ModusToolbox/tools_<version>/project-creator/



Refer to the Project Creator user guide for more details.

***Note**: The **Target IDE** option (on the Select Application page) is used to generate necessary files for the selected IDE. The Dashboard passes any selected **Target IDE** to the Project Creator tool. If you launch the Project Creator GUI tool from the Eclipse-based IDE, if installed, the tool seamlessly exports the created application for use in the Eclipse IDE.*

2.3.2 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_<version>/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.

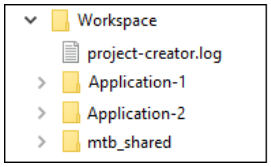
```
./project-creator-cli \
/project-creator-cli.exe \
--board-id KIT_PSC3M5_EVK \
--board-uri http://github-mirror.infineon.com/Infineon/TARGET_KIT_PSC3M5_EVK \
--board-commit latest-v2.X \
--app-id mtb-example-ce240510-hello-world \
--app-uri http://github-mirror.infineon.com/Infineon/mtb-example-ce240510-hello-world \
--app-commit latest-v1.X \
--target-dir C:/examples \
--user-app-name Hello_World \
--use-modus-shell
```

In this example, the project-creator-cli tool clones the Hello World code example template from our GitHub server (https://github.com/Infineon). It also updates the `TARGET` variable in the *Makefile* to match the selected BSP (`--board-id`), and obtains 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.

***Note**: You can run the `git clone` and `make getlibs` commands directly from a terminal; however, we recommend using the Project Creator tools (GUI or CLI) because some applications require additional processes to acquire all the submodules. If you choose to run the commands manually, make sure you thoroughly understand all the requirements of the selected application. Refer to the code example README.md file for details as needed.*

2.4 Understand application structures

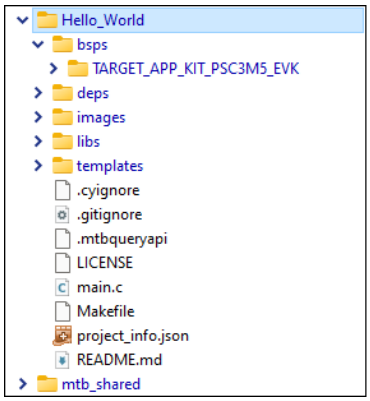
After creating one or more applications, they will be located in a top-level container, or workspace directory, that contains a project creation log file, one or more application directories, plus a *mtb_shared* directory. Depending on the example you chose to create the application, it can be either single-core or multi-core.



***Note**: Some older code examples default to 2.x application format. If you're using one of those, refer to Knowledge Base Article KBA236134.*

2.4.1 Single-core 3.x application

A typical single-core 3.x application, such as "Hello World," is one project directory with application source code, a Makefile, and assorted files, in addition to the bsps, deps, images, and libs subdirectories. A single-core application uses the ModusToolbox™ build system to produce a single ELF file for use on a single-core MCU.



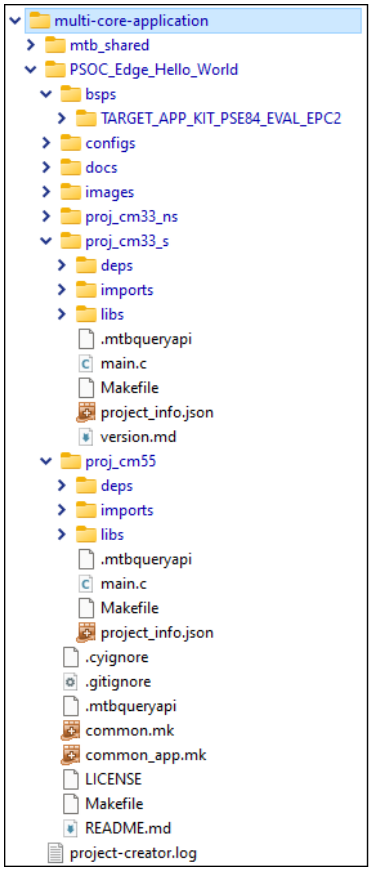
The following describe the contents for a single-core project directory:

- *bsps* subdirectory – This directory contains one or more BSPs for this specific project.
- *deps* subdirectory – By default, this subdirectory contains *<library>.mtb* files for libraries that were included directly or for which you changed using the Library Manager.
  - This subdirectory also contains the *assetlocks.json* file, which keeps track of the version for each dependent library.
- *images* subdirectory – If a project has images used by the *README.md* file, for example, this directory contains those images.
- *libs* subdirectory – This subdirectory may contain different types of files generated by the project creation process, based on how the project is created. You can regenerate these files using the Library Manager, so you do not need to add these files to source control.
  - If you update your project to specify any libraries to be local, then this directory will 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.
- *.gitignore* file – This file contains information about files for Git to ignore such as common, tool- or user-specific files that are typically not checked into a version control system.
- mtbqueryapi – A background file that provides project information.
- *LICENSE* file – This is the license agreement.
- Source code – This is one or more files for your project'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 project's *Makefile*, which contains configuration information such as `TARGET` for the BSP, `TOOLCHAIN`, and `MTB_TYPE` for the type of application; in this case, `COMBINED`.
- project_info.json – A json file containing the product ID.
- *README.md* file – This file describes the code example that was used to create the project.

***Note**: If an application needs to modify a standard BSP's configuration, then it will include a templates directory with various BSP templates, which contain configuration files (for example, design.modus) and a reserved resources list. If an application uses the BSP's configuration as-is, then it won't include a templates directory.*

2.4.2 Multi-core 3.x application

A multi-core 3.x application, such as "PSOC_Edge_Hello_World," includes multiple makefiles and various assorted files described under Single-core 3.x application. It also contains separate subdirectories for each of the core projects, plus the bsps subdirectory that applies to all the core projects in the application. A multi-core application directory hierarchy builds multiple ELF files for various purposes (for example, to support boot loading, multi-core support, or secure enclave scenarios).



2.4.2.1 Multi-core application directory

A multi-core application directory contains the following files and subdirectories:

- *Makefile* – The application *Makefile* contains the `MTB_TYPE` variable set to `APPLICATION`, plus the `MTB_PROJECTS` variable to specify the included projects. This file also includes the *common_app.mk* file and the path information to the *application.mk* file in the installation *tools_<version>* directory. This is responsible for forwarding build related requests to the individual core projects and dealing with post-build activities (for example, generating single monolithic HEX files that can be used to program all projects simultaneously) when they are complete.
- *common.mk* – This makefile is shared across all projects. It contains variables including: `MTB_TYPE`, `TARGET`, `TOOLCHAIN`, and `CONFIG`. In this case, `MTB_TYPE=PROJECT`. This file also includes a reference to the *common_app.mk* file.
- *common_app.mk* – This makefile is shared across the entire application and all its projects. It contains path information to indicate the location of the installation *tools_<version>* directory.
- *bsps* subdirectory – This contains one or more BSPs for all projects in the multi-core application.
- Multi-core project subdirectories – These contain the source code and *Makefile* for each specific core project. The name format is *proj_<core>*; for example, "*proj_cm7_0*" or "*proj_cm0p*".

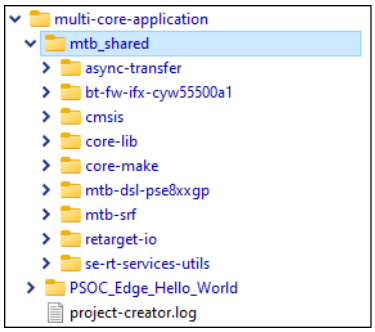
2.4.2.2 Multi-core project directories

Each multi-core project directory contains its own project Makefile that is responsible for compiling and linking a single ELF image. Multi-core project directories are similar to single-core project directories in that they contain source code, as well as libs and deps subdirectories. One main difference is that multi-core project directories do not have a bsps subdirectory, because they use the same BSP from the multi-core application directory.

- Source code – This is one or more files for the core project's code.
- *Makefile* – This is the core project's *Makefile*. It includes numerous variables used for the projects, such as `COMPONENTS`, `CORE`, `CORE_NAME`, and other variables used to specify flags and pre-build and post-build commands. This file also includes path information for source code discovery, shared repo location, and path to the compiler. Plus, it includes the *common.mk* file from the application and the path information to the *start.mk* file in the installation *tools_<version>* directory.
- *README.md* file – This file contains information for the specific core project.
- *deps* subdirectory – By default, this subdirectory contains *<library>.mtb* files for libraries that were included directly or for which you changed using the Library Manager.
  - This subdirectory also contains the *assetlocks.json* file, which keeps track of the version for each dependent library.
- *libs* subdirectory – This subdirectory may contain different types of files generated by the project creation process, based on how the project is created. You can regenerate these files using the Library Manager, so you do not need to add these files to source control.
  - If you update your project to specify any libraries to be local, then this directory will 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.

2.4.3 mtb_shared directory

Each workspace you create with one or more applications will also include a *mtb_shared* directory adjacent to the application directories, and this is where the shared 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. You can regenerate these files using the Library Manager, so you do not need to add these files to source control.



2.5 Build and program

After the application has been created, you can use the supported IDE of your choice for building and programming. 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, KIT_PSC3M5_EVK
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 [0 - 3]

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	Variable
GCC_ARM	GNU Arm® Embedded Compiler	macOS, Windows, Linux™	CY_COMPILER_GCC_ARM_DIR
ARM	Arm® compiler	Windows, Linux™	CY_COMPILER_ARM_DIR
IAR	Embedded Workbench	Windows	CY_COMPILER_IAR_DIR
LLVM_ARM	Open-source compiler	macOS, Windows, Linux™	CY_COMPILER_LLVM_ARM_DIR

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_IAR_DIR=[yourpath]. If this path is blank, the build infrastructure looks in the *ModusToolbox* install directory. These variables can also be set as environment variables for your system, so you don't have to change it for each application. See Path make variables for details about different variables' usage.

2.5.1 Use an IDE

The ModusToolbox™ ecosystem supports the following third-party IDEs, and we provide user guides for using those IDEs with a ModusToolbox™ application:

- Visual Studio Code for Modus Toolbox™ user guide
- Eclipse IDE for ModusToolbox™ user guide
- Keil µVision for ModusToolbox™ user guide
- IAR Embedded Workbench for Modus Toolbox™ user guide

2.5.2 Use command line

2.5.2.1 make build

When the Project Creator tool finishes creating the application and imports all the required dependencies, the application is ready to build. From the appropriate terminal, 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
```

In multi-core applications, it is possible to build only the current project instead of the entire application by using the following command:

```
make build_proj
```

2.5.2.2 make program

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

```
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.

3 Updating the example application

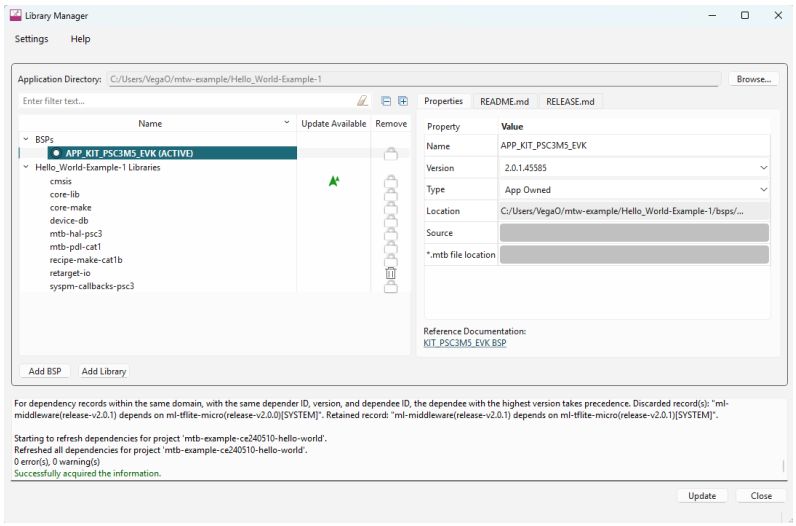
After completing the process to create and build an example application and program a board, you may wish to update the application in various ways to explore its capabilities. This chapter covers some of the basic tasks, including:

- Update libraries
- Create/edit BSPs
- Configure settings for devices, peripherals, and libraries
- Write application code
- Debug the application

3.1 Update libraries

Use the Library Manager tool to add or remove BSPs and libraries for your application, as well as change versions for libraries. You can also change the active BSP for your application.

Open the Library Manager GUI tool from the application directory using the `make library-manager` command. The Library Manager opens for the selected application and its available BSPs and libraries.

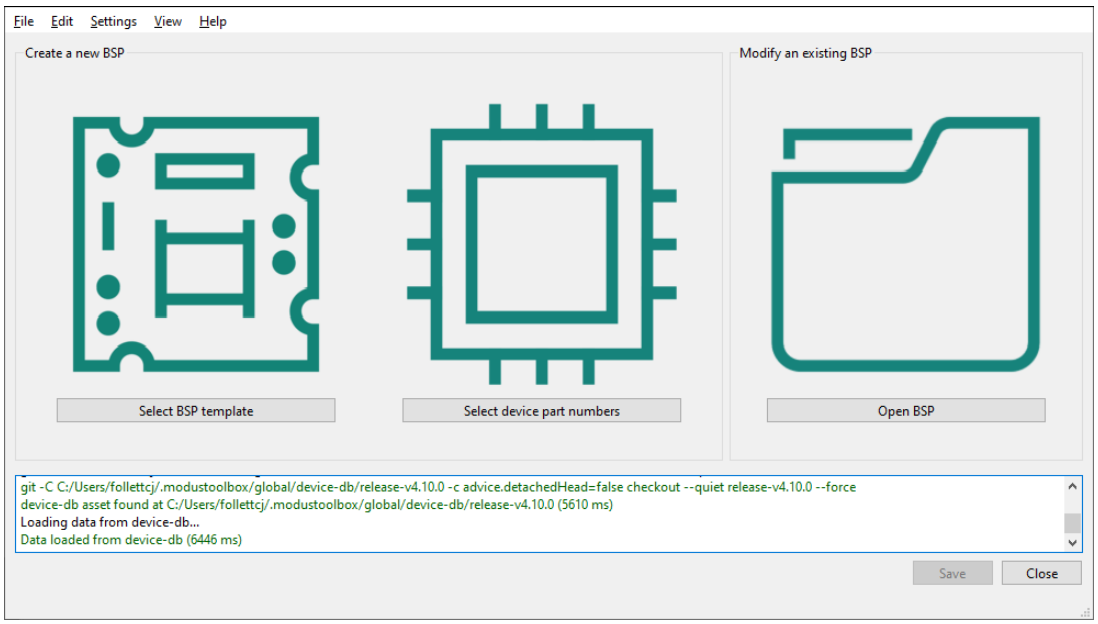


***Note**: There are several ways to open the Library Manager; refer to the Library Manager user guide for more details.*

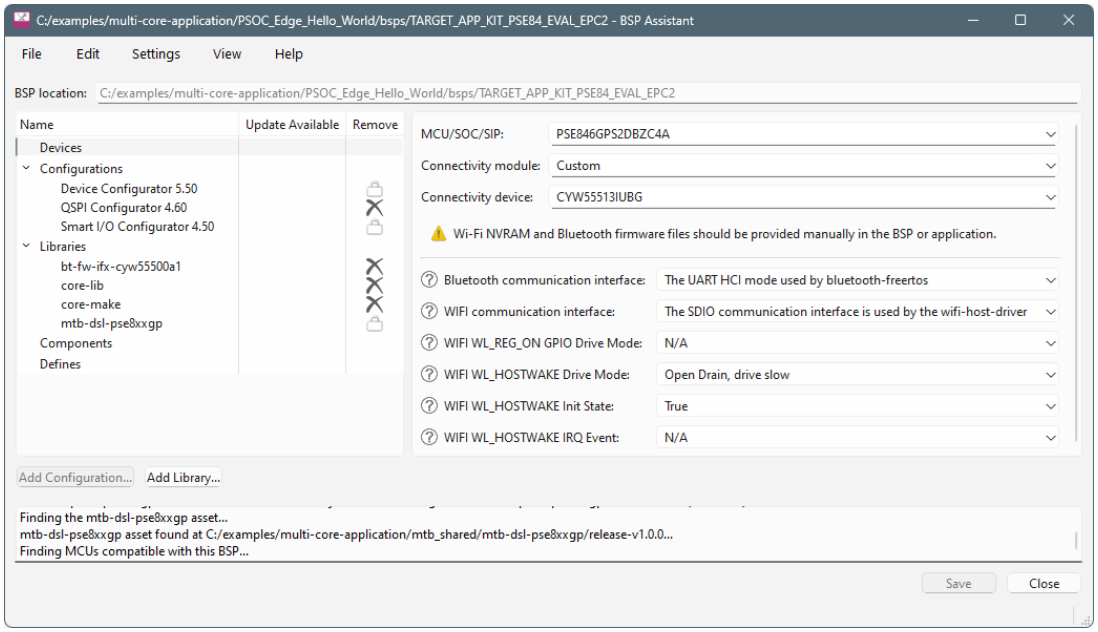
3.2 Create/edit BSPs

Use the BSP Assistant to change devices or add and remove configurations for the BSP in your application. The tool offers GUI and CLI versions. For more details, refer to the BSP Assistant user guide.

You can open the BSP Assistant GUI tool from Dashboard tool, or as applicable for your operating system. When opened this way, the BSP Assistant provides options to open an existing BSP, or create a new one.



If you have an existing application, open the tool from the application directory using the `make bsp-assistant` command. This opens the BSP Assistant for the selected BSP.



3.3 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. That decision affects how you configure settings for devices, peripherals, and libraries.

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. Also, each configurator provides a separate document, available from the configurator's **Help** menu, that provides information about how to use the specific configurator.

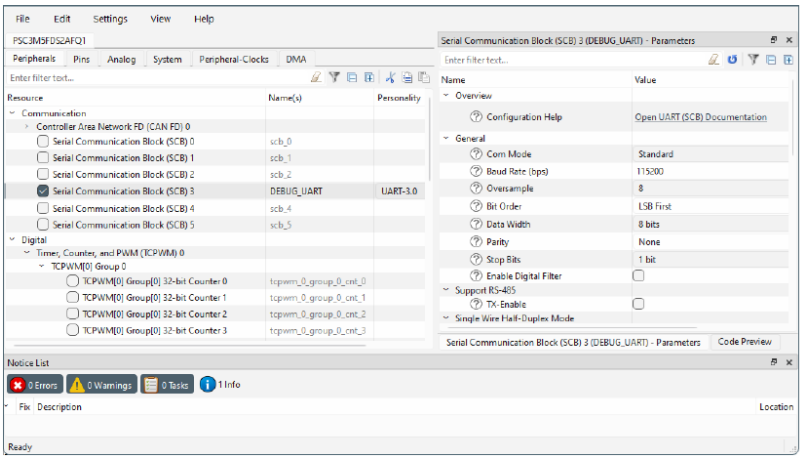
3.3.1 Configurator GUI tools

You can use VS Code or the Eclipse IDE, if installed, to open configurators. For example, if you select the "Device Configurator" link in the Eclipse IDE Quick Panel, the tool opens with the application's design.modus file.

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

```
make device-configurator
```

All these methods open the Device Configurator with the current application's design.modus configuration file.



As described under Tools targets, you can use the make command with appropriate arguments to open any configurator. For example, to open the CAPSENSE™ Configurator, run:

```
make capsense-configurator
```

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

3.3.2 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.

3.4 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.

ModusToolbox™ software is designed to enable your workflow. It includes support for VS Code, Eclipse IDE, IAR Embedded Workbench, and Keil µVision (see Using supported 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.

3.5 Debug the application

When you've added and changed code in your application, it is likely that something will not work as expected. At that point, you need to debug the application to determine what is wrong, or how to optimize the desired behavior. Similar to building an application and programming the board, you can use an IDE to debug the application. Refer to the following user guide as applicable for your IDE:

- Visual Studio Code for Modus Toolbox™ user guide
- Eclipse IDE for ModusToolbox™ user guide
- Keil µVision for ModusToolbox™ user guide
- IAR Embedded Workbench for Modus Toolbox™ user guide

3.5.1 Use Eclipse IDE

When using the Eclipse IDE, if installed, click the appropriate "**Program**" link in the Quick Panel for the selected application.

Refer to the "Program and Debug" chapter in the Eclipse IDE for ModusToolbox™ user guide for details about launch configs and various debugger settings.

3.5.2 Export to another IDE

If you prefer to use an IDE other than Eclipse, refer to your preferred IDE's documentation for debugging instructions. As noted under the Build and program section, you export a ModusToolbox™ application to a supported IDE following instructions in Using supported IDEs. There are also some debugging set-up instructions in that section.

3.5.3 Switch to J-Link

By default, most ModusToolbox™ applications are set to use the KitProg3 connection on the kit. **For Eclipse and VS Code**, if you want to use J-Link instead, you must set a couple of make variables as follows:

1. Open your ModusToolbox™ single-core application's *Makefile*, or multi-core application's *common.mk* file, and enter the following variable:

```
BSP_PROGRAM_INTERFACE=JLink
```

2. If you install J-Link software in a non-default location, or intend to use make commands, such as `make program` and `make qprogram`, also enter the following variable:

```
MTB_JLINK_DIR=<path to J-Link>
```

***Note**: If you want all projects/applications for the BSP to use the J-Link programmer/debugger, you can specify the `BSP_PROGRAM_INTERFACE` and `MTB_JLINK_DIR` variables in the bsp.mk file instead.*

3. Save the *Makefile*.

4. In a bash terminal run the applicable `make <ide>` variable for the IDE you wish to use.
   - For a VS code example, run:

```
make vscode
```

   - For an Eclipse example click **Generate Launches for ...**, or run:

```
make eclipse
```

   When the command completes, J-Link configurations will be shown where applicable.

For IAR Embedded Workbench and Keil µVision, you can make changes using their GUI tools.
For more specific details, refer to the specific user guide for the IDE you're using.

4 Version Control and sharing applications

If you are working on a design with more than one person, it is common to share an application using some type of version control system, by manually copying files, or exporting from a supported IDE. This section covers the files to include or exclude when sharing, as well as how to share using various methods.

- Files to include/exclude
- Basic sharing steps

4.1 Files to include/exclude

No matter which method you choose to share an application, you should know what is critical to copy or check in to version control, as well as what can be regenerated easily. The main files to consider when sharing an application include anything that you have changed or added, and that will not be regenerated. These files include source code, BSPs and configurations, *Makefile*, etc.

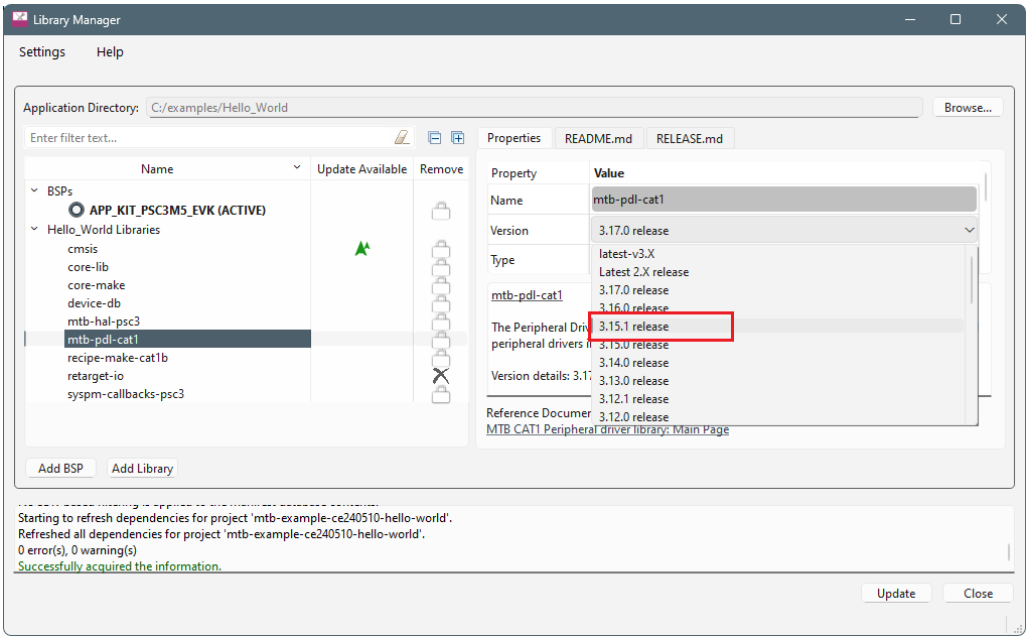
There are several directories in the application that can be recreated and therefore do not need to be copied or checked into version control. These include the *libs*, *mtb_shared*, *build*, and *GeneratedSource* directories. The processes that create them are:

- *libs* and *mtb_shared*: Created by running the Library Manager and clicking the **Update** button, or by running `make getlibs` on the command line. Either one will clone the libraries from GitHub to the appropriate locations.
- *build*: Created during the build process.
- *GeneratedSource*: Generated by running the associated configurator such as the Device Configurator or Bluetooth® Configurator. The build process run Configurators automatically if the *GeneratedSource* files are out of date.

4.2 Basic sharing steps

When sharing an application, follow these basic steps:

1. **Lock the library versions**. Some of the library references in your project might be pointing to the latest version. If a newer version of these becomes available, they could be automatically updated using the Library Manager or the make getlibs command. To prevent this, lock libraries to specific versions using the Version setting in the Library Manager. See the Library Manager user guide for more details.



2. **Lock the tools versions**. Communicate the version of the ModusToolbox™ tools package used to create the application. Different versions of tools may behave differently when sharing an application. The recipient should set the `CY_TOOLS_PATHS` variable to the same package version that you use. See Specifying alternate tools version for more details.
3. **Remove unneeded files**. See Files to include/exclude.
4. **Share the application**. Use version control software, manually copy files, and/or use an IDE as described in the following sections. Make sure to include all files and folders that you modified and/or added that will not be regenerated.

4.2.1 Using version control software

If you are working on a production design, you likely use version control software to manage the design and any potential revisions. This allows all users to stay synchronized with the latest version of an application. You can use any version control method or software that you prefer.

ModusToolbox™ code examples have a default *.gitignore* file that excludes directories that can be easily recreated, as well as files containing IDE-specific information that need not be checked in. If you are using Git as your version control software, you can often use that file as-is. However, you are free to change it to fit your needs. If you are using version control software other than Git, you can use the *.gitignore* file as a guide for configuring the software that you are using.

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

1. Get a copy of the checked-in data. This will vary depending on the version control software (for example using Git, `git clone <url>`).
2. Use the command line and run `make <IDE>` for the IDE you wish to use. For example, `make eclipse` or `make vscode`.
3. Work with the application as usual. The *build* and *GeneratedSource* files will be created automatically as needed.
4. When finished with your changes, check in your updates following your version control process.

4.2.2 Manual file copy

If you are not using version control software, you can just copy a complete application directory from one user to another. If desired, you can exclude the directories listed under Files to include/exclude since the libraries can be recreated, and the other files are regenerated when the application is built.

4.2.3 Saving/exporting from IDE

Another method to share files is by using your preferred IDE's export or **Save As** method. Refer to your IDE's documentation for details, keeping in mind certain files and folders need not be exported.

We provide the mtbarchive tool that you can use to create a ZIP archive.

5 ModusToolbox™ build system

This chapter covers various aspects of the ModusToolbox™ build system. Refer to CLI set-up instructions for getting started information about using the command line tools. This chapter is organized as follows:

- Overview
- make help
- make getlibs
- BSPs
- Environment variables
- Adding source files
- Pre-builds and post-builds
- Available make targets
- Available make variables
- Tools targets

5.1 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.

***Note**: User-defined command line make parameters are not supported and the only supported variables are TOOLCHAIN and TARGET.*

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.

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. 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 BSP library.

The majority of the *Makefiles* are two separate types: 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.

5.2 make help

The ModusToolbox™ build system includes a `make help` target that provides help documentation. 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.*

5.3 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 the source location of a library repo, a specific tag for a version of the code, and the location to store the library.

The `getlibs` target finds and processes all *.mtb* files and uses the `git clone` command to clone or pull the code as appropriate. The target generates *.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.

***Note**: ModusToolbox™ version 3.x no longer supports the old LIB flow, thus all .lib files are ignored.*

- 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 standard 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.

5.3.1 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 "our" 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 (that is, all changes must be committed). You may also use the *.gitignore* file for adding untracked files when running `make getlibs`. See also KBA231252.

5.4 BSPs

An application must specify a target BSP through the `TARGET` variable in the *Makefile*. We provide BSPs based on our kits to use as a starting point. When you create an application, the selected BSP is then owned by that application, and you can modify it as needed. 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 replace a BSP in an application.

5.5 Environment variables

ModusToolbox™ software supports custom installation paths, and you can use the Settings tool to set most of them. See the Settings tool user guide for more details. There is one variable that you have to set manually with an environment variable:

- `CY_TOOLS_PATHS` (path to the installation "*tools_<version>*" directory)

***Note**: When entering variables that require a path, use a Windows-style path (not Cygwin-style, like /cygdrive/c/). Also, use forward slashes. For example, "C:/MyPath/ModusToolbox/tools_<version>".*

5.6 Adding source files

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

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

5.6.1 Auto-discovery

The build system implements auto-discovery of 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.

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

5.6.1.1 .cyignore

Prior to applying auto-discovery and filtering, the build system will first search for *.cyignore* files in the roots of applications/projects, libraries, and BSPs, and construct a set of directories and files to exclude. This can be used to skip searching directories that contain no source code, or skip source code you don't want included. As an example, many ModusToolbox™ BSPs and libraries contain a directory named *docs* in the root directory, which contains no source code. So to speed up auto-discovery, these BSPs and middleware libraries include a *.cyignore* file that contains a line `docs`, which causes auto-discovery to ignore that directory. The system will only detect *.cyignore* files in the application/project root, as well as in the roots of libraries and BSPs, so there is not a recursive search.

All paths inside a *.cyignore* file are relative to the location of that specific *.cyignore* file. You could use `keep/all/except` if you wanted to keep "all" but exclude "except." You also can use "../" to look up and out from the current directory.

As mentioned, some library might contain source code that you do not want added to your build. Specifying that directory's location would require hard-coding the version of the library in use, which could potentially cause problems later. In this case, the variable `$(SEARCH_library)` solves this problem. For example, if the bluetooth-freertos library contains a directory named *firmware_deprecated*, you could add the following to the *.cyignore* file in the application root directory to exclude it:

```
$(SEARCH_bluetooth-freertos)/firmware_deprecated/
```

This causes that directory inside the bluetooth-freertos library to be ignored no matter where it is inside the workspace.

Another useful feature in a *.cyignore* file is a comment line. Any line that starts with "#" is skipped. So, you could write a description about why a file or directory is to be excluded. Or, you could just comment out an entry to assist in debugging your application.

***Note**: Wildcards and expressions are not allowed, and each directory or file requires its own entry in the file. However, the .cyignore file can contain make variables, such as SEARCH as shown above.*

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.*

5.6.1.2 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 variable. All others are ignored. ModusToolbox™ supports IAR, ARM, GCC_ARM, and LLVM_ARM.

5.6.1.3 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=APP_KIT_PSC3M5_EVK ignores all *TARGET_* directories except *TARGET=APP_KIT_PSC3M5_EVK*.

***Note**: The TARGET_ directory is often associated with the BSP, but it can be used in a generic sense. For example, 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.*

5.6.1.4 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.*

5.6.1.5 COMPONENT_<NAME>

Any directory that has the prefix "COMPONENT_" is interpreted as a directory that is component specific. This is used to enable/disable optional code. 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.

5.6.1.6 BSP makefile

Auto-discovery will also search for a *bsp.mk* file (aka, BSP makefile). If no matching BSP makefile is found, it will fail to build.

5.7 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:

```
project_prebuild:
    command1 -arg1
    Command2 -arg2
```

The sequence of execution in a build is as follows:

1. Recipe pre-build – Defined using `recipe_prebuild` target.
2. BSP pre-build – Defined using `bsp_prebuild` target.
3. Project pre-build – Defined using `project_prebuild` target.
4. Source compilation and linking.
5. Recipe post-build – Defined using `recipe_postbuild` target.
6. BSP post-build – Defined using `bsp_postbuild` target.
7. Project post-build – Defined using `project_postbuild` target.

5.8 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.

5.8.1 General make targets

Target	Description
`all`	Same as `build`. That is, builds the application. This target is equivalent to the `build` target.
`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. For multi-core applications, running this target builds all core projects in the application, and generates a combined *hex* file.
`build_proj`	Build a single project. Build a single target in the application. In single core-applications, this target is the same as the `build` target.

Target	Description
`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.
`qbuild_proj`	Builds a single project using the previous build's source list. In the single project-applications, this target is the same as the `qbuild` target. 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 application and programs it to the target device. In multi-core applications, this will program the combined hex file. The build process performs the same operations as the `build` target. Upon completion, the artifact is programmed to the board.
`program_proj`	Build and program only the current project to the target device. In single-core applications, this target is the same as the `program` target. The build process performs the same operations as the `build` target. Upon 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.
`qprogram_proj`	Programs a built project to the target device without rebuilding. In single-core applications, this target is the same as the `qprogram` target. This target allows programming an existing artifact to the board without a build step.
`erase`	Erases the MCU's internal flash memory and provides the option to erase any connected external nonvolatile memory. This operation is irreversible. This target performs a mass erase of the device's internal nonvolatile memory, removing firmware including any user data, configuration settings, and calibration data. By default, only internal memory is erased. To include supported external memory (for example, QSPI flash), use `make erase MTB_ERASE_EXT_MEM=1`. Erase is irreversible and may be blocked by device protection or security settings; unlock or clear protection first if required. Ensure there is stable power and do not interrupt the debug session during the operation.
`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.
`prebuild`	Generates code for the application. Runs custom prebuild commands to generate source code.

5.8.2 IDE make targets

Target	Description
`eclipse`	Generates Eclipse IDE launch configs and project files. This target generates a *.cproject* and a *.project* if they do not exist in the application root directory.

Target	Description
`vscode`	Generates VS Code IDE files. This target generates VS Code files for debug/program launches, IntelliSense, and custom tasks. These overwrite the existing files in the application directory except for *settings.json*.
`ewarm` / `ewarm8`	This target generates an IAR Embedded Workbench compatible *.ipcf* file that can be imported into IAR-EW. The *.ipcf* file is overwritten every time this target is run. Project generation requires Python 3 to be installed and present in the `PATH` variable. For applications that were created using core-make-3.0 or older, you must use the `make ewarm8` command instead.
`uvision` / `uvision5`	Generates a Keil µVision IDE .cprj file. This target generates a CMSIS-compatible file that can be imported into Keil µVision. The file is overwritten every time this target is run. Files in the default cmsis output directory will be automatically excluded when calling `make uvision`. Project generation requires Python 3 to be installed and present in the `PATH` variable. For applications that were created using core-make-3.0 or older, you must use the `make uvision5` command instead.

5.8.3 Utility make targets

Target	Description
`progtool`	Performs specified operations on the programmer/firmware-loader. Only available for devices that use KitProg3. This target expects user-interaction on the shell while running it. When prompted, you must specify the command(s) to run for the tool.
`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.

5.9 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.

5.9.1 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`
CORE	Specifies the name of the Arm® core for which a project is building (e.g. CM4). Example Usage: `make build CORE=CM4` Use this variable to select compiler and linker options to build a project for a specified Arm® core.
CORE_NAME	Specifies the name of the on-chip core for which a project is building (e.g. CM7_0). Example Usage: `make build CORE_NAME=CM7_0` Use this variable to select compiler and linker options to build a project for a specified on-chip core. This variable is applicable for some multi-core devices only (e.g. XMC7xxx).
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). This variable may also be used when generating launch configs when invoking the `eclipse` target.
TOOLCHAIN	Specifies the toolchain used to build the application. For example, IAR. Example Usage: `make build TOOLCHAIN=IAR` Supported toolchains for this include GCC_ARM, IAR, ARM and LLVM_ARM. When setting `TOOLCHAIN=IAR`, you should also specify the heap type using `LDFLAGS`. The `--advanced_heap` option is required if the program uses a library that requires it.
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.

5.9.2 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` 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` These MUST NOT have `-D` prepended.
VFP_SELECT	Selects hard/soft ABI or full software for floating-point operations [softfp hardfp softfloat]. If not defined, this value defaults to softfp. Example Usage (within *Makefile*): `VFP_SELECT=hardfp`
VFP_SELECT_PRECISION	Selects single-precision or double-precision operating mode for floating-point operations. If not defined, this value defaults to double-precision. Enable single-precision mode by using the "singlefp" option. Example Usage (within *Makefile*): `VFP_SELECT_PRECISION=singlefp`
CFLAGS	Prepends additional C compiler flags. Example Usage (within *Makefile*): `CFLAGS+= -Werror -Wall -O2`
CXXFLAGS	Prepends additional C++ compiler flags. Example Usage (within *Makefile*): `CXXFLAGS+= -finline-functions`
ASFLAGS	Prepends additional assembler flags. Usage is similar to `CFLAGS`.
LDFLAGS	Prepends additional linker flags. Example Usage (within *Makefile*): `LDFLAGS+= -nodefaultlibs`
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. Additional linker scripts can be added for GCC via the `LDFLAGS` variable as a `-L` option.

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 include the corresponding make variable to have that feature library be included in the build. For example, create a directory named *COMPONENT_ACCELEROMETER* into auto-discovery. Then, add the following make variable to the *Makefile*: COMPONENT=ACCELEROMETER. If the make variable does not include the <VALUE>, then that library will not be included in the build.
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.
SEARCH	List of paths to include in auto-discovery. For example, *../my_source/lib1*. Example Usage (within *Makefile*): SEARCH+=directory_containing_source_files The SEARCH variable can also be used by the application to include other directories to auto-discovery.
SKIP_CODE_GEN	Disables code generation from configurators when building. When set to a non-empty value, the build process will no longer run code generation from configurators.
MERGE	List of projects in the application to generate a combined *hex* file from. By default, building a multi-project application will generate a combined hex file from its sub-projects. This variable can be set from the application *Makefile* to override the set of projects to generate combined hex file from.
VCORE_ATTRS	Virtual core attribute. Currently supported values are "TRUSTZONE_SECURE" and "TRUSTZONE_NON_SECURE" which specify the core property for Arm® TrustZone.
TRUSTZONE_VENEER	The path of the veneer object file used for Arm® TrustZone. When specified for a secure project, it specifies the path of the veneer object file to generated. When specified for a non-secure project, it specifies the path of the veneer object file to include.

5.9.3 BSP make variables

Variable	Description
DEVICE	Device ID for the primary MCU on the target board/kit. Set by *bsp.mk*. The device identifier is mandatory for all board/kits.
ADDITIONAL_DEVICES	IDs for additional devices on the target board/kit. Set by *bsp.mk*. These include devices such as radios on the board/kit. This variable is optional.

Variable	Description
BSP_PROGRAM_INTERFACE	Specifies the debugging and programming interface to use. The default value and valid values all depend on the BSP. Possible values include KitProg3, JLink, and FTDI. Most BSPs will only support a subset of this list.

5.9.4 Getlibs make variables

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

***Note**: When entering variables that require a path, use a Windows-style path (not Cygwin-style, like /cygdrive/c/). Also, use forward slashes. For example, "C:/MyPath/ModusToolbox/tools_<version>."*

Variable	Description
MTB_USE_LOCAL_CONTENT	If set to non-empty, enable local content storage. Enable local content storage to allow use of ModusToolbox™ software without requiring internet access. Refer to the LCS Manager CLI User guide for more details.
CY_GETLIBS_PATH	Path to the intended location of libs info directory. The directory contains local libraries and metadata files about shared libraries.
CY_GETLIBS_DEPS_PATH	Path to where the library-manager stores *.mtb* files. Setting this path allows relocating the directory that the library-manager uses to store the .mtb files in your application. The default location is in a directory named deps.
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 is set from the project *Makefile*.
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. By default, this is set from the project *Makefile*.

5.9.5 Path make variables

These variables are used to specify various paths.

***Note**: When entering variables that require a path, use a Windows-style path (not Cygwin-style, like /cygdrive/c/). Also, use forward slashes. For example, "C:/MyPath/ModusToolbox/tools_<version>."*

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 the Makefile.
`CY_COMPILER_GCC_ARM_DIR`	Absolute path to the gcc-arm toolchain directory. Setting this path overrides the default GCC_ARM toolchain directory. It is used when the compiler is located at a non-default directory. Make uses this variable for the path to the assembler, compiler, linker, objcopy, and other toolchain binaries. For example, `CY_COMPILER_GCC_ARM_DIR=C:/Program Files (x86)GNU arm Embedded Toolchain/10 2021.10` When set in the *Makefile*, no quotes are required.
`CY_COMPILER_IAR_DIR`	Absolute path to the IAR toolchain directory. Setting this path overrides the default IAR toolchain directory. It is used when the compiler is located at a non-default directory. Make uses this variable for the path to the assembler, compiler, linker, objcopy, and other toolchain binaries. For example, `CY_COMPILER_IAR_DIR=C:/Program Files/IAR Systems/Embedded Workbench 9.1/arm` When set in the *Makefile*, no quotes are required.
`CY_COMPILER_ARM_DIR`	Absolute path to the ARM toolchain directory. Setting this path overrides the default ARM toolchain directory. It is used when the compiler is located at a non-default directory. Make uses this variable for the path to the assembler, compiler, linker, objcopy, and other toolchain binaries. For example, `CY_COMPILER_ARM_DIR=C:/Program Files/ARMCompiler6.16` When set in the *Makefile*, no quotes are required.
`CY_COMPILER_LLVM_ARM_DIR`	Absolute path to the LLVM toolchain directory. Setting this path overrides the default LLVM toolchain directory. It is used when the compiler is located at a non-default directory. Make uses this variable for the path to the assembler, compiler, linker, objcopy, and other toolchain binaries. For example, `CY_COMPILER_LLVM_ARM_DIR=C:/Program Files/LLVM` When set in the *Makefile*, no quotes are required.

Variable	Description
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.
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 however CY_PYTHON_PATH is defined, it will use that python executable.
MTB_JLINK_DIR	Specifies the path to the SEGGER J-Link software install directory "JLink". Example Usage: MTB_JLINK_DIR:=C:/Program Files/SEGGER/JLink Setting this path allows the make system to locate the JLink executable when calling make program. If not specified, make will default to the JLink executable in the PATH variable. When generating launch configurations for IDEs, this will override the default J-Link path.

5.9.6 Miscellaneous make variables

These are miscellaneous variables used for various make targets.

***Note**: When entering variables that require a path, use a Windows-style path (not Cygwin-style, like /cygdrive/c/). Also, use forward slashes. For example, "C:/MyPath/ModusToolbox/tools_<version>."*

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_IDE_PRJNAME	This variable contains the name that will be used to create the application name and support files for an IDE during application export. Use this variable to provide a custom name of an application for the IDE during application export. If CY_IDE_PRJNAME is not set on the command line, then APPNAME (which is specified in the *Makefile*) is used as the default. This name can be important for some IDEs to set a correct meaningful name for the application, support files, and launch configurations in the IDE.

Variable	Description
CY_SIMULATOR_GEN_AUTO	If set to 1, automatically generate a simulator archive (if supported by the target device). When enabled, the build make target will generate a debugging *tgz* archive for the Infineon online simulator as part of the post-build process.

5.10 Tools targets

***Note**: There are various targets to launch tools and configurators that are not part of the make system, but they can be used in the application directory. The following table lists a few of the common targets as a convenience. Refer to the applicable user guide for details for the given configurator or tool.*

Target	Description
bsp-assistant	Launches the BSP Assistant with the active BSP for the application.
bt-configurator	Launches the Bluetooth® Configurator GUI for the application's *cybt* file.
capsense-configurator	Launches the CAPSENSE™ Configurator GUI for the target's *cycapsense* file.
capsense-tuner	Launches the CAPSENSE™ Tuner GUI for the target's *cycapsense* file.
device-configurator	Launches the Device Configurator on the application's *.*modus* file.
library-manager	Launches the Library Manager for the application to add/remove libraries and to upgrade/downgrade existing libraries.
lin-configurator	Launches the LIN Configurator GUI for the target's *mtblin* file.
qspi-configurator	Launches the QSPI Configurator GUI for the target's *cyqspi* file.
seglcd-configurator	Launches the Segment LCD Configurator GUI for the target's *cyseglcd* file.
smartio-configurator	Launches the Smart I/O Configurator GUI for the target's *modus* file.
usbdev-configurator	Launches the USB Configurator GUI for the target's *cyusbdev* file.

6 Board support packages

This chapter describes a board support package (BSP). The major headings include:

- Overview
- What's in a BSP
- Creating your own BSP

6.1 Overview

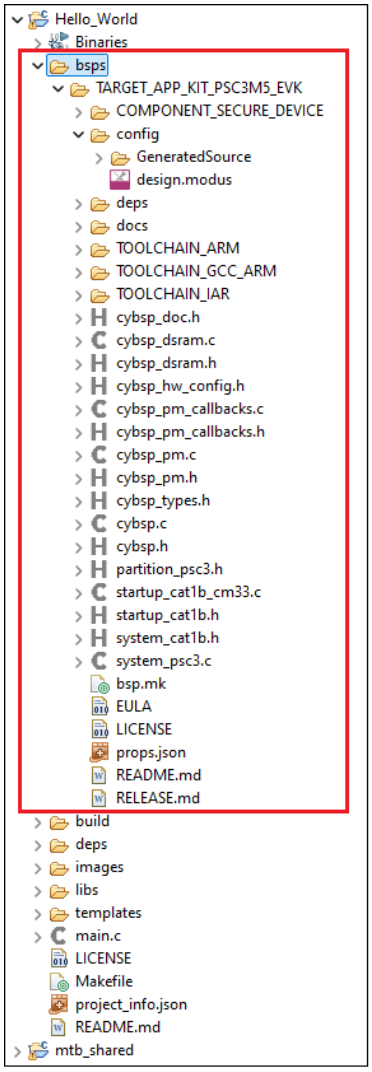
A BSP provides a standard interface to a board's features and capabilities. The API is consistent across our kits. Other software (such as middleware or an application) can use the BSP to configure and control the hardware. BSPs do the following:

- 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

6.2 What's in a BSP

This section presents an overview of the key resources that are part of a BSP. BSPs are owned by an application.

The following shows a typical PSOC™ Control BSP located in the *bsps* subdirectory.



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

6.2.1 TARGET

This is the top-level directory for a BSP. All BSPs begin with "TARGET" and this is referenced in the application *Makefile* for the active BSP.

6.2.2 config

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.

6.2.3 COMPONENT

Some applications may have "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.

6.2.4 deps subdirectory

The *deps* subdirectory inside the BSP contains *.mtbx* files for various library dependencies for the BSP.

6.2.5 docs subdirectory

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

6.2.6 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()`.

6.2.7 bsp.mk

This file defines the `DEVICE` and other BSP-specific make variables such as `COMPONENTS`. These are described in the ModusToolbox™ build system chapter. This file 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.

6.2.8 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.

6.2.9 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.

6.3 Creating your own BSP

For ModusToolbox™ version 3.x, there is a tool called the BSP Assistant to create and modify BSPs. Refer to the BSP Assistant user guide for details about using that tool.

6 Board support packages

For a better understanding of the contents and structure of a BSP and more detailed information about how to create a custom BSP, as well as update the Wi-Fi and Bluetooth® connectivity device and firmware in a BSP, refer to Application Note AN235297.

7 Manifest files

This chapter describes a manifest files. The major headings include:

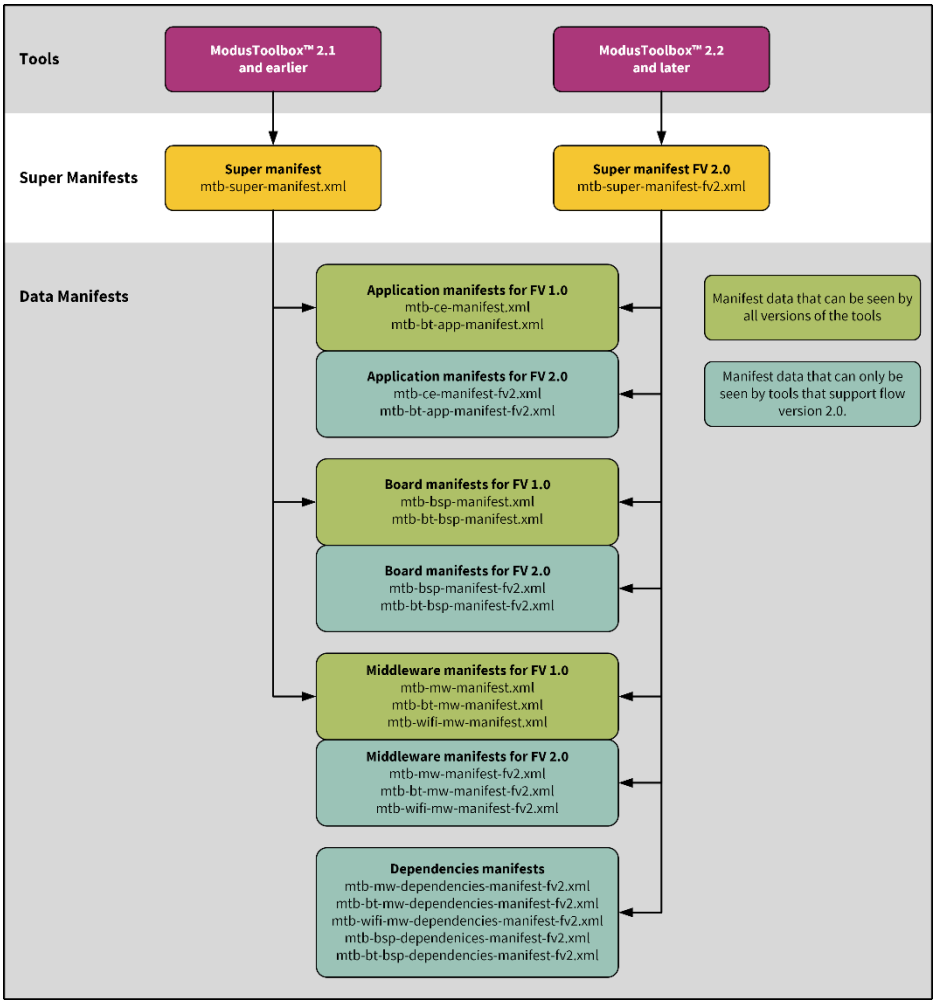
- Overview
- Create your own manifest
- Local content storage

7.1 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.

There are two versions of manifest files: ones for earlier versions of ModusToolbox™ software (2.1 and earlier), and one for newer versions of ModusToolbox™ (2.2 and later, aka "fv2"). The older super-manifest file for use with earlier versions contains only references manifests that contain items that support the older ModusToolbox™ flow. The newer super-manifest file for use with the ModusToolbox™ 2.2 release and later contains references to all the manifest files.



7.2 Create your own manifest

By default, the ModusToolbox™ tools look for our manifest files maintained on our server. So, the initial lists of BSPs, code examples, and middleware available to use are limited to our 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 files provided on GitHub:

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

Make sure you look at the "fv2" manifest files if you are using the flow for ModusToolbox™ version 2.2 and later.

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

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

- You can create supplementary super-manifest files that identify additional content. The tools will merge your additional content with the default super-manifest.
- You can replace the default super-manifest file used by the tools.

7.2.1 Supplementing super-manifest using manifest.loc

In addition to the standard super-manifest file, you can specify "custom" super-manifest files. This allows you to add additional items (BSPs, code examples, libraries) along with the standard items. You can do this by creating a *manifest.loc* file in a hidden subdirectory 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
```

If this file exists, then each line in the file is treated as the URL to another super-manifest file, which is exactly like the standard super-manifest file. The data from these manifests is combined with data from the standard super-manifest. See the Conflicting data section for dealing with conflicts.

7.2.2 Replacing standard super-manifest using variable

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. This removes all Infineon content from the tools, by default. For example:

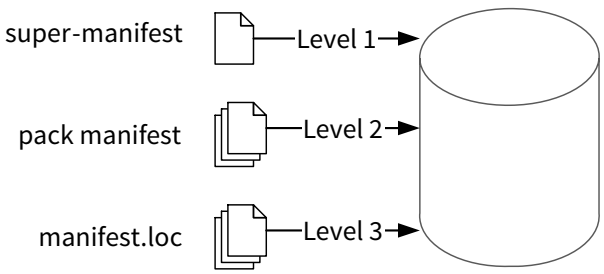
```
CyRemoteManifestOverride=https://myURL.com/mylocation/super-manifest.xml
```

7.2.3 Processing

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

- The first level is to access the super-manifest file(s) to obtain a list of manifest files.
- The second level is to retrieve the manifest data from any packs that were installed.
- The third level is to retrieve the manifest data from manifest.loc file, if it exists.

All the manifest data is merged 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.



7.2.4 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 level 1 primary super-manifest (and linked manifests), data that comes from the level 2 pack manifest, if present, and data that comes from the level 3 *manifest.loc* file, if present. In this case, the data in the level 2 pack manifest overrides the data from the level 1 standard super-manifest, and the data in the level 3 *manifest.loc* file overrides the data in the level 2 pack manifest. 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.

7.3 Local content storage

The Local content storage (LCS) feature provides a command-line tool called lcs-manager-cli that allows you to create your own local, offline content on demand.

For more details, refer to the LCS Manager CLI user guide.

8 Using applications with third-party tools

ModusToolbox™ software includes a variety of ways to use applications with third-party tools. This chapter covers the following:

- Using supported IDEs
- Multi-core debugging
- Generating files for XMC™ Simulator tool

8.1 Using supported IDEs

8.1.1 Overview

As described in the Getting started chapter, the Project Creator tool includes a **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 different supported IDEs. We have created user guides for each of them.

8.1.2 Eclipse

If you install Eclipse from the Setup program, it 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™ user guide.

8.1.3 VS Code

For VS Code, you can create an application using the Project Creator tool, or export an existing application using `make vscode`. Then, open the workspace file in VS Code. For more details, refer to the Visual Studio Code for ModusToolbox™ user guide.

8.1.4 IAR EWARM (Windows only)

For IAR Embedded Workbench, you can create an application using the Project Creator tool, or export an existing application using `make ewarm TOOLCHAIN=IAR`.

***Note**: For applications that were created using core-make-3.0 or older, you must use the `make ewarm8` command instead.*

Then, follow procedures in the IAR Embedded Workbench for ModusToolbox™ user guide to open and configure the application.

8.1.5 Export to Keil µVision (Windows only)

For Keil µVision, you can create an application using the Project Creator tool, or export an existing application using `make uvision TOOLCHAIN=ARM`.

***Note**: For applications that were created using core-make-3.0 or older, you must use the `make uvision5` command instead.*

Then, follow procedures in the Keil µVision for ModusToolbox™ user guide to open and configure the application.

8.2 Multi-core debugging

Infineon provides different multi-core MCUs in its portfolio. Sometimes you need to debug complex problems usually connected with IPC. Multi-core debugging allows you to simultaneously debug two or more cores available on the target MCU. This section applies to PSOC™ 6 MCUs, as well as the XMC7000 device family.

Multi-core debugging is supported for the following IDEs: Eclipse IDE for ModusToolbox™, VS Code, IAR EWARM, and Keil µVision. Refer to the applicable user guide for the IDE you plan to use.

8.2.1 Timing

When launching a multi-core debug group, do not start debugging (resume, step, etc.) in the first launched sessions until all the remaining launch configurations in a group have been initiated and started successfully.

8.2.2 CM0+ core rule

In PSOC™ 6, TRAVEO™ 2, and XMC7000 devices, system calls are always performed by the primary CM0+ core, even if it is initiated (via NMI) by the secondary core (CM4 or CM7). Because of this, you have to follow this rule for a smooth debugging experience of a multi-core application:

Attention**: **The CM0+ core must NOT be halted (suspended at the breakpoint) when another core (CM4 or CM7) is requesting system calls. You must resume the CM0+ core and let it run some code in your application [for example, `Cy_SysLib_Delay()`], or just perform several single-step operations, while the CM4 code is invoking the system call.

If you deviate from this rule, you may experience different issues depending on the usage scenarios, including the application state, IDE, and debug

- The debugger can be confused by the unexpected value of the Program Counter for the CM0+ core when, instead of performing the single-step operation, it jumps to the SROM area executing the system call requested by the CM4 or CM7 core.
- The CM4 or CM7 core may be stuck in an endless loop in the code that just initiated the system call and waiting for its completion, while the CM0+ core is suspended at the breakpoint.

8.3 Generating files for XMC™ Simulator tool

For the XMC1100, XMC1200, XMC1300, and XMC1400 families of devices, you can generate an archive file to upload to the XMC™ Simulator tool (https://design.infineon.com/tinaui/designer.php) for simulation and debugging. To do this:

Specify the `CY_SIMULATOR_GEN_AUTO=1` variable as follows:

- Edit the application *Makefile* to add the `CY_SIMULATOR_GEN_AUTO=1` variable, and then build the application, or
- Add the variable on the command line: `make build CY_SIMULATOR_GEN_AUTO=1`

When the build completes, it generates an archive file (*<application-name>.tar.tgz*) in the *<Application-Name>\build\<Kit-Name>\Debug* directory, and the build message displays the URL to the appropriate simulator tool. For example:

```
================================================================================
= Generating simulator archive file =
================================================================================
Simulator archive file C:/Users/XYZ/mtw/5699/xmc-2/Empty_XMC_App/build/KIT_XMC12_BOOT_001/Debug/
mtb-example-xmc-empty-app.tar.tgz successfully generated
```

- If using the Eclipse IDE, click the link in the Quick Panel under **Tools** to open the XMC™ Simulator tool in the default web browser.
- If using the command line, run `make online_simulator`.

Upload the generated archive file to the XMC™ Simulator tool, and follow the tool's instructions for using the tool as appropriate.

Trademarks

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

Edition 2026-08-15

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2026 Infineon Technologies AG

All Rights Reserved.

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-ipt1778788852283

Important notice

Products which may also include samples and may be comprised of hardware or software or both ("Product(s)") are sold or provided and delivered by Infineon Technologies AG and its affiliates ("Infineon") subject to the terms and conditions of the frame supply contract or other written agreement(s) executed by a customer and Infineon or, in the absence of the foregoing, the applicable Sales Conditions of Infineon. General terms and conditions of a customer or deviations from applicable Sales Conditions of Infineon shall only be binding for Infineon if and to the extent Infineon has given its express written consent.

For the avoidance of doubt, Infineon disclaims all warranties of non-infringement of third-party rights and implied warranties such as warranties of fitness for a specific use/purpose or merchantability. Infineon shall not be responsible for any information with respect to samples, the application or customer's specific use of any Product or for any examples or typical values given in this document.

If this document is marked as 'Preliminary', 'Target', 'Draft' or 'Proposal', Infineon reserves the right to change all information given in this document at any time without notice. Subject to the development and release of a Product for series supply by Infineon, the technical specifications of the Product are set forth in the relevant datasheet provided by Infineon without such marking.

The data contained in this document is exclusively intended for technically qualified and skilled customer representatives. It is the responsibility of the customer to evaluate the suitability of the Product for the intended application and the customer's specific use and to verify all relevant technical data contained in this document in the intended application and the customer's specific use. The customer is responsible for properly designing, programming, and testing the functionality and safety of the intended application, as well as complying with any legal requirements related to its use.

Unless otherwise explicitly approved by Infineon, Products may not be used in any application where a failure of the Products or any consequences of the use thereof can reasonably be expected to result in personal injury.

However, the foregoing shall not prevent the customer from using any Product in such fields of use that Infineon has explicitly designed and sold it for, provided that the overall responsibility for the application lies with the customer. Infineon expressly reserves the right to use its content for commercial text and data mining (TDM) according to applicable laws, e.g. Section 44b of the German Copyright Act (UrhG). If the Product includes security features:

Because no computing device can be absolutely secure, and despite security measures implemented in the Product, Infineon does not guarantee that the Product will be free from intrusion, data theft or loss, or other breaches ("Security Breaches"), and Infineon shall have no liability arising out of any Security Breaches.

If this document includes or references software:

The software is owned by Infineon under the intellectual property laws and treaties of the United States, Germany, and other countries worldwide. All rights reserved. Therefore, you may use the software only as provided in the software license agreement accompanying the software.

If no software license agreement applies, Infineon hereby grants you a personal, non-exclusive, non-transferable license (without the right to sublicense) under its intellectual property rights in the software (a) for software provided in source code form, to modify and reproduce the software solely for use with Infineon hardware products, only internally within your organization, and (b) to distribute the software in binary code form externally to end users, solely for use on Infineon hardware products. Any other use, reproduction, modification, translation, or compilation of the software is prohibited.

For further information on the Product, technology, delivery terms and conditions, and prices, please contact your nearest Infineon office or visit www.infineon.com.