

ModusToolbox™ tools package user guide

ModusToolbox™ tools package version 3.1.0

About this document

Scope and purpose

This guide provides information and instructions for using the ModusToolbox™ tools provided by the version 3.1.0 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 describes the ModusToolbox™ build system.
- Chapter 5 covers different aspects of the ModusToolbox™ board support packages (BSPs).
- Chapter 6 explains the ModusToolbox™ manifest files and how to use them with BSPs, libraries, and code examples.
- Chapter 7 provides instructions for using a ModusToolbox™ application with various third-party tools.

Document conventions

Convention	Explanation
Bold	Emphasizes heading levels, column headings, menus and sub-menus
<i>Italics</i>	Denotes file names and paths.
Courier New	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

The following define the abbreviations and terms used in this document that you may not be familiar with:

- BSP – board support package
- PDL – peripheral driver library
- HAL – hardware abstraction layer
- WHD – Wi-Fi host driver
- WCM – Wi-Fi connection manager

Table of contents

Table of contents

1	Introduction	4
1.1	What is ModusToolbox™ software?	4
1.2	Run-time software	4
1.3	Tools package	7
1.4	Product versioning	13
1.5	Partner ecosystems	17
2	Getting started	18
2.1	Install and configure software	18
2.2	Launch Dashboard	19
2.3	Create application from template	19
2.4	Understand application structures	21
2.5	Build and program	25
3	Updating the example application	27
3.1	Update libraries	27
3.2	Create/edit BSPs	28
3.3	Configure settings for devices, peripherals, and libraries	29
3.4	Write application code	30
3.5	Debug the application	31
4	ModusToolbox™ build system	33
4.1	Overview	33
4.2	make help	34
4.3	make getlibs	34
4.4	BSPs	35
4.5	Environment variables	35
4.6	Adding source files	35
4.7	Pre-builds and post-builds	37
4.8	Available make targets	38
4.9	Available make variables	40
5	Board support packages	46
5.1	Overview	46
5.2	What's in a BSP	46
5.3	Creating your own BSP	48
6	Manifest files	49
6.1	Overview	49
6.2	Create your own manifest	50
6.3	Local content storage	51
7	Using applications with third-party tools	52
7.1	Version Control and sharing applications	52
7.2	Using supported IDEs	53
7.3	Multi-core debugging	54



Table of contents

7.4	Generating files for XMC™ Simulator tool.....	55
-----	---	----

Introduction

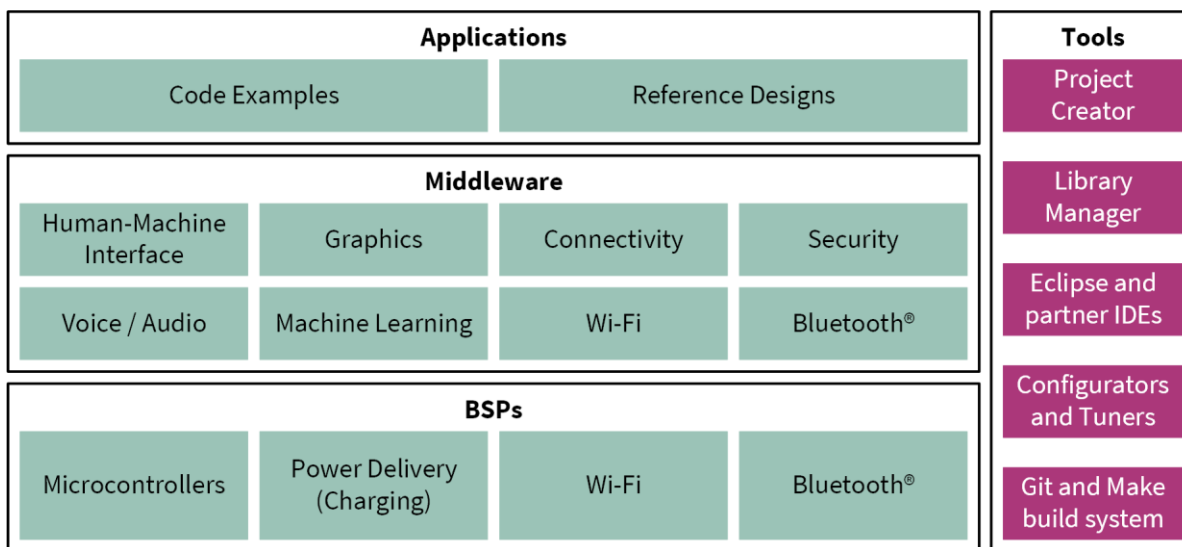
1 Introduction

This chapter provides an overview of the ModusToolbox™ software environment, which provides support for many types of devices and ecosystems.

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 Run-time software

ModusToolbox™ tools also include 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.

Introduction

The extensive middleware collection includes an ever-growing set of sensor interfaces, display support, and connectivity-focused libraries. The ModusToolbox™ installer also conveniently bundles packages of all the necessary run-time components you need to leverage the key Infineon technology focus areas. Refer to <https://github.com/Infineon/modustoolbox-software#libraries> for more details.

1.2.1 Code examples

All current ModusToolbox™ examples can be found through the GitHub [code example page](#). There you will find links to examples for the Bluetooth® SDK, PSoC™ 6 MCU, PSoC™ 4 device, among others. For most code examples, you can use the [Project Creator tool](#) 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 repository to instantiate the example. Instructions vary based on the nature of the application and the targeted ecosystem.

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

You can control the versions of the libraries being downloaded and also their location on disk, and whether they are shared or local to the application. Refer to the [Library Manager user guide](#) for more details.

1.2.2 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 downloads all the libraries your application needs. See [ModusToolbox™ build system](#) chapter to understand how all this works.

All current ModusToolbox™ libraries can be found through the GitHub [ModusToolbox™ software page](#). A ModusToolbox™ application can use different libraries based on the Active BSP. In general, there are several categories of libraries. Each library is delivered in its own repository, complete with documentation.

1.2.2.1 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.

1.2.2.2 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

Introduction

1.2.2.3 BSP-specific base libraries:

BSP-specific libraries include `mtb-hal`, `mtb-pdl`, and `recipe-make`. Some of these are identified as device-specific using the following categories:

- `cat1/cat1a` = PSoC™ 6 MCUs (`mtb-hal-cat1`, `recipe-make-cat1a`, etc.)
- `cat2` = PSoC™ 4 devices and XMC™ Industrial MCUs (`mtb-hal-cat2`, `mtb-pdl-cat2`)
- `cat3` = XMC™ Industrial MCUs (`recipe-make-cat3`)
- `cat4` = AIROC™ CYW43907 and CYW54907 (`mtb-hal-cat4`)

1.2.2.4 PSoC™ 6 additional libraries:

Due to the nature of the PSoC™ 6 MCU, plus the combo devices, certain PSoC™ 6 BSPs have additional libraries, including:

- Bluetooth® Middleware Libraries – These are for the BTStack and Bluetooth® FreeRTOS.
- PSoC™ 6 Middleware – These are libraries specific to the PSoC™ 6 MCU, such as EMEEPROM and DFU.
- Wi-Fi Middleware Libraries – These are libraries for connectivity applications on a PSoC™ 6 MCU with AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chip.

1.2.3 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*)
- startup code and linker files for the device
- other libraries that are required to support a kit

BSPs are aligned with our development/evaluation kits; they provide files for basic device functionality. A BSP typically has a *design.modus* file that configures clocks and other board-specific capabilities. That file is used by the ModusToolbox™ configurators. A BSP also includes the required device support code for the device on the board. You can modify the configuration to suit your application.

1.2.3.1 Supported devices

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

- AIROC™ Wi-Fi and Bluetooth® chips
- PMG1 USB-C Power Delivery Microcontroller
- PSoC™ 4 Configurable Microcontroller (See [AN79953: Getting Started with PSoC™ 4](#) for the supported PSoC™ 4 devices.)
- PSoC™ 6 MCU
- PSoC™ 64 "Secure Boot" MCU
- XMC™ Industrial Microcontroller

Introduction

1.2.3.2 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_CY8CPROTO-062-4343W](#). Each repository provides links to relevant documentation. The following links use this BSP as an example. Each BSP has its own documentation. The information provided varies, but typically includes one or more of:

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

A BSP is specific to a board and the device on that board. For custom development, you can create or modify a BSP for your device.

1.3 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.

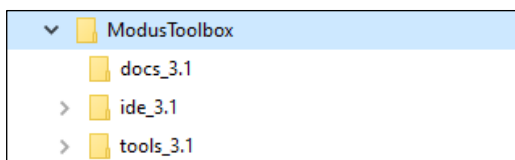
The [ModusToolbox™ tools package installer](#) provides required and optional core resources for any application. This section provides an overview of the available resources:

- [Directory structure](#)
- [Tools](#)

The installer does not include [code examples](#) or [libraries](#), but it does provide the tools to access them.

1.3.1 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 directories are organized as follows:



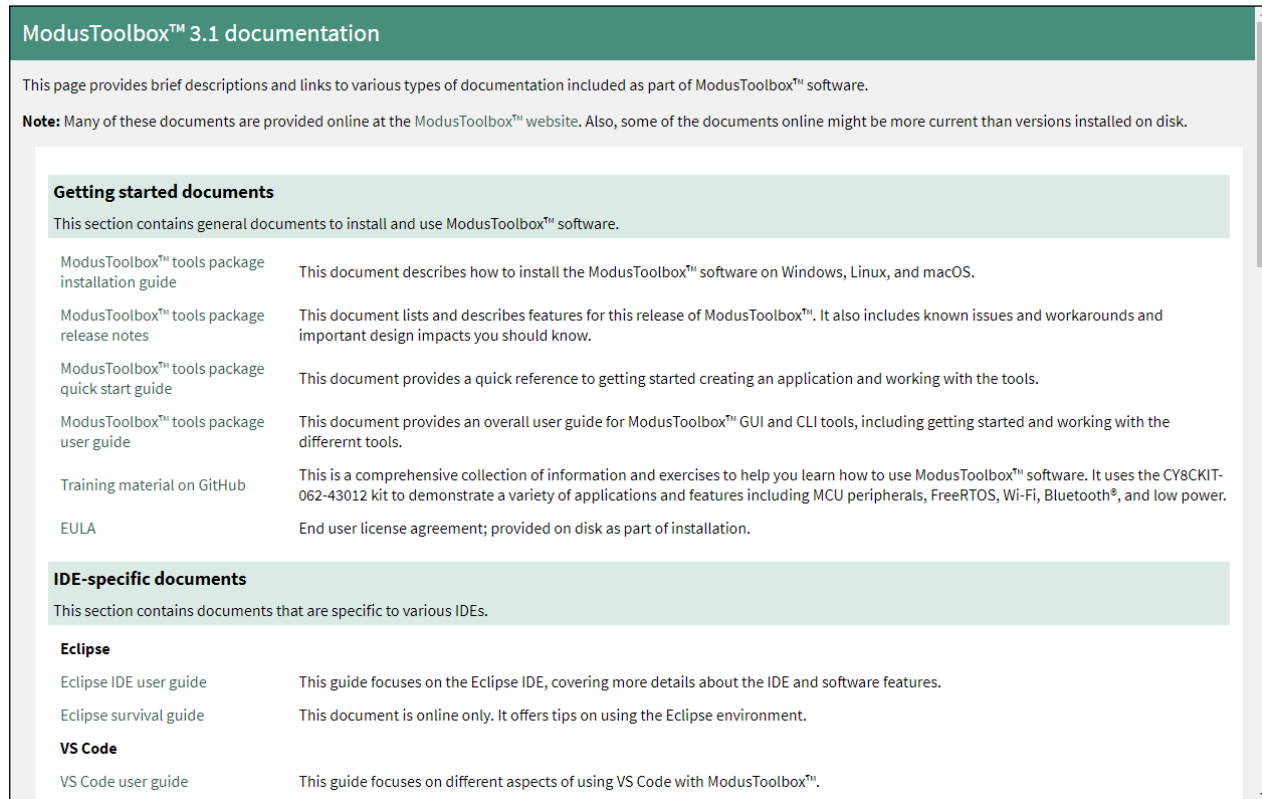
Note: This image shows ModusToolbox™ version 3.1 installed. Your installation may include more than one ModusToolbox™ version. Refer to the [Product versioning](#) section for more details.

Introduction

The *ModusToolbox* directory contains the following subdirectories for version 3.1:

1.3.1.1 docs_3.1

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.



The screenshot shows the 'ModusToolbox™ 3.1 documentation' landing page. It features a green header and a main content area with a light green background. The page includes a brief introduction, a note about online vs. local documents, and two main sections: 'Getting started documents' and 'IDE-specific documents'. Each section contains a list of documents with their titles and brief descriptions.

Getting started documents	
This section contains general documents to install and use ModusToolbox™ software.	
ModusToolbox™ tools package installation guide	This document describes how to install the ModusToolbox™ software on Windows, Linux, and macOS.
ModusToolbox™ tools package release notes	This document lists and describes features for this release of ModusToolbox™. It also includes known issues and workarounds and important design impacts you should know.
ModusToolbox™ tools package quick start guide	This document provides a quick reference to getting started creating an application and working with the tools.
ModusToolbox™ tools package user guide	This document provides an overall user guide for ModusToolbox™ GUI and CLI tools, including getting started and working with the different tools.
Training material on GitHub	This is a comprehensive collection of information and exercises to help you learn how to use ModusToolbox™ software. It uses the CY8CKIT-062-43012 kit to demonstrate a variety of applications and features including MCU peripherals, FreeRTOS, Wi-Fi, Bluetooth®, and low power.
EULA	End user license agreement; provided on disk as part of installation.

IDE-specific documents	
This section contains documents that are specific to various IDEs.	
Eclipse	
Eclipse IDE user guide	This guide focuses on the Eclipse IDE, covering more details about the IDE and software features.
Eclipse survival guide	This document is online only. It offers tips on using the Eclipse environment.
VS Code	
VS Code user guide	This guide focuses on different aspects of using VS Code with ModusToolbox™.

1.3.1.2 ide_3.1

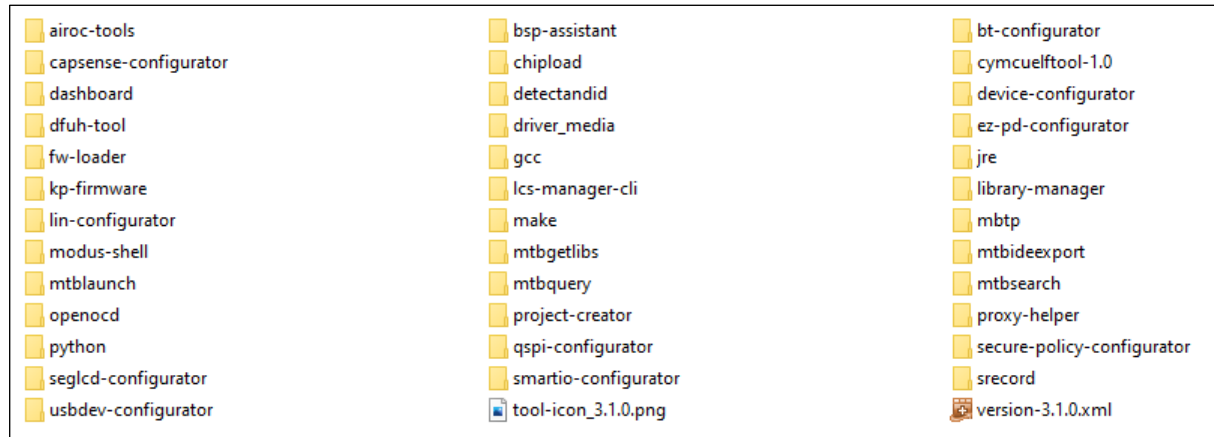
For convenience, the tools package installation includes the Eclipse IDE for ModusToolbox™. However, we fully support the following IDEs and their corresponding compiler technology, so you are free to develop the way you wish:

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

Introduction

1.3.1.3 tools_3.1

This contains all the various tools and scripts installed as part of ModusToolbox™ tools package. See [Tools](#) for more information.



1.3.1.4 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, 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.

1.3.2 Tools

The `tools_3.1` directory includes the following configurators, tools, and utilities:

1.3.2.1 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:

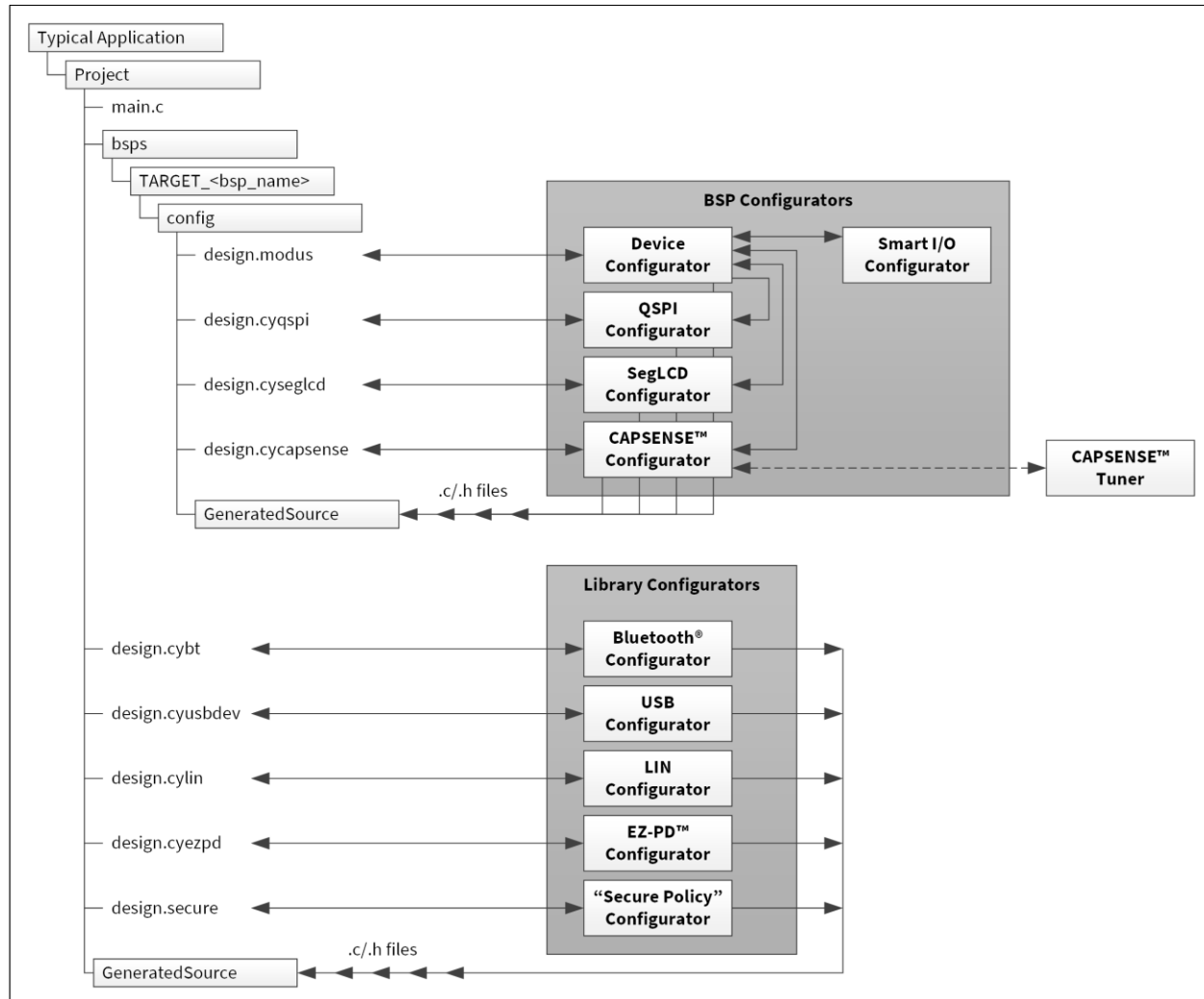
- Displaying a user interface for editing parameters
- Setting up connections such as pins and clocks for a peripheral
- Generating code to configure middleware

Note: Some configurators may not be useful for your application.

Introduction

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.



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:** 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.

Introduction

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:** 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.
- **Secure Policy Configurator:** Open, create, and change policy configuration files for PSoC™ 64 "Secure Boot" MCU devices. Refer to the [Secure Policy Configurator user guide](#) for more details.

1.3.2.2 Other tools

ModusToolbox™ software includes other tools that provide support for application creation, device firmware updates, and so on. All tools are installed by the [ModusToolbox™ 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 <i>Makefile</i> .	user guide
bsp-assistant	Create and update BSPs.	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 <i>docs</i> directory

Introduction

Other tools	Details	Documentation
dfuh-tool	Communicate with a PSoC™ 6 MCU that has already been programmed with an application that includes device firmware update capability. Provided as a GUI and a command-line tool. Depending on the ecosystem you target, there may be other over-the-air firmware update tools available.	user guide
proxy-helper	Command-line tool for configuring proxy settings.	See -h help.
lcs-manager-cli	Command-line tool to create local content to work without Internet.	user guide

1.3.2.3 Utilities

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

Utility	Description
GCC	Supported toolchain included with the ModusToolbox™ installer.
GDB	The GNU Project Debugger is installed as part of GCC.
JRE	Java Runtime Environment; required by the Eclipse IDE integration layer.
SRecord	Collection of tools for manipulating EPROM load files. This is used to merge multi-core application images into a combined programmable HEX image.

1.3.2.4 Build system infrastructure

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

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

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

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

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

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

The build system implements the `make getlibs` command. This command finds each *.mtb* file, clones the specified repository, checks out the specified commit, and collects all the files into the specified directory. Typically, the `make getlibs` command is invoked transparently when you create an application or use the

Introduction

Library Manager, although you can invoke the command directly from a command line interface. See [ModusToolbox™ build system](#) for detailed documentation on the build system infrastructure.

1.3.2.5 Program and debug support

ModusToolbox™ software supports the [Open On-Chip Debugger](#) (OpenOCD) using a GDB server, and supports the J-Link debug probe.

You can use various IDEs to program devices and establish a debug session (see [Using supported IDEs](#)). For programming, [CYPRESS™ Programmer](#) is available separately. It is a cross-platform application for programming PSoC™ 6 devices. It can program, erase, verify, and read the flash of the target device.

Cypress Programmer and the Eclipse IDE use KitProg3 low-level communication firmware. The firmware loader (fw-loader) is a software tool you can use to update KitProg3 firmware, if you need to do so. The fw-loader tool is installed with the ModusToolbox™ software. The latest version of the tool is also available separately in a [GitHub repository](#).

Tool	Description	Documentation
CYPRESS™ Programmer	CYPRESS™ Programmer functionality is built into ModusToolbox™ Software. CYPRESS™ Programmer is also available as a stand-alone tool.	Programming tools page, go to the documentation tab
fw-loader	A simple command line tool to identify which version of KitProg is on a kit, and easily switch back and forth between legacy KitProg2 and current KitProg3.	<i>readme.txt</i> 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

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 through the ModusToolbox™ installation package.

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).

Introduction

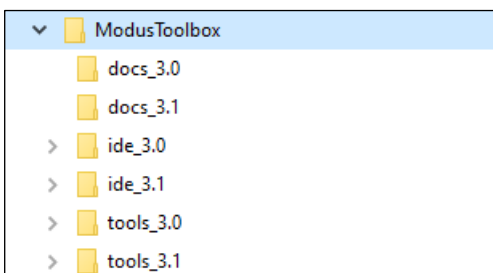
Code Examples include various libraries automatically. Prior to the ModusToolbox™ 2.3 release, these libraries were typically the latest versions. From the 2.3 release and newer, when you create a new application from a code example, any of the included libraries specified with a "latest-style" tag are 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 *locking_commit.log* in the *deps* subdirectory inside your application directory. This file maintains a history of all latest to release conversions made to ensure consistency with any libraries added in the future.

1.4.2 Tools package versioning

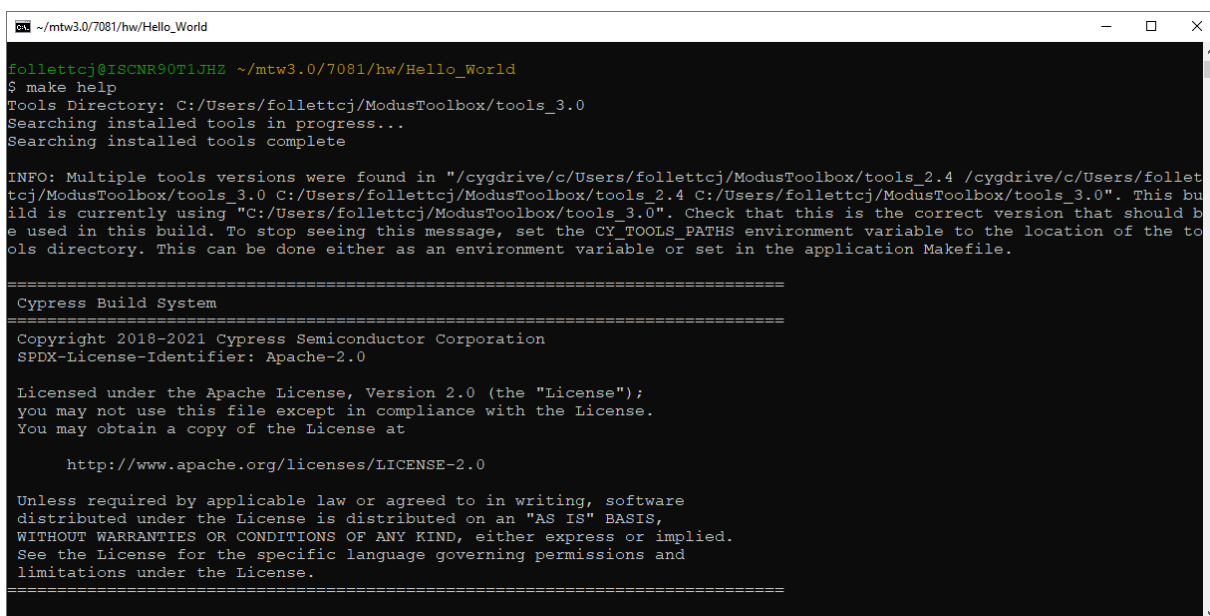
The ModusToolbox™ tools installation package is versioned as MAJOR.MINOR.PATCH. The file located at `<install_path>/ModusToolbox/tools_3.1/version-3.1.0.xml` also indicates the build number.

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, as follows:



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.



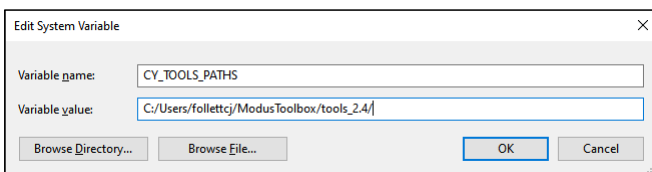
Introduction

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 3.1 and 3.0 installed, and if you launch the Eclipse IDE from the ModusToolbox™ 3.0 installation, the IDE will use the tools from the `tools_3.1` directory to launch configurators and build an application. This section describes how to specify the path to the desired version.

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:



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

`C:/Users/XYZ/ModusToolbox/tools_2.4/`

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

1.4.4.2 Specific project Makefile

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

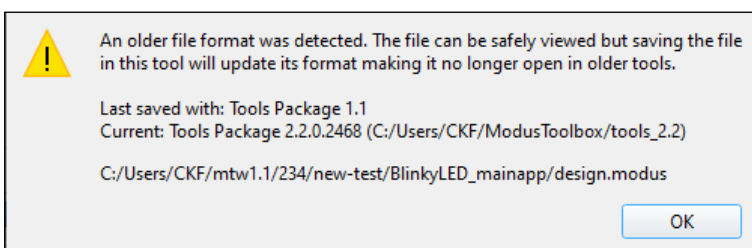
```
# If you install the IDE in a custom location, add the path to its
# "tools_X.Y" folder (where X and Y are the version number of the tools
# folder).
CY_TOOLS_PATHS+=C:/Users/XYZ/ModusToolbox/tools_2.3
```

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.

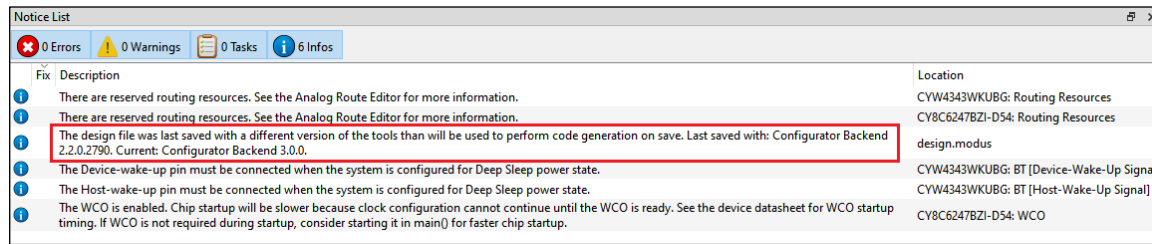
1.4.5.1 Configurator messages

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:



Introduction

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

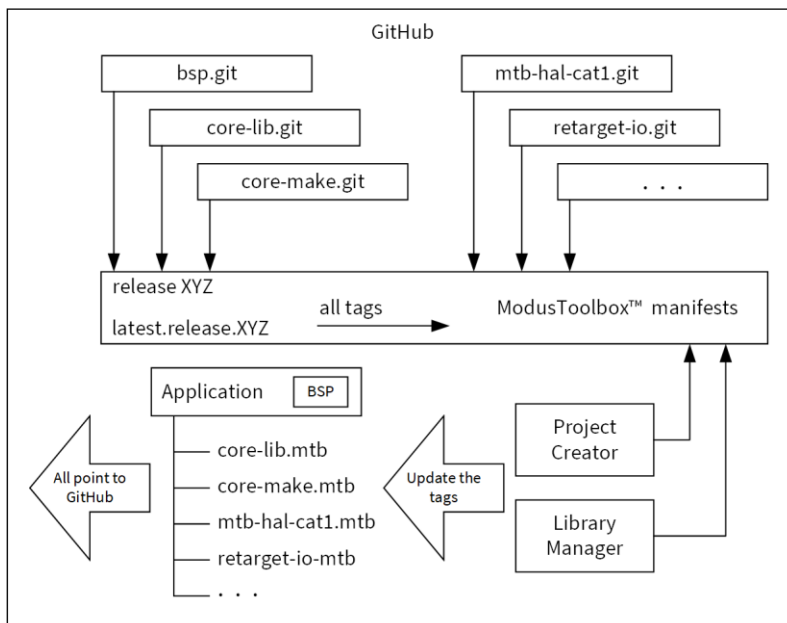
GitHub libraries follow the same versioning scheme: MAJOR.MINOR.PATCH. The GitHub libraries, besides the code itself, also provide two files in MD format: README and RELEASE. The latter includes the version and the change history.

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.



Introduction

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.

1.5 Partner ecosystems

To support Infineon microcontrollers in our partner ecosystems, some tools and middleware from ModusToolbox™ software are also integrated into Amazon FreeRTOS. Refer to <https://aws.amazon.com/freertos/> to learn more about developing applications in those environments.

Getting started

2 Getting started

ModusToolbox™ software provides various graphical user interface (GUI) and command-line interface (CLI) tools to create and configure applications the way you want. You can use the included Eclipse-based IDE, which provides an integrated flow with all the ModusToolbox™ tools. Or you can use other IDEs, such as VS Code, 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™ tools package is located on our website:

<https://www.infineon.com/cms/en/design-support/tools/sdk/modustoolbox-software/>

You can install the software on Windows, Linux, and macOS. Refer to the [ModusToolbox™ tools package installation guide](#) for specific instructions.

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

2.1.1 GUI set-up instructions

In general, the IDE and other GUI-based tools included as part of the ModusToolbox™ tools package work out of the box without any changes required. Simply launch the executable for the applicable GUI tool. On Windows, most tools are on the **Start** menu.

2.1.2 CLI set-up instructions

Before using the CLI tools, ensure that the environment is set up correctly. 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 from the **Start** menu **ModusToolbox 3.1 > modus-shell**, or type "modus-shell" in the Windows search box.

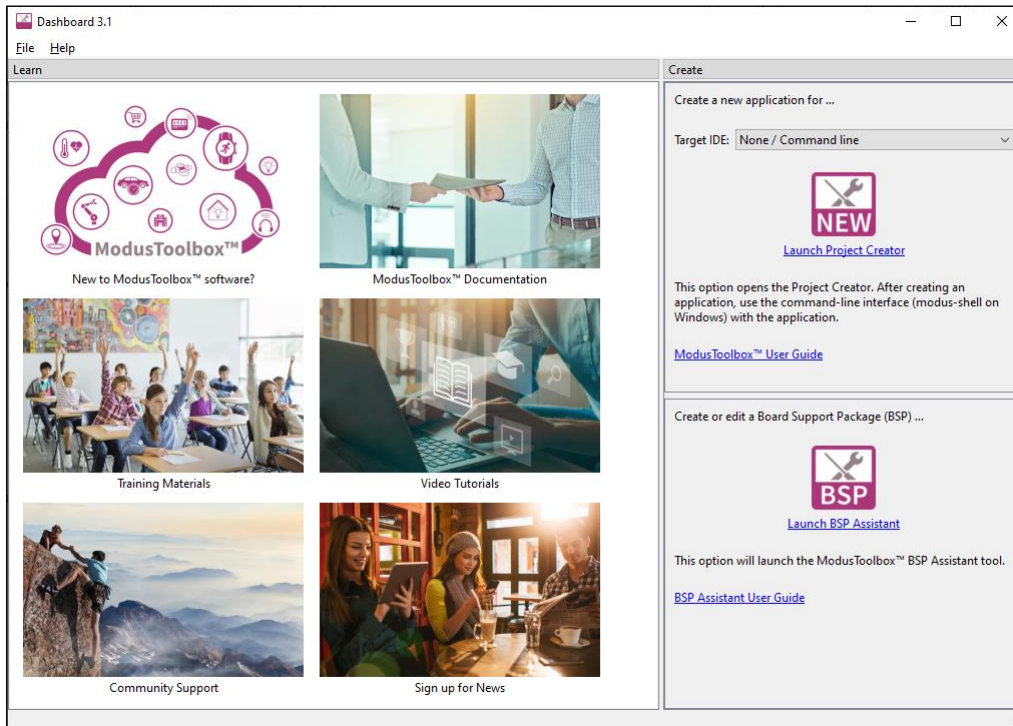
- Type `which make`. For most environments, it should return `/usr/bin/make`.
- Type `which git`. For most environments, it should return `/usr/bin/git`.

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.

Getting started

2.2 Launch Dashboard

Beginning with version 3.1.0, the ModusToolbox™ tools package includes a new tool called the Dashboard. On Windows, you can launch this optional tool from the last step of the installer. 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 two starting points: create a new application and create/edit a BSP.

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.

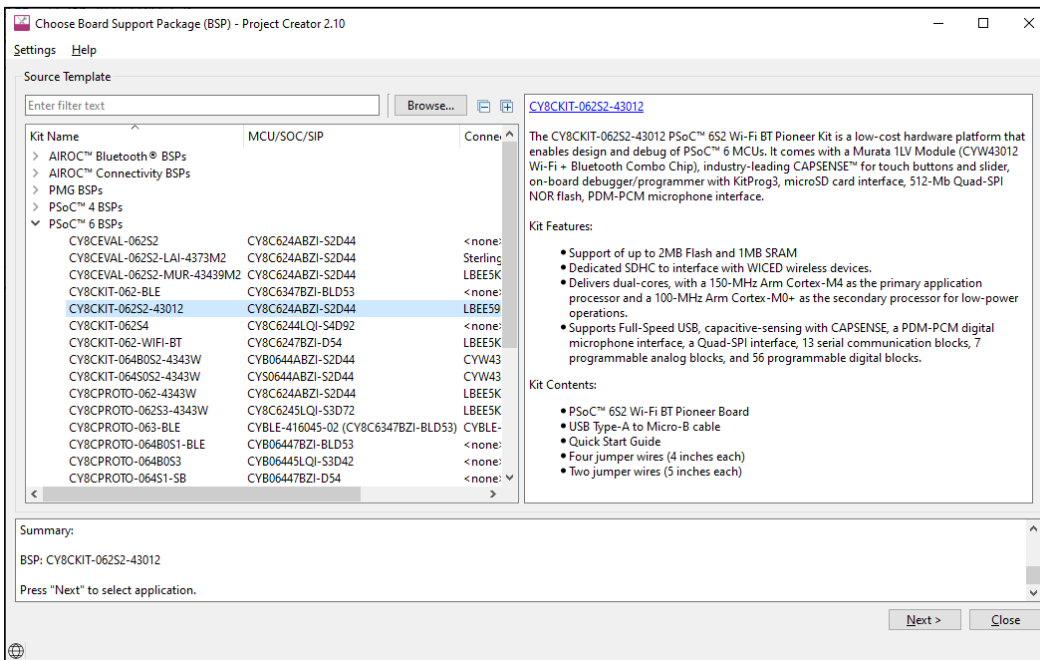
Getting started

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_3.1/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 included Eclipse-based IDE, 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_3.1/project-creator/`). To see all the options available, run the tool with the `-h` option:

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

The following example shows running the tool with various options.

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

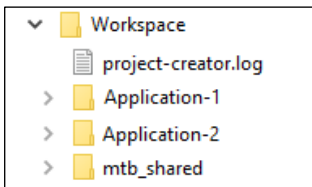
Getting started

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.



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

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

Item	Version 2.x	Version 3.x
BSP Assistant usage	Not applicable	Creates and updates 3.x BSPs
Default BSP type	Git repo, to make changes requires custom BSP	Application-owned, can be directly modified
Local BSP location	Under the <code>libs</code> directory	Under the <code>bsps</code> directory
<code>design.modus</code> file location	<code>libs/COMPONENT_BSP_DESIGN_MODUS</code> subdirectory	<code>bsps/config</code> subdirectory
<code>Makefile</code> <code>MTB_TYPE</code> variable	Not applicable	Identifies single-core vs. multi-core applications

This user guide focuses on the 3.x application structure. For more details about 2.x applications and BSPs, refer to the older revision of this user guide, located in the `docs_2.4` directory of the ModusToolbox™ 2.4 installation.

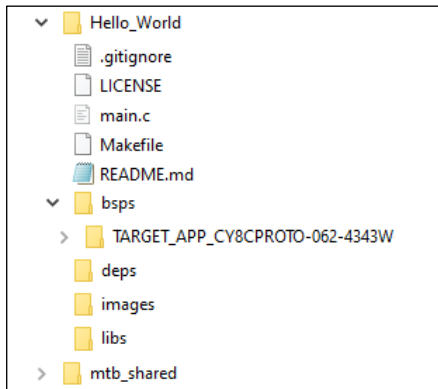
To take full advantage of the newest features, you can easily migrate version 2.x applications to the 3.x structure following Knowledge Base Article [KBA236134](#). This KBA provides instructions for replacing your BSP and associated libraries with compatible versions for 3.x.

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

Getting started

2.4.2 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:

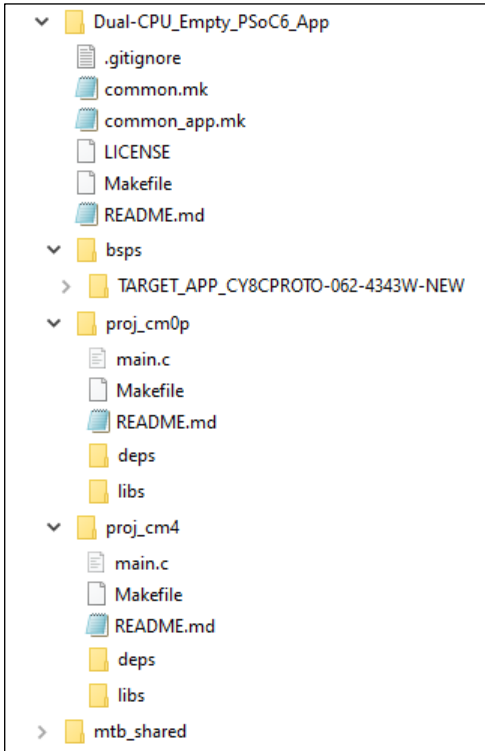
- **.gitignore file** – This file contains information about files for Git to ignore such as common, tool- or user-specific files that are typically not checked into a version control system.
- **LICENSE file** – This is the license agreement.
- **Source code** – This is one or more files for your project's 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`.
- **README.md file** – This file describes the code example that was used to create the project.
- **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 *locking_commit.log* 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.

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.

Getting started

2.4.3 Multi-core 3.x application

A multi-core 3.x application, such as "Dual-CPU_Empty_PSoC6_App," includes three 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).



2.4.3.1 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".

Getting started

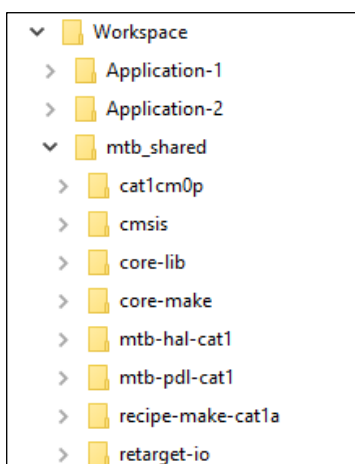
2.4.3.2 Multi-core project directories

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 *locking_commit.log* file, which keeps track of the version for each dependent library.
- **libs subdirectory** – This subdirectory may contain different types of files generated by the 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.

2.4.4 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.



Getting started

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, CY8CPROTO-062-4343W
APPNAME	Specifies the name of the application
TOOLCHAIN	Specifies the build tools used to build the application
CONFIG	Specifies the configuration option for the build [Debug Release]
VERBOSE	Specifies whether the build is silent or verbose [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
GCC_ARM	GNU Arm Embedded Compiler	macOS, Windows, Linux
ARM	Arm compiler	Windows, Linux
IAR	Embedded Workbench	Windows

In the *Makefile*, set the `TOOLCHAIN` variable to the build tools of your choice. For example:
`TOOLCHAIN=GCC_ARM`. There are also variables you can use to pass compiler and linker flags to the toolchain.

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

2.5.1 Use an IDE

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

- [Eclipse IDE for ModusToolbox™ user guide](#)
- [Visual Studio Code for ModusToolbox™ user guide](#)
- [Keil μVision for ModusToolbox™ user guide](#)
- [IAR Embedded Workbench for ModusToolbox™ 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
```

Getting started

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

```
make build -j16
```

2.5.2.2 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.

Updating the example application

3 Updating the example application

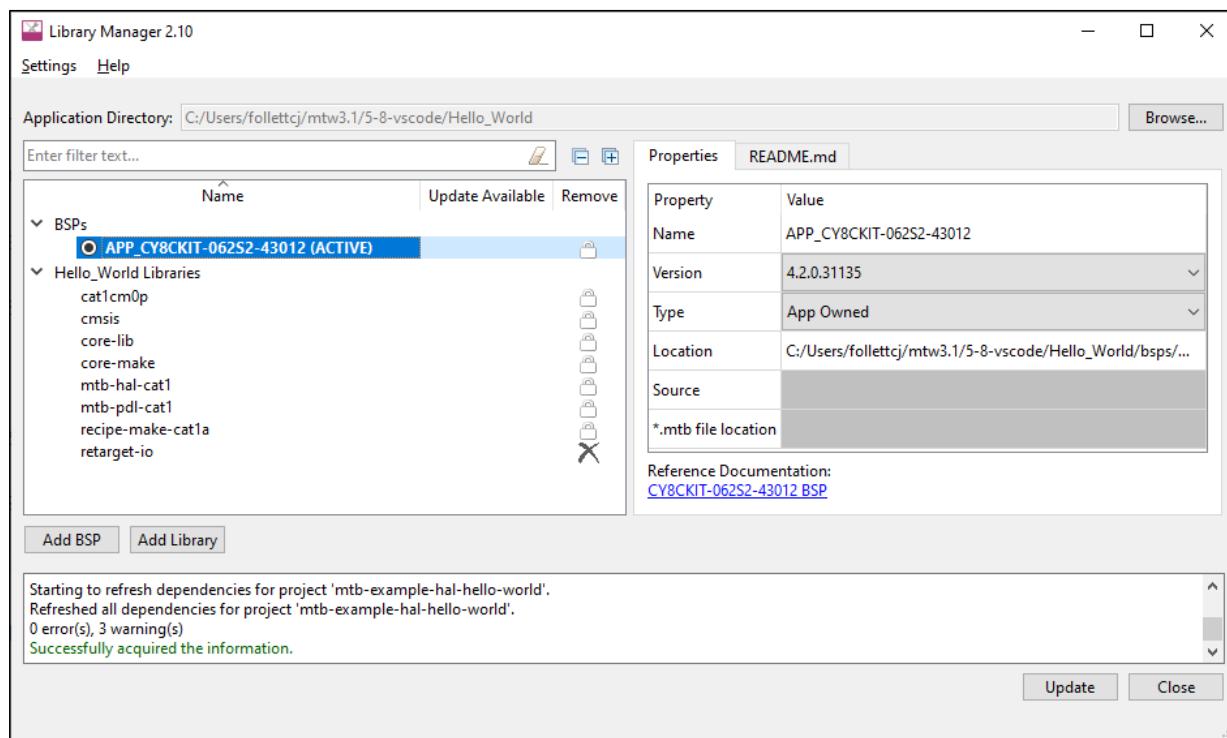
After completing the process to create and build an example application and program a board, you may wish to update the application in various ways to explore its capabilities. This chapter covers some of the basic tasks, including:

- [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.



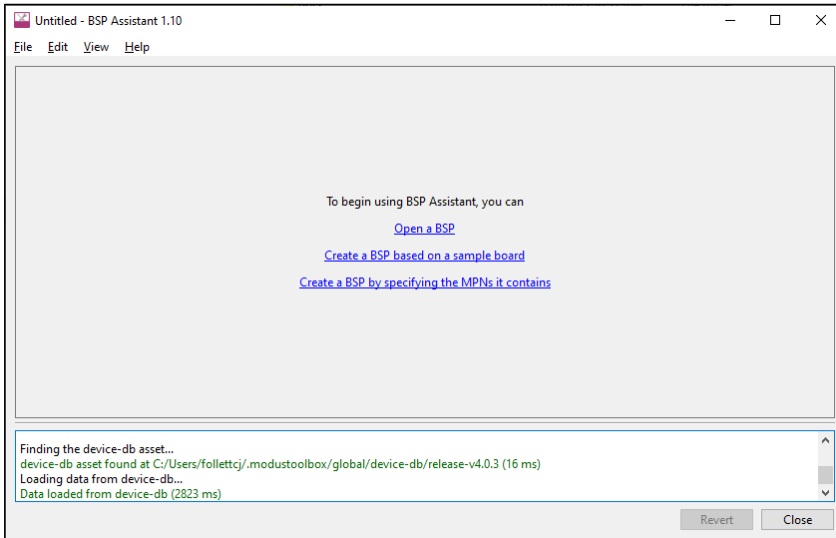
Note: There are several ways to open the Library Manager; refer to the [Library Manager user guide](#) for more details.

Updating the example application

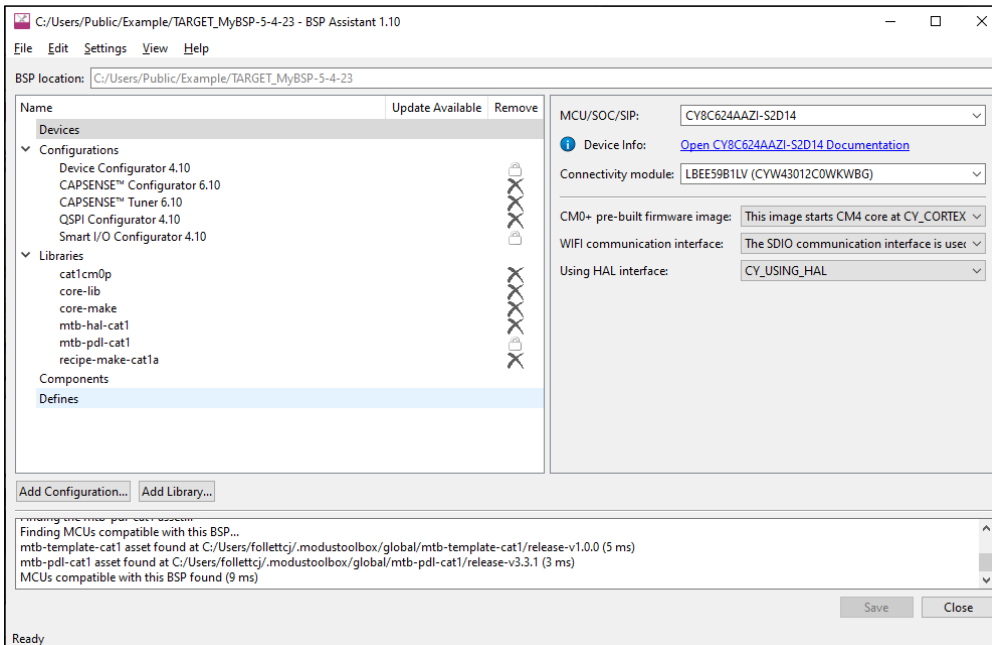
3.2 Create/edit BSPs

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.



Updating the example application

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; see [Application layers](#). That decision affects how you configure settings for devices, peripherals, and libraries.

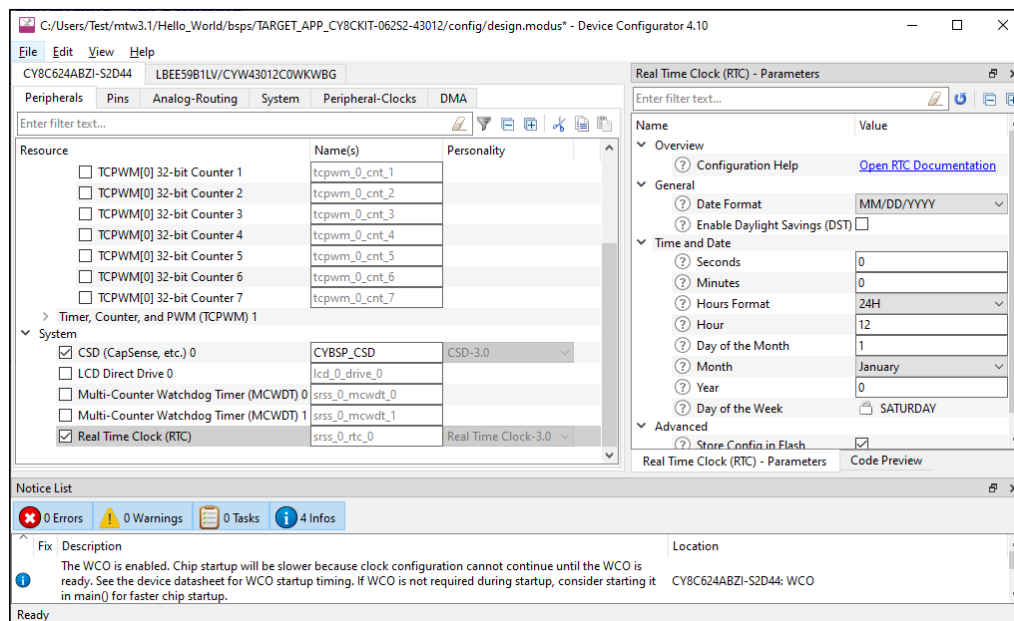
The configurators can be run as GUIs to easily update various parameters and settings. Most can also be run as command line tools to regenerate code as part of a script. For more information about configurators, see the [Configurators](#) section. Also, each configurator provides a separate document, 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 open various configurator GUIs using the appropriate make command from the application directory. For example, to open the Device Configurator, run:

```
make device-configurator
```

This opens 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
```

You can also use the Eclipse IDE provided with ModusToolbox™ software to open configurators. For example, if you select the "Device Configurator" link in the IDE Quick Panel, the tool opens with the application's *design.modus* file. Refer to the [Eclipse IDE for ModusToolbox™ user guide](#) for more details about the Eclipse IDE.

Updating the example application

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.

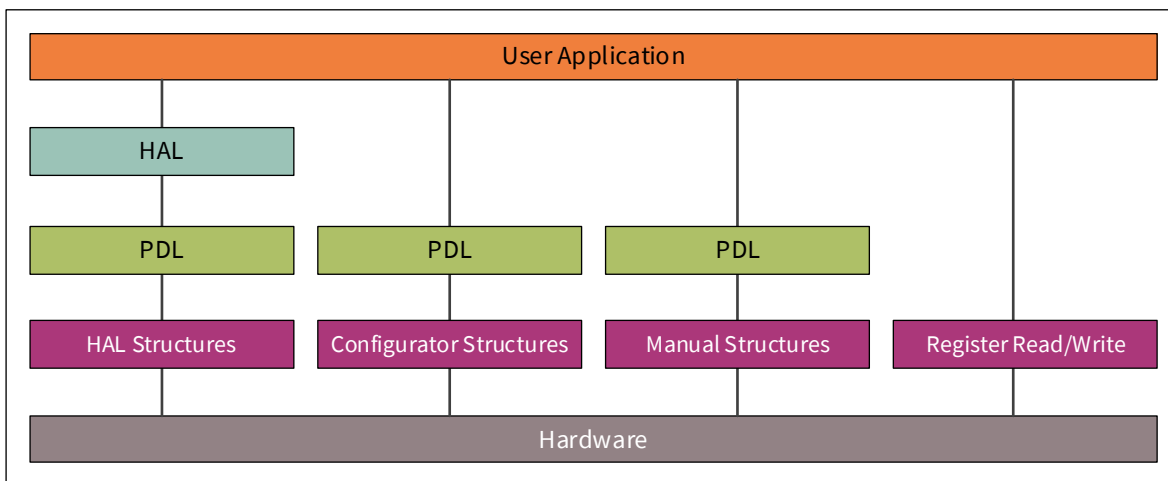
3.4 Write application code

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; see [Application layers](#).

ModusToolbox™ software is designed to enable your workflow. It includes an integrated Eclipse IDE, as well as support for Visual Studio (VS) Code, IAR Embedded Workbench, and Keil μVision (see [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.

3.4.1 Application layers

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



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

Note: A single application may use different methods for different peripherals.

Updating the example application

3.4.1.1 HAL

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

The advantages include:

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

The disadvantages include:

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

3.4.1.2 PDL

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

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

3.4.1.3 Configurators

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

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

3.5 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 or command line options to debug the application.

Updating the example application

3.5.1 Use Eclipse IDE

When using the provided Eclipse IDE, 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 Use command line

When debugging via command line, use the following commands, as applicable:

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

4 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](#)

4.1 Overview

The ModusToolbox™ build system is based on GNU make. It performs application builds and provides the logic required to launch tools and run utilities. It consists of a light and accessible set of *Makefiles* deployed as part of every application. This structure allows each application to own the build process, and it allows environment-specific or application-specific changes to be made with relative ease. The system runs on any environment that has the make and git utilities.

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 included with the ModusToolbox™ installer.

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

The majority of the *Makefiles* are deployed as git repositories (called "repos"), in the same way that libraries are deployed in the ModusToolbox™ software. There are two separate repos: *core-make* used by all recipes and a *recipe-make-xxx* that contains BSP/target specific details. These are the minimum required to enable an application build. Together, these *Makefiles* form the build system.

ModusToolbox™ build system

4.2 make help

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

```
make help
```

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

```
make help CY_HELP=TOOLCHAIN
```

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

4.3 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` 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.

4.3.1 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](#).

ModusToolbox™ build system

4.4 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 add, update, or remove a BSP from an application. You can also add a *.mtb* file that contains the URL and a version tag of interest in the application.

4.5 Environment variables

ModusToolbox™ software supports custom installation paths, and we provide the following variables to specify locations of tools and support files other than the default:

- `CY_TOOLS_PATHS` (path to the installation "tools_<version>" directory)
- `CyManifestLocOverride` (path to the local *manifest.loc* file)
- `CyRemoteManifestOverride` (URL to a specific manifest file)

For ModusToolbox™ version 3.x, we also include a global path for assets like device-db using the variable named `CY_GETLIBS_GLOBAL_PATH`. If the variable does not exist, it assumes a default path of `~/.modustoolbox/global`.

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_3.1"`.

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

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

4.6.1 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.

ModusToolbox™ build system

4.6.1.1 .cyignore

Prior to applying auto-discovery and filtering, the build system will first search for *.cyignore* files and construct a set of directories and files to exclude. It contains a set of directories and files to exclude, relative to the location of the *.cyignore* file. The *.cyignore* file can contain make variables. For example, you can use the `SEARCH_` variable to exclude code from other libraries as shown for the "Test" directory in a library called `<library-name>`:

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

The `CY_IGNORE` variable can also be used in the *Makefile* to define directories and files to exclude.

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

4.6.1.2 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`. All others are ignored. ModusToolbox™ supports IAR, ARM, and `GCC_ARM`.

4.6.1.3 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=CY8CPROTO-062-4343W` ignores all `TARGET_` directories except `TARGET_CY8CPROTO-062-4343W`.

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

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

4.6.1.4 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.

ModusToolbox™ build system

4.6.1.5 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.

4.6.1.6 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.

4.7 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.

ModusToolbox™ build system

4.8 Available make targets

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

4.8.1 General make targets

Target	Description
all	Same as build. That is, builds the application. This target is equivalent to the "build" target.
getlibs	Clones the repositories and checks out the identified commit. When using .mtb files, the repos are cloned to the shared location \$(CY_GETLIBS_SHARED_PATH) / \$(CY_GETLIBS_SHARED_NAME) . By default, this directory is specified by the project Makefile.
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.
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..
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.

ModusToolbox™ build system

Target	Description
prebuild	Generates code for the application. Runs configurators and custom prebuild commands to generate source code.

4.8.2 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.
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 <i>settings.json</i> .
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. <i>Note:</i> Project generation requires Python 3 to be installed and present in the PATH variable. <i>Note:</i> For applications that were created using core-make-3.0 or older, you must use the <i>make ewarm8</i> 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 <i>make uvision</i> . <i>Note:</i> Project generation requires Python 3 to be installed and present in the PATH variable. <i>Note:</i> For applications that were created using core-make-3.0 or older, you must use the <i>make uvision5</i> command instead.

4.8.3 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
library-manager	Launches the Library Manager for the application to add/remove libraries and to upgrade/downgrade existing libraries.
bsp-assistant	Launches the BSP Assistant with the active BSP for the application.
device-configurator	Launches the Device Configurator on the application's *.modus file.
bt-configurator	Launches the Bluetooth® Configurator GUI for the application's cybth 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.
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.

ModusToolbox™ build system

Target	Description
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.

4.8.4 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.
printlibs	Prints the status of the library repos. This target parses through the library repos and prints the SHA1 commit. It also shows whether the repo is clean (no changes) or dirty (modified or new files).
check	Checks for the necessary tools. Not all tools are necessary for every build recipe. This target allows you to get an idea of which tools are missing if a build fails in an unexpected way.

4.9 Available make variables

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

4.9.1 Basic configuration make variables

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

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

Variable	Description
TARGET	Specifies the target board/kit (that is, BSP). For example, CY8CPROTO-062-4343W. Example usage: <code>make build TARGET=CY8CPROTO-062-4343W</code>
CORE	Specifies the name of the Arm core for which a project is building (e.g. CM4). Example Usage: <code>make build CORE=CM4</code> 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: <code>make build CORE_NAME=CM7_0</code> Use this variable to select compiler and linker options to build a project for a specified on-chip core. <i>Note: This variable is applicable for some multi-core devices only (e.g. XMC7xxx).</i>
APPNAME	Specifies the name of the application. For example, "AppV1" > AppV1.elf. Example usage: <code>make build APPNAME="AppV1"</code> This variable is used to set the name of the application artifact (programmable image). <i>Note: This variable may also be used when generating launch configs when invoking the eclipse target.</i>

ModusToolbox™ build system

Variable	Description
TOOLCHAIN	<p>Specifies the toolchain used to build the application. For example, GCC_ARM.</p> <p>Example Usage: <code>make build TOOLCHAIN=IAR</code></p> <p>Supported toolchains for this include GCC_ARM, IAR, and ARM.</p> <p><i>Note: 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.</i></p>
CONFIG	<p>Specifies the configuration option for the build [Debug Release].</p> <p>Example Usage: <code>make build CONFIG=Release</code></p> <p>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.</p> <p>The optimization flags are toolchain specific. If you go with your custom config, then you can manually set the optimization flag in the CFLAGS.</p>
VERBOSE	<p>Specifies whether the build is silent [false] or verbose [true].</p> <p>Example Usage: <code>make build VERBOSE=true</code></p> <p>Setting VERBOSE to true may help in debugging build errors/warnings. By default, it is set to false.</p>

4.9.2 Advanced configuration make variables

These variables define advanced aspects of building an application.

Variable	Description
SOURCES	<p>Specifies C/C++ and assembly files outside of application directory.</p> <p>Example Usage (within Makefile): <code>SOURCES+=path/to/file/Source1.c</code></p> <p>This can be used to include files external to the application directory. The path can be both absolute or relative to the application directory.</p>
INCLUDES	<p>Specifies include paths outside of the application directory.</p> <p>Example Usage (within Makefile): <code>INCLUDES+=path/to/headers</code></p> <p><i>Note: These MUST NOT have -I prepended. The path can be either absolute or relative to the application directory.</i></p>
DEFINES	<p>Specifies additional defines passed to the compiler.</p> <p>Example Usage (within Makefile): <code>DEFINES+=EXAMPLE_DEFINE=0x01</code></p> <p><i>Note: These MUST NOT have -D prepended.</i></p>
VFP_SELECT	<p>Selects hard/soft ABI or full software for floating-point operations [softfp hardfp softfloat].</p> <p>If not defined, this value defaults to softfp.</p> <p>Example Usage (within Makefile): <code>VFP_SELECT=hardfp</code></p>
VFP_SELECT_PRECISION	<p>Selects single-precision or double-precision operating mode for floating-point operations.</p> <p>If not defined, this value defaults to double-precision. Enable single-precision mode by using the "singlefp" option.</p> <p>Example Usage (within Makefile): <code>VFP_SELECT_PRECISION=singlefp</code></p>
CFLAGS	<p>Prepends additional C compiler flags.</p> <p>Example Usage (within Makefile): <code>CFLAGS+= -Werror -Wall -O2</code></p>

ModusToolbox™ build system

Variable	Description
CXXFLAGS	Prepends additional C++ compiler flags. Example Usage (within <i>Makefile</i>): <code>CXXFLAGS+= -finline-functions</code>
ASFLAGS	Prepends additional assembler flags. Usage is similar to <code>CFLAGS</code> .
LDFLAGS	Prepends additional linker flags. Example Usage (within <i>Makefile</i>): <code>LDFLAGS+= -nodefaultlibs</code>
LINKER_SCRIPT	Specifies a custom linker script location. Example Usage (within <i>Makefile</i>): <code>LINKER_SCRIPT=path/to/file/Custom_Linker1.ld</code> This linker script overrides the default. <i>Note:</i> Additional linker scripts can be added for GCC via the <code>LDFLAGS</code> variable as a <code>-L</code> option.
COMPONENTS	Adds component-specific files to the build. Example Usage (within <i>Makefile</i>): <code>COMPONENTS+=CUSTOM_CONFIGURATION</code> Create a directory named <code>COMPONENT_<VALUE></code> and place your files. Then include the following make variable to have that feature library be included in the build. For example, create a directory named <code>COMPONENT_ACCELEROMETER</code> into auto-discovery. Then add the following make variable to the <i>Makefile</i> : <code>COMPONENT=ACCELEROMETER</code> . If the make variable does not include the <code><VALUE></code> , then that library will not be included in the build.
DISABLE_COMPONENTS	Removes component-specific files from the build. Example Usage (within <i>Makefile</i>): <code>DISABLE_COMPONENTS=BSP_DESIGN_MODUS</code> Include a <code><VALUE></code> to this make variable to have that feature library be excluded in the build. For example, to exclude the contents of the <code>COMPONENT_BSP_DESIGN_MODUS</code> directory, set <code>DISABLE_COMPONENTS=BSP_DESIGN_MODUS</code> .
SEARCH	List of paths to include in auto-discovery. For example, <code>../mtb_shared/lib1</code> . Example Usage (within <i>Makefile</i>): <code>SEARCH+=directory_containing_source_files</code> The <code>SEARCH</code> 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 <i>Makefile</i> to override the set of projects to generate combined hex file from.

4.9.3 BSP make variables

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

ModusToolbox™ build system

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.

4.9.4 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_3.1."`

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 <code>.mtb</code> files. Setting this path allows relocating the directory that the library-manager uses to store the <code>.mtb</code> files in your application. The default location is in a directory named <code>deps</code> .
CY_GETLIBS_SHARED_PATH	Relative path to the shared repo location. All <code>.mtb</code> files have the format, <code><URI><COMMIT><LOCATION></code> . If the <code><LOCATION></code> field begins with <code>\$\$ASSET_REPO\$\$</code> , then the repo is deposited in the path specified by the <code>CY_GETLIBS_SHARED_PATH</code> variable. The default is set from the project <i>Makefile</i> .
CY_GETLIBS_SHARED_NAME	Directory name of the shared repo location. All <code>.mtb</code> files have the format, <code><URI><COMMIT><LOCATION></code> . If the <code><LOCATION></code> field begins with <code>\$\$ASSET_REPO\$\$</code> , then the repo is deposited in the directory specified by the <code>CY_GETLIBS_SHARED_NAME</code> variable. By default, this is set from the project <i>Makefile</i> .

ModusToolbox™ build system

4.9.5 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_3.1."

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 <i>Makefile</i> .
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 <i>Note: When set in the Makefile, no quotes are required.</i>
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 <i>Note: When set in the Makefile, no quotes are required.</i>
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 <i>Note: When set in the Makefile, no quotes are required.</i>
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 <i>tools_<version></i> directory location, which contains the root <i>Makefile</i> and the necessary tools and scripts to build an application. Application <i>Makefiles</i> 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.

ModusToolbox™ build system

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

4.9.6 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_3.1."

Variable	Description
CY_IGNORE	<p>Adds to the directory and file ignore list. For example, <code>./file1.c./inc1</code></p> <p>Example Usage: <code>make build CY_IGNORE="path/to/file/ignore_file"</code></p> <p>Directories and files listed in this variable are ignored in auto-discovery. This mechanism works in combination with any existing <code>.cyignore</code> files in the application.</p>
CY_SIMULATOR_GEN_AUTO	<p>If set to 1, automatically generate a simulator archive (if supported by the target device).</p> <p>When enabled, the <code>build</code> make target will generate a debugging tgz archive for the Infineon online simulator as part of the postbuild process.</p>

Board support packages

5 Board support packages

5.1 Overview

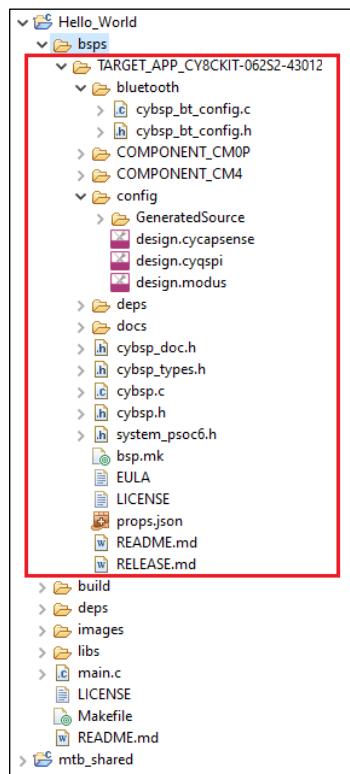
A BSP provides a standard interface to a board's features and capabilities. The API is consistent across our kits. Other software (such as middleware or an application) can use the BSP to configure and control the hardware. BSPs do the following:

- 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.
- include the libraries for the default capabilities on the board. For example, the BSP for a kit with CAPSENSE™ capabilities includes the CAPSENSE™ library.

5.2 What's in a BSP

This section presents an overview of the key resources that are part of a BSP. Applications can share libraries. BSPs are owned by an application. For more details about library management, refer to the [Library Manager user guide](#).

The following shows a typical PSoC™ 6 BSP located in the *bsp* subdirectory.



Board support packages

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

Note: Starting with ModusToolbox 3.x, the `COMPONENT_CUSTOM_DESIGN_MODUS` mechanism will no longer be supported. Thus, the `COMPONENT_DESIGN_MODUS` folder can be removed from the BSP and contents of the folder can be moved to the bsp root directory.

5.2.1 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.

5.2.2 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.

5.2.3 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.

5.2.4 deps subdirectory

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

5.2.5 docs subdirectory

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

5.2.6 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()`.

5.2.7 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. It also defines board-specific information such as the device ID, compiler and linker flags, pre-builds/post-builds, and components used with this board implementation.

5.2.8 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.

Board support packages

5.2.9 BTSDK-specific BSP files

BTSDK BSPs may optionally provide the following types of files:

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

5.3 Creating your own BSP

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.

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.

Manifest files

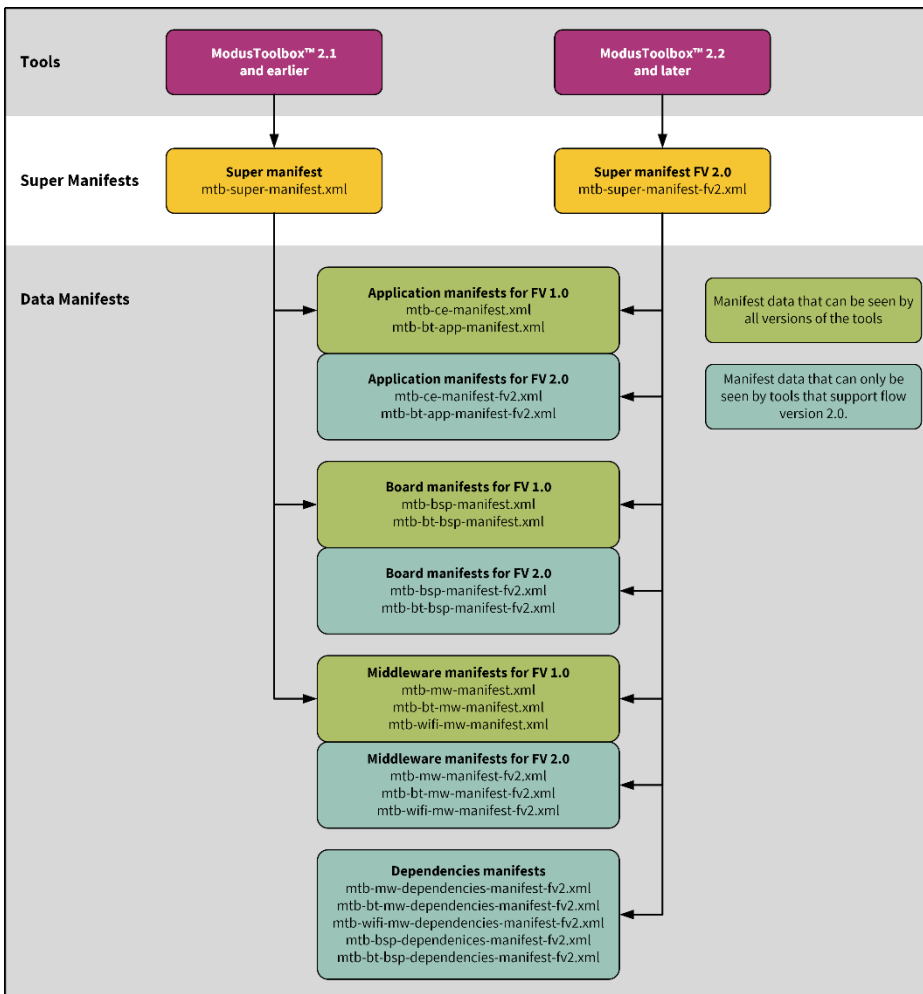
6 Manifest files

6.1 Overview

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

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

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.



Manifest files

6.2 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.

6.2.1 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.

6.2.2 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
```

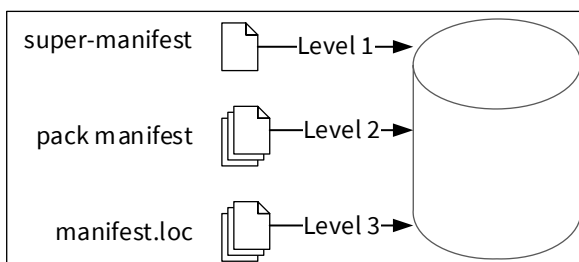
Manifest files

6.2.3 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.



6.2.4 Conflicting data

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

The first kind is a conflict between data that comes from the 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.

6.3 Local content storage

Local content storage (LCS) is a new feature in the ModusToolbox™ 3.1.0 tools package that replaces the older feature called offline content. The 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](#).

Using applications with third-party tools

7 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:

- [Version Control and sharing applications](#)
- [Using supported IDEs](#)
- [Generating files for XMC™ Simulator tool](#)

7.1 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.

7.1.1 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 Manger 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.

7.1.2 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. ModusToolbox™ assets are provided using Git, but you can use any version control method or software that you prefer.

ModusToolbox™ code examples have a default *.gitignore* file that excludes 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. For example, you may want to check in all of the libraries from *libs* and *mtb_shared*, even though they are available on Infineon's GitHub site.

If you are using version control software other than Git, you can use the *.gitignore* file as a guide for configuring the software that you are using.

Using applications with third-party tools

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. Run the Library Manager and click the **Update** button, or open a terminal and run the command `make getlibs`. Either one will get all of the libraries required by the application.
3. 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.

7.1.3 Manual file copy

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

7.1.4 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.

One such example is the Eclipse IDE **Export as Archive**. Refer to the [Eclipse IDE for ModusToolbox™ user guide](#) for more details.

7.2 Using supported IDEs

7.2.1 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.

7.2.2 Eclipse

The easiest way to create a ModusToolbox™ application for Eclipse is to use the Eclipse IDE included with the ModusToolbox™ software. The tools package 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](#).

Using applications with third-party tools

7.2.3 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](#).

7.2.4 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.

7.2.5 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.

7.3 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.

7.3.1 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.

7.3.2 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.*

Using applications with third-party tools

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

- 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.

7.4 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/mtw3.1/5699/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.

Revision history

Revision history

Revision	Date	Description of change
**	2020-03-24	New document.
*A	2020-03-27	Updates to screen captures and associated text.
*B	2020-04-01	Fix broken links.
*C	2020-04-29	Fix incorrect link.
*D	2020-08-28	Updates for ModusToolbox™ 2.2.
*E	2020-09-23	Corrections to Build system and Board support packages chapters.
*F	2020-09-29	Added links to KBAs; updated text for cyignore.
*G	2020-10-02	Added details for BTSDK v2.8 BSPs/libraries.
*H	2021-01-14	Updated Manifest chapter and fixed broken links.
*I	2021-03-23	Updates for ModusToolbox™ 2.3.
*J	2021-05-24	Updated information for creating a custom BSP.
*K	2021-09-27	Updates for ModusToolbox™ 2.4.
*L	2021-11-29	Merged chapter 3 (software overview) into chapter 1 (introduction). Updated sections 6.2.3 and 6.2.4 with notes and minor details. Added section 6.3 with information for patched flashloaders and 3 rd party IDEs.
*M	2022-02-24	Added link to PSoC™ 4 Application Note.
*N	2022-04-07	Updated various links to the Infineon website.
*O	2022-09-29	Updated for version 3.0.
*P	2022-10-06	Updated IAR multi-core instructions for XMC7000 and TRAVEO™ II.
*Q	2022-11-01	Updated IAR export instructions for programming and erasing external memory.
*R	2023-01-23	Update to the BSP chapter to remove duplicate information. Update to the Export to IAR section for XMC1000/XMC4000 devices.
*S	2023-06-02	Updates for version 3.1. Added information for the Dashboard. Updated make variables. Removed information for using 3 rd party IDEs; those instructions are now included in separate user guides. Removed old offline content and cache variables. Added information for Local Content Manager. Added note when setting <code>TOOLCHAIN=IAR</code> that you should also specify the heap type using <code>LDFLAGS</code> .

Trademarks

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

Edition 2023-06-02

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2023 Infineon Technologies AG.

All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

002-29893 Rev. *S

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffensgarantie")

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.