ModusToolbox™ & Friends

About this document

Scope and purpose

This document guides you in learning how you can become a part of the Infineon Partner Program and integrate your software content into the ModusToolbox™ ecosystem.

Intended audience

This document is intended for partners registered in the Infineon Partner Program and who want to integrate their software content into the ModusToolbox™ ecosystem.

Reference documents

See the following documents and sites for more information as needed:

- ModusToolbox™ user guide
- ModusToolbox™ runtime software
- Project Creator user guide
- Library Manager user guide
- https://github.com/Infineon

Table of contents

Table of contents

Table of contents

1 Introduction

[ModusToolbox™ & Friends](#) is a development program which extends the increased productivity, and feature-rich platform of ModusToolbox™ with highly innovative, robust and product ready partner software for developers. By opening ModusToolbox™ to partners, it provides a simple and tested integration of valuable software to developers for easy evaluation and integration to meet their product needs. Each partner owns their own software license, allowing them to maintain control of their engagement model.

To get started with this program, you first need to be a part of the [Infineon Partner Network](#). See upcoming sections for more details.

1.1 What is a partner?

A partner is a company selected by Infineon based on their proven competence and ability to design and deliver strong and trustworthy solutions, especially for new technologies and application fields. This way the partners can be a huge value add to the ModusToolbox™ ecosystem through development kits / production boards, innovative software IP, middleware or system references to showcase their design skills. See [Partner Network](#) webpage for more information.

1.2 Why become a partner?

ModusToolbox™ provides a rich platform with support for multiple IDEs, toolchains, software tools and libraries, development kits, reference examples etc. making it the platform of choice for developers. By being a partner, you can leverage everything ModusToolbox™ has to offer and distribute your own software via ModusToolbox™ to market your product and services directly to developers. See [Partner Brochure](#) to learn more.

1.3 How to become a partner?

To become a part of the partner ecosystem, the process is fairly simple. Go to the [Infineon Partner Network](#) webpage and click "Join the program" to become a partner. The partner management team will evaluate how you can contribute to the network and help with onboarding.

2 ModusToolbox™ software overview

ModusToolbox™ software is a modern, extensible development environment supporting a wide range of Infineon microcontroller devices, wireless connectivity devices, and sensors. 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. For more information, refer to the ModusToolbox ™ user guide.

2.1 Types of software content

ModusToolbox™ comprises mainly three types of software content that are hosted online using Git repositories (Infineon GitHub and third-party Git servers):

- Code examples
- Middleware
- Board Support Packages (BSPs)

As a partner, you can contribute to any or all of these software types. An illustration of how all these software types interact with each other is shown below:
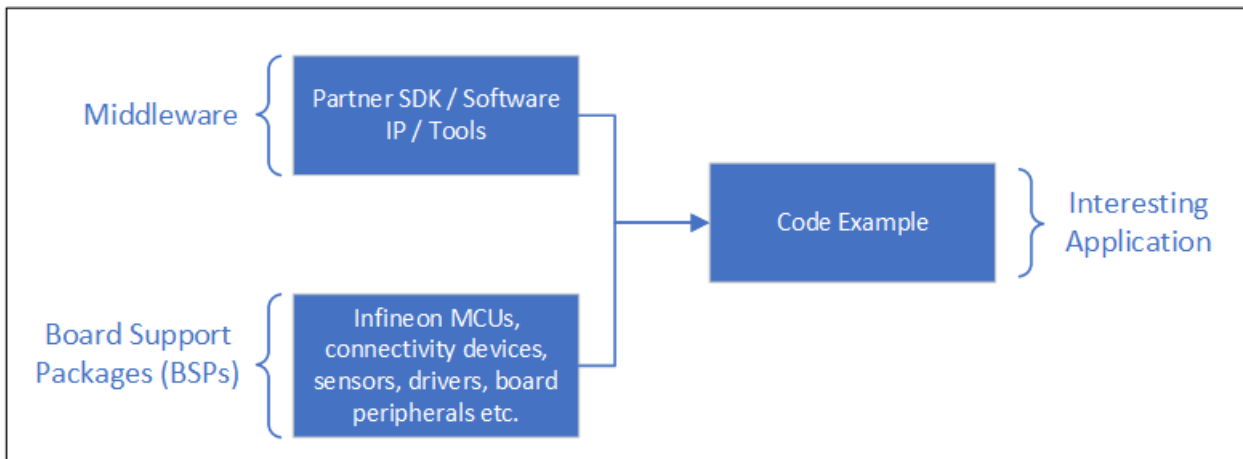
In the upcoming sections, we discuss each of the software types in brief and the use-cases which mandate the creation of content in these categories. To understand more about these software types in detail, refer to the ModusToolbox™ user guide and ModusToolbox™ run-time software reference guide.

2.1.1 Code examples

The code examples are ModusToolbox™ applications intended to demonstrate the usage of a particular product, feature, or tool. Users should be able to easily consume these examples by copying and pasting portions of the code into their own applications.

All current ModusToolbox™ code examples can be found through the GitHub code example page. There you will find links to examples for the Bluetooth® SDK, PSoC™ 6 MCU, and PSoC™ 4 device, among others. See Creating a code example for more information.

Let's look at a few demonstrative scenarios to understand when you would develop a code example.

Scenario #1 – Demonstrate integrating your proprietary firmware SDK or software IP: For example, if you are a partner dealing with cutting edge image processing solutions, you would demonstrate how an Infineon microcontroller can be leveraged to interface with your solutions to create some interesting application like face recognition or object detection.
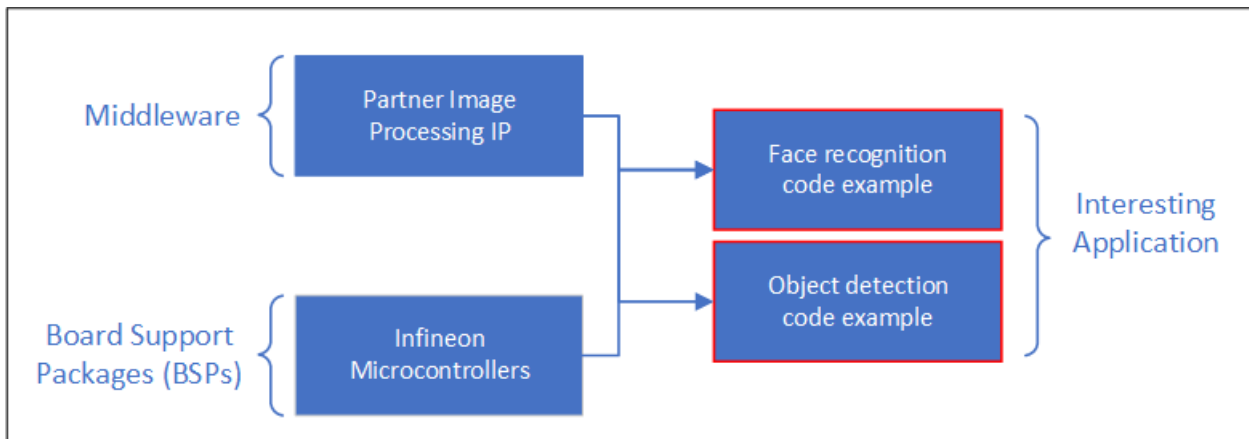
Scenario #2 – Demonstrate your tools to simplify processes: As a partner focused on cutting-edge tools to simplify a particular process, the usage of these tools would normally be explained in a code example. For example, a tool that simplifies creation of graphics on LCD displays may be demonstrated using a graphics-based code example with instructions on tool usage.
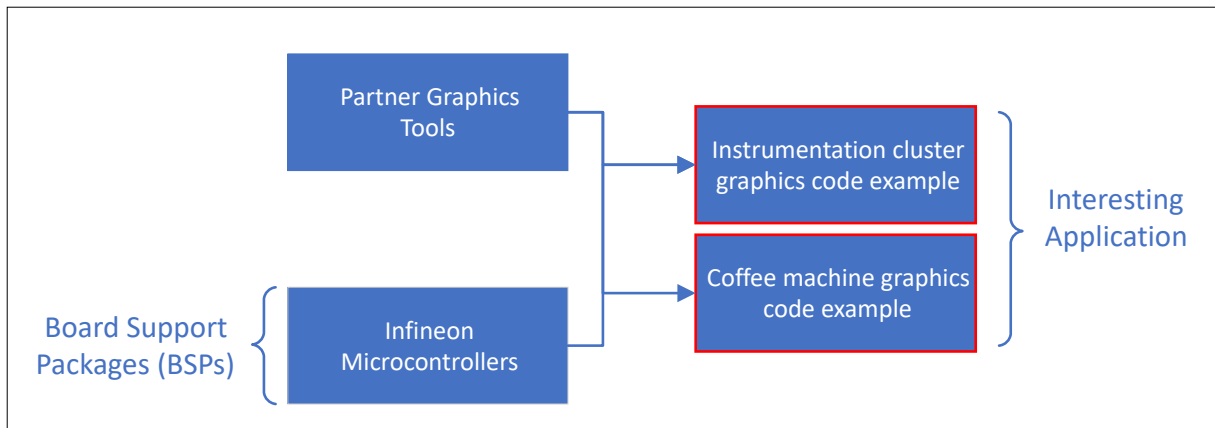
Scenario #3 – Demonstrate your modules: As a partner focused on creating modules or hardware using Infineon chipsets, a code example can be used to demonstrate how the custom module or evaluation kits (called custom BSP) can be interfaced to create interesting applications. For example, a partner focused on creating IoT prototyping modules using Infineon chipsets can develop a code example to demonstrate how to use their module to send data to the cloud.
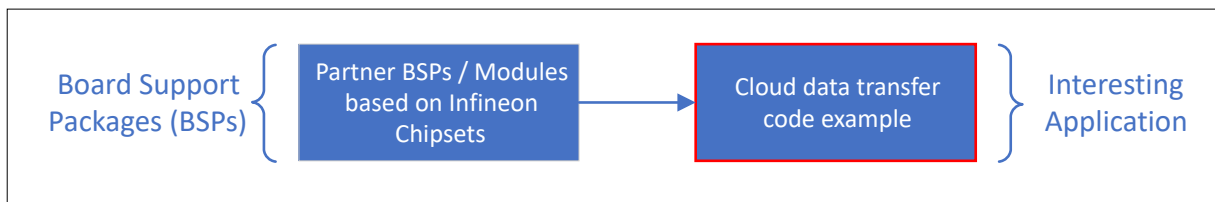
Note: *These scenarios are not an exhaustive list and any number of combinations between the different software types is possible and permitted. If you are unsure where your content belongs, you can contact technical support to seek help. See section* Technical Support *for more details.*

2.1.2   Middleware

A middleware library is usually a wrapper on top of the low-level device driver libraries intended to demonstrate a particular feature or application by further abstracting the hardware level details from the user. For example, an Emulated EEPROM middleware library (emeeprom) operates on top of the Flash driver and allows users to easily store and manage data in an emulated EEPROM memory region.
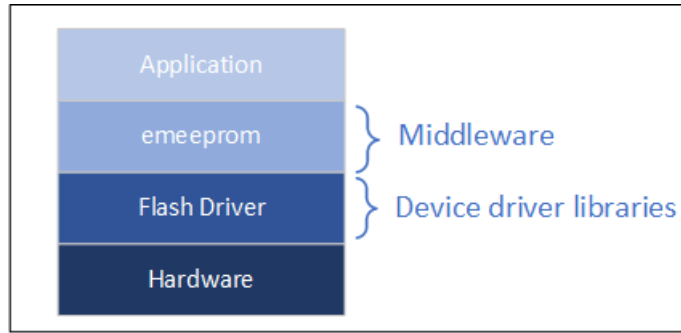
There could be dependencies between several middleware libraries. For example, wifi-mw-core depends on other libraries like abstraction-rtos and clib-support as shown below to work correctly.
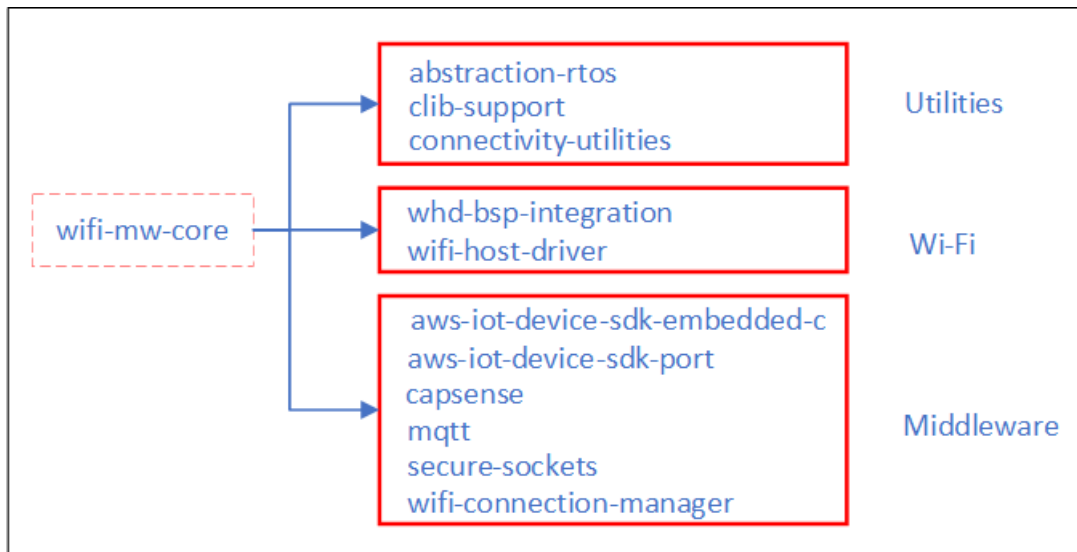
More information on creating your own middleware can be found in Section Creating a middleware library.

Let's look at a few scenarios to understand when you would develop a middleware library.

Scenario #1 - As a partner, your proprietary firmware SDK or software IP usually belongs in the middleware category. To support creation of applications using your firmware SDK or software IP on Infineon microcontrollers, some modifications might be necessary to create porting layers as shown below.
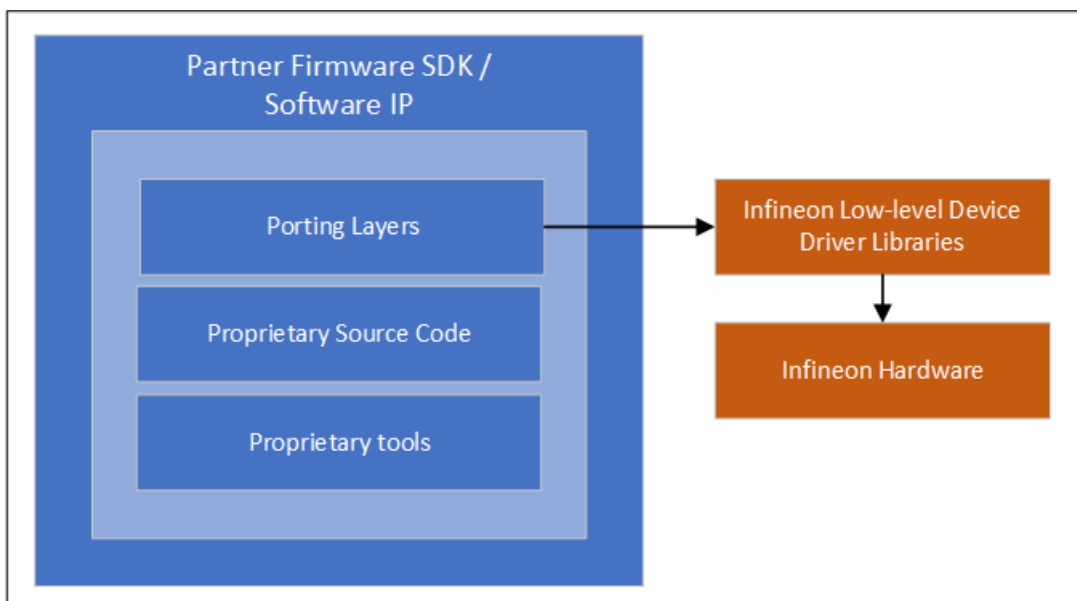
2 ModusToolbox™ software overview

This modified firmware SDK or software IP with the porting layers is then offered as a middleware library that users can leverage in their code examples to interface with Infineon microcontrollers to create interesting applications.
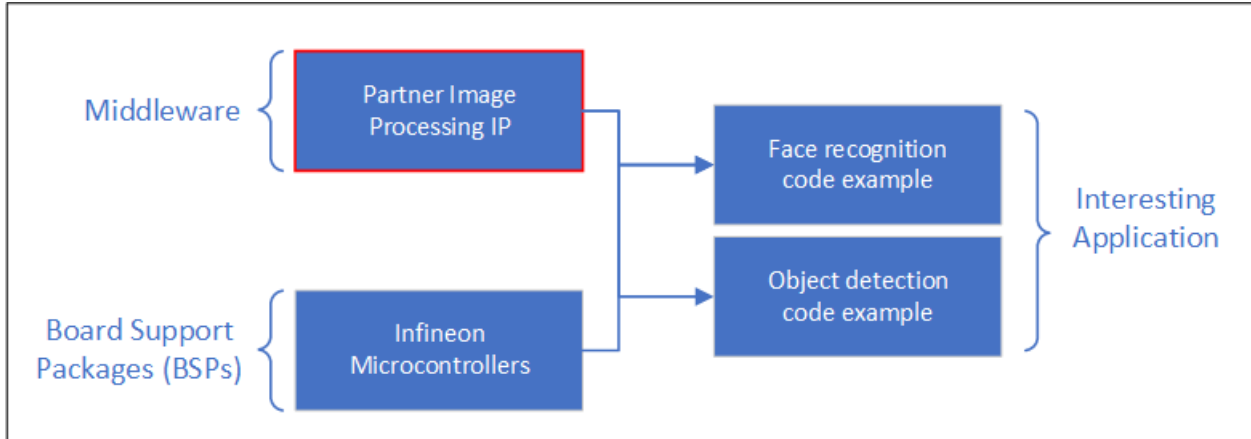
Scenario #2 - If the tools require a particular IP or an SDK to work correctly, the tools should be offered as part of the middleware and not directly as a code example. For example, a partner focused on graphics tools that rely on a software SDK must bundle the SDK and tools together as shown below.
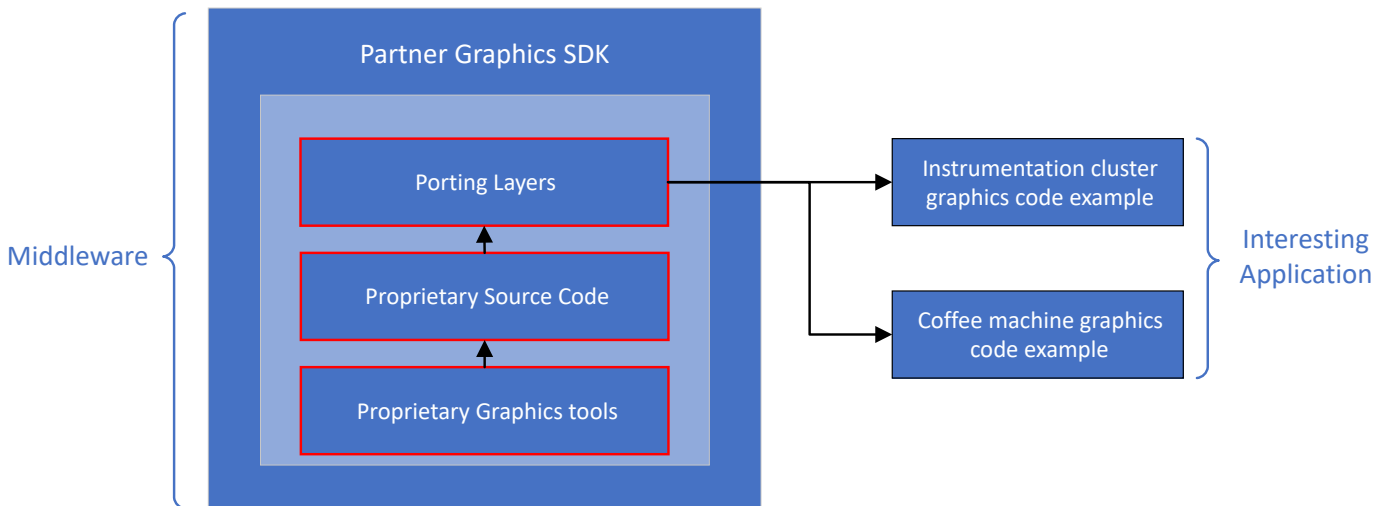
Scenario #3 - As a partner offering shields or sensors, the source code for interfacing with the shield or sensor should be provided as a middleware library. For example, a partner manufacturing a pressure sensor will develop the middleware library that allows users to interact with their sensor module easily using a simple API. This allow users to easily interface with any Infineon kit and obtain pressure data.
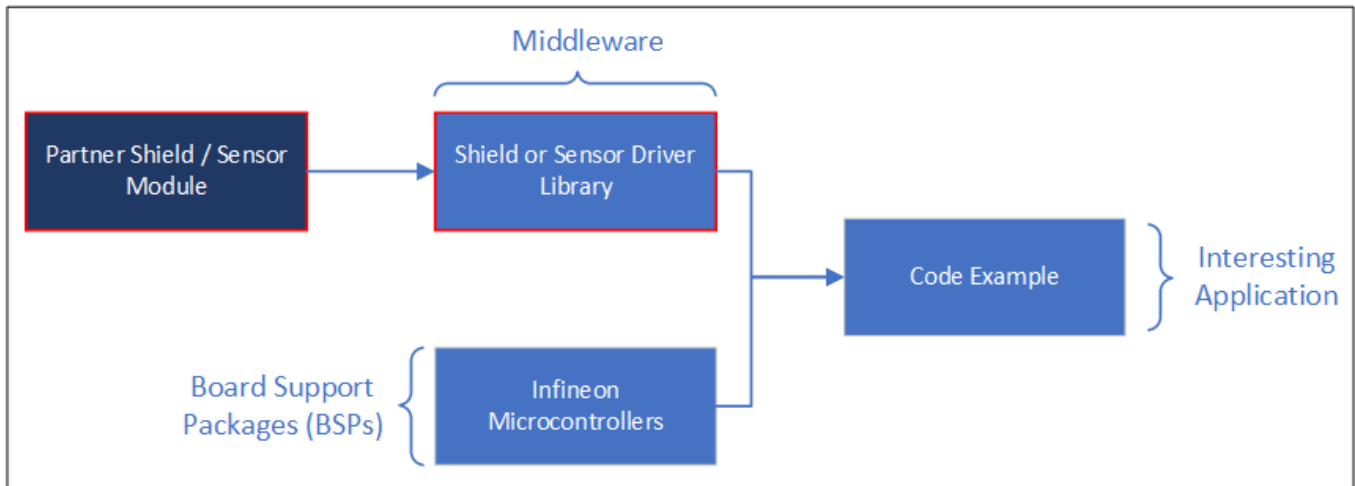
Reference examples that fall under this use case:

- **CY8CKIT-028-EPD** – This middleware library from Infineon provides the pin mapping and APIs to easily interface with the E-ink display shield board.
- **BMI160 Sensor Driver** – This middleware library from Bosch provides APIs to interface with the BMI160 inertial measurement unit (IMU) sensor.

Note:	*These use-cases are not an exhaustive list and any number of combinations between the different software types is possible and permitted. If you are unsure where your content belongs, you can contact technical support to seek help. See section* Technical Support *for more details.*

## 2.1.3	Board support packages (BSPs)

The Board Support Package (BSP) is intended to specify critical hardware items needed by 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

This section is intended for partners who offer custom modules or hardware based on Infineon chipsets. For partners focusing on only software solutions/services, you can leverage the existing BSPs offered by Infineon.

Let's look at a demonstrative scenario to understand when you would develop a BSP:

As a partner focused on creating modules or hardware using Infineon chipsets, a code example can be used to demonstrate how the custom module or evaluation kits (called custom BSPs) can be interfaced to create interesting applications. For example, a partner focused on creating IoT prototyping modules using Infineon chipsets can develop a BSP to simplify the development of applications targeting their modules or evaluation kits.
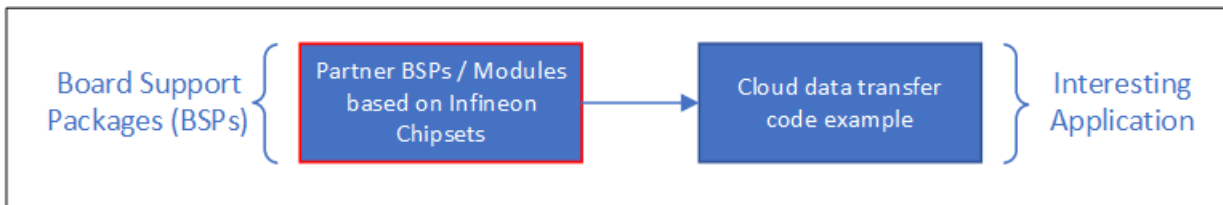
Note:	*These scenarios are not an exhaustive list and any number of combinations between the different software types is possible and permitted. If you are unsure where your content belongs, you can contact technical support to seek help. See section* Technical Support *for more details.*

## 2.2	Manifests

ModusToolbox™ uses the concept of manifests to load the content to be displayed in its tools (Project Creator and Library Manager). Manifests are XML formatted files that contain a list of URLs that point to the appropriate libraries. The tools discover the locations of these Git repositories by loading the manifests hosted from preconfigured locations in Infineon GitHub.

There are multiple types of manifest files:

- The "super manifest" file contains a list of URLs that software uses to find the board, code example, and middleware manifest files.
- An "app manifest" file contains a list of code examples that should be made available to the user.

- A "board manifest" file contains a list of 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.
- A "middleware manifest" file contains a list of available middleware libraries. There is also a separate middleware dependencies manifest that lists the dependent libraries associated with each middleware library.

The super manifest can list any number of app, board, and middleware manifest files. Each entry in an app, board, or middleware manifest file can contain more than one version of each library if desired. This allows new versions to be released while still allowing existing users to use any version they desire for their application needs.

Beginning with the ModusToolbox™ 2.2 release, there are two versions of manifest files: the old versions for the 'LIB' flow used with ModusToolbox™ 2.1 and earlier, and new versions for the 'MTB' flow (aka "fv2"), which is used with ModusToolbox™ 2.2 and later. In this application note, we will only cover the new "fv2" version of the manifests and content developed using ModusToolbox™ 2.2 and later.
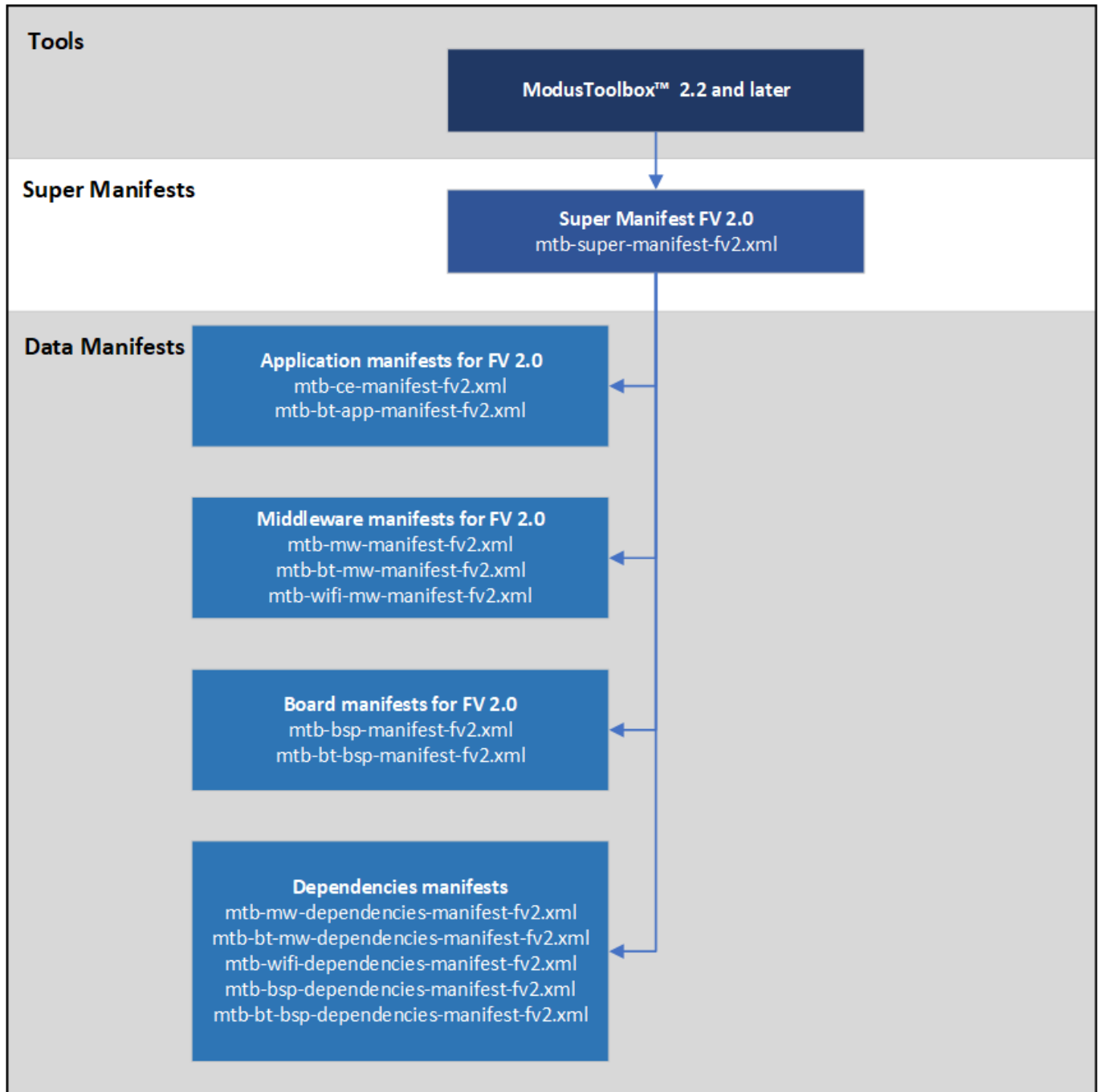
Partners can set up a similar infrastructure of manifests to point to content and showcase it within ModusToolbox™ tools (Project Creator and Library Manager). See upcoming sections to learn more about how you can set up such an infrastructure.

2.3 Git versioning control system

The Git versioning control system is central to how software content in ModusToolbox™ is offered. All the software content is hosted on Git repositories (Infineon GitHub and third-party Git servers). This allows content to be released regularly without requiring any update to the ModusToolbox™ tools.

As discussed previously, the manifests point to these Git repositories to allow discovery of the content from the tools. The tools run Git commands in the background to fetch the content from Git into your development environment. This is why it is important to host content on version control platforms based on Git so that the tools can work with them seamlessly.

The software content contains release tags that allow a specific version of the library to be brought in to the development environment. Because the software content is version-controlled, if any issues occur in a newly released version, it can be easily mitigated by rolling back to the last working version.

2.4   How does everything come together in ModusToolbox™?

ModusToolbox™ tools (Project Creator and Library Manager) point to a pre-configured super manifest location on GitHub. The super manifest points to the code example, middleware, and BSP manifests as explained earlier. This allows the tools to discover all the software content hosted on GitHub.
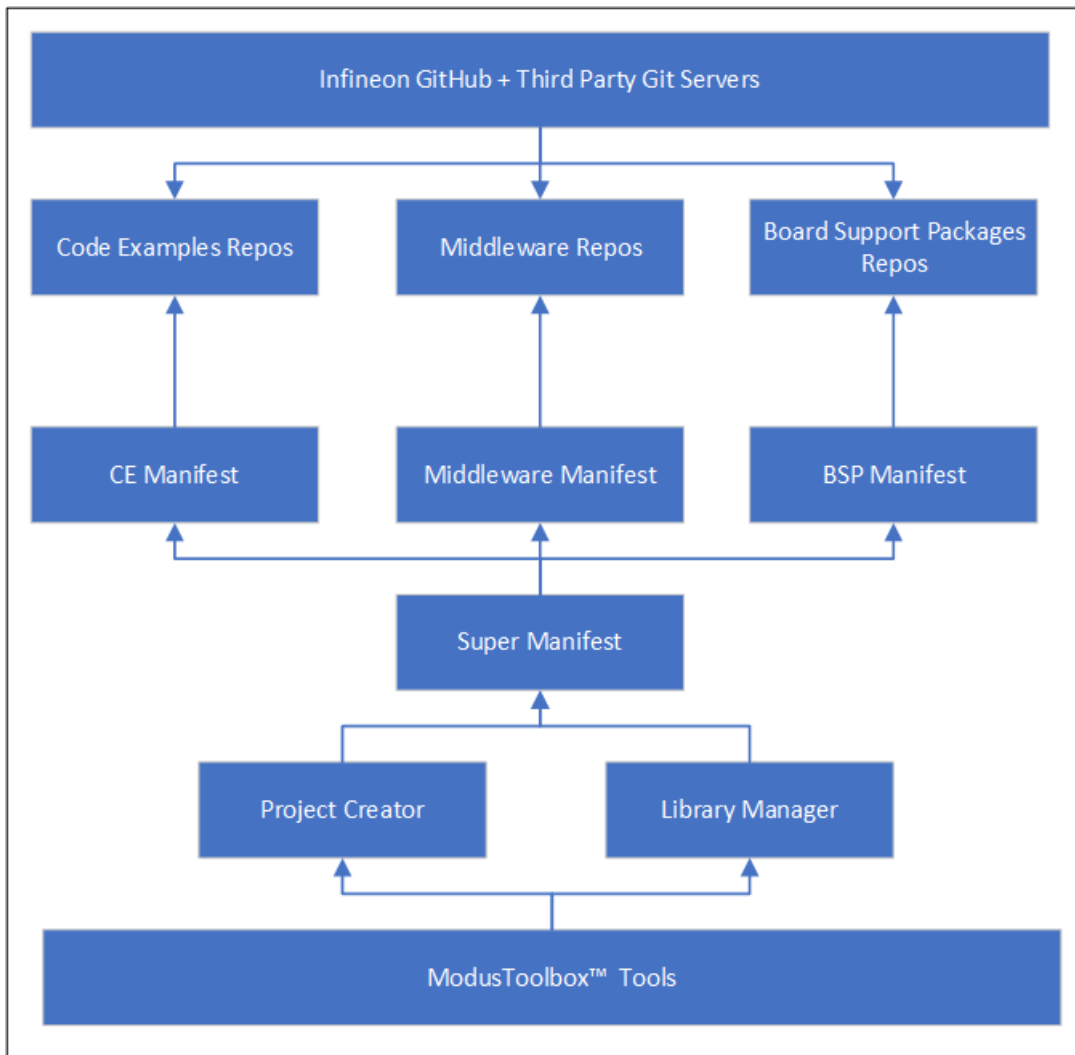
2.5   How does partner integration work?

The partner software content is delivered in a very similar fashion as the content offered by Infineon. As long as the content is hosted on Git servers and conforms to the structure ModusToolbox™ expects, all the content can be brought into ModusToolbox™ tools with the same concepts as shown below:

2 ModusToolbox™ software overview
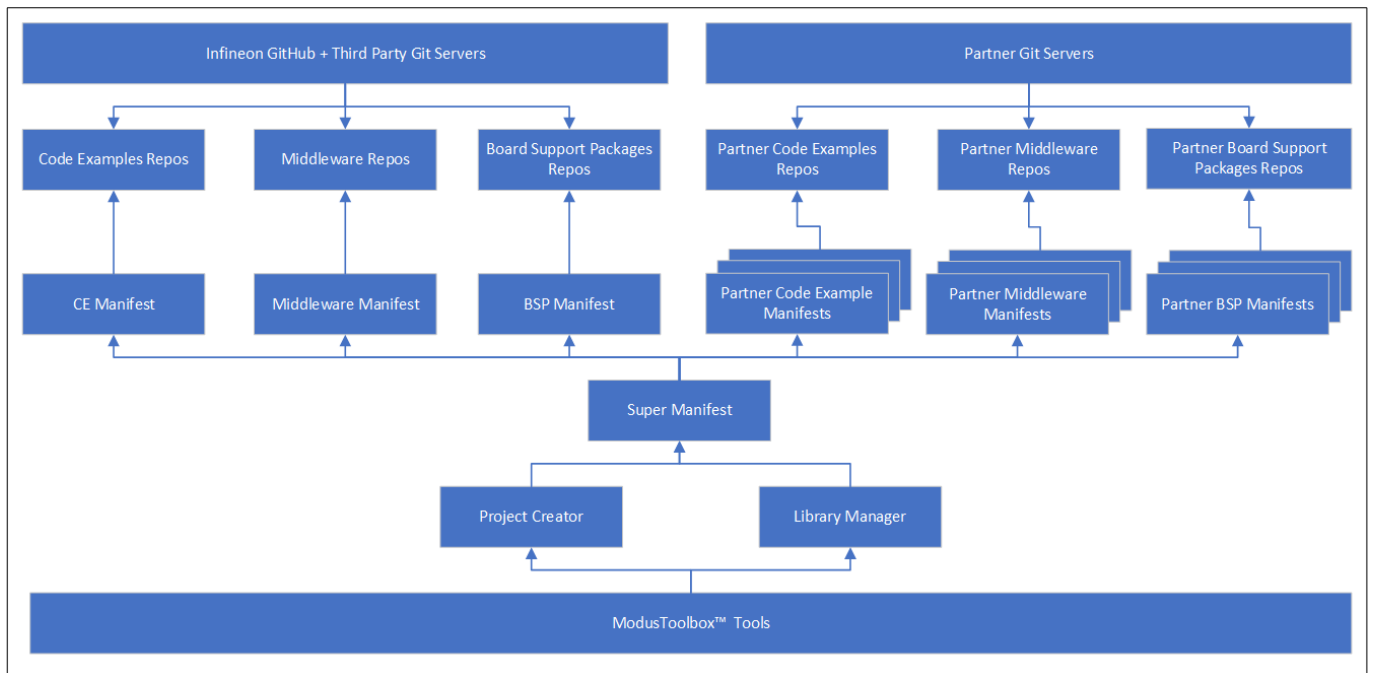
The super-manifest points to all the partner manifests in addition to Infineon's own manifests so that the partner software content can be easily integrated and discovered in the tool. This allows partners to manage their own content and deployment without any intervention from Infineon once your manifests are included in the super manifest.

3 Setting up your own Git infrastructure

As we learned in the previous sections, all the software content needs to be hosted on a platform that supports Git-based version control. Other version control systems like Mercurial, Perforce, and Subversion are not supported.

There are several platforms that provide Git hosting solutions today such as GitHub, GitLab, and Bitbucket. ModusToolbox™ can support any hosting platform as long it supports Git version control.

Let's look at the steps to set up your own Git infrastructure.

3.1 Choosing your Git hosting platform

There is no specific recommendation when it comes to choosing a Git hosting platform. ModusToolbox™ is not optimized to work better with any specific Git hosting platform. The tools only use the underlying Git version control system to bring in the necessary software content.

You should make sure that the platform you choose is accessible in all geographic regions in which you wish to supply your content.

3.2 Choosing your development flow

A development flow is a process used to develop, test, and deploy your software content in a systematic, sequential manner that eliminates errors and meets a certain quality standard. The upcoming sections describe two development workflows that can be used, but assume that you are familiar with Git and Git hosting service platforms to implement such a workflow. How to use Git and Git hosting service platforms are out of scope of this application note. The two most common development flows that can be adopted are as follows:

3.2.1 Single-stage workflow

A single-stage workflow is when you have a single Git hosting server where the visibility of the repositories is kept private until the development and testing is complete. This is slightly riskier considering that any human error could result in an incorrect or unstable version of the software being released to the public. Certain checks can be put in place to ensure these errors are caught before release.
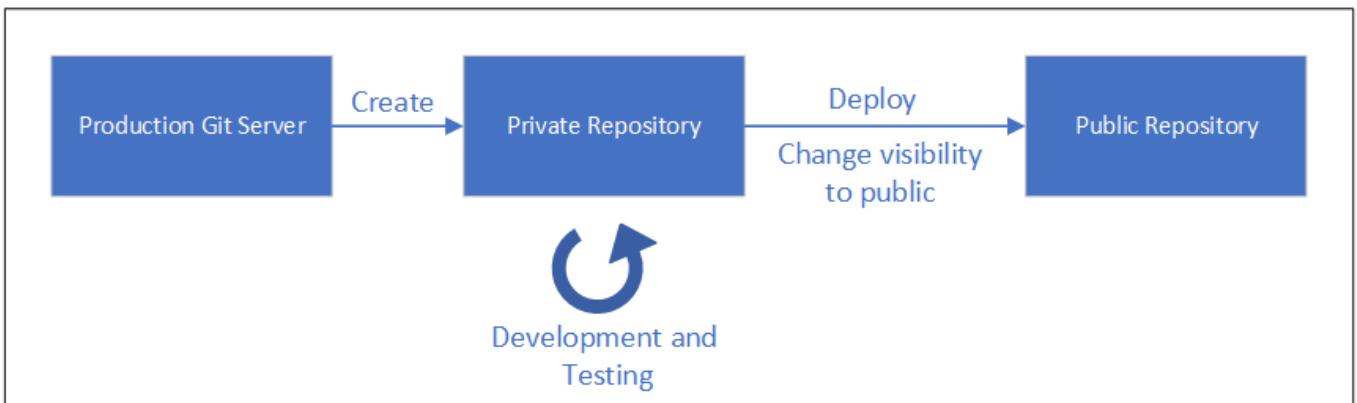
Note: *ModusToolbox™ tools does not work directly with private repositories and requires the setup of tokens as explained in the "Access to Private Repositories" section in the* ModusToolbox™ user guide *to get access to them. This extra step of setting up the tokens is just a one-time effort.*

3.2.2 Dual-stage workflow

A dual-stage setup consists of separate staging and production servers with the following characteristics:

- **Staging** – Internal Git hosting that is private and where the development and testing happens. Any content here is considered functional, reviewed, and pending final approval for release.
- **Production** – External facing Git hosting platform that is public and hosts only fully tested software and released software content.
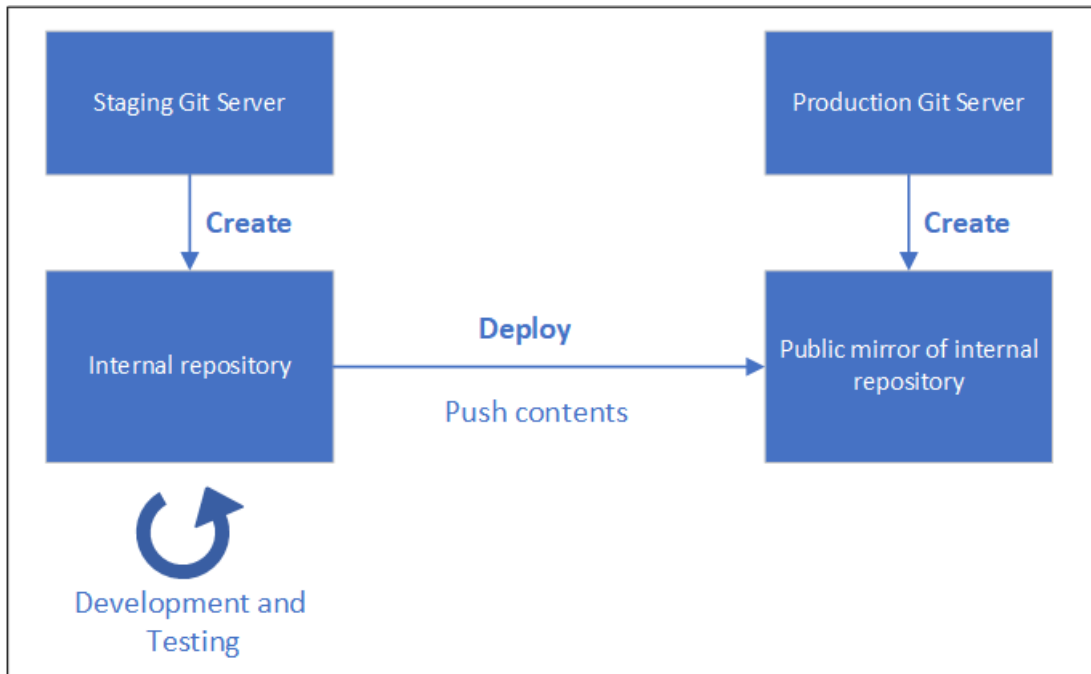
This setup allows developers to create software content in the staging environment without worrying about any mishaps or errors that could be exposed to the customer because it is internal to the company. Development and testing can be done iteratively in staging. Because the staging environment simulates a near-production-level environment, any issues can be caught before they reach end users. The software is deployed into production only after the content is error-free and completely tested.

The staging environment uses continuous integration and continuous delivery (CI/CD) pipelines to automate the process of testing the software content. The actions are performed on each commit to the repo, or after events like merging a topic branch into the main branch. If GitLab is used for the staging server, the actions performed are determined using the `.gitlab-ci.yml` configuration file, which is part of the development repo.

The software content is pushed to production only once all tests have passed. You get to decide how rigorous the testing should be to ensure the quality of the software content offered. See CI/CD documentation for more information on setting up such automated test pipelines.

Infineon uses the dual-stage development flow for creating its ModusToolbox™ software content.
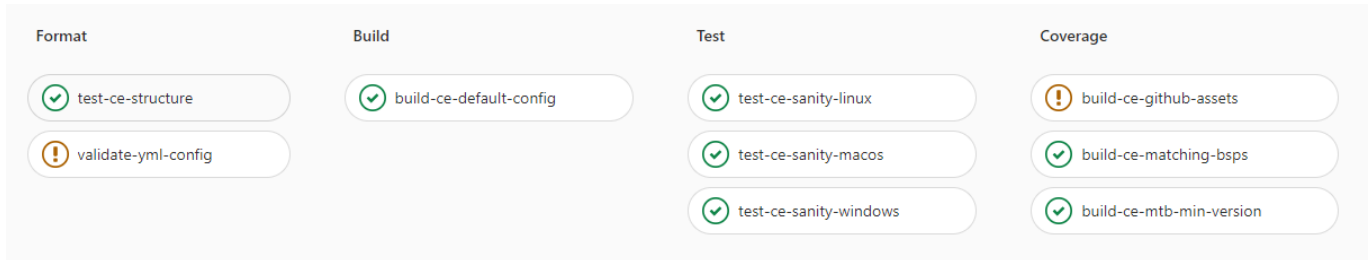
3.3 Designing your internal staging setup

The internal staging setup can implement CI/CD pipelines which are a set of automated jobs that perform particular tests based on some rules. These pipelines allow the software to be fully tested whenever there is any change to the repo. It is left to the partners to decide which jobs are necessary in the pipeline and which jobs are optional. Some examples of the types of testing jobs that can be implemented are as follows:

- **Documentation test** – This test validates the documentation. It goes through documentation such as markdown files to check if a particular template is being followed and if all the required sections exist. Additionally, all the links in the document can be tested to find any broken ones.

- **Software test** – This test validates the working of the code against all the constraints such as BSPs, ModusToolbox™ tools versions, toolchains, cross-platform support (Windows, Linux, macOS), and build configurations, and provides information on any errors or warnings that the code may have.
- **Code coverage test** – This test allows the code to be tested for any poor coding practices and conditions that can be optimized.

3.4 Designing your external production setup

The external production setup can implement the same CI/CD pipelines (explained in the previous section) to test the software content in a dual-stage setup but it is redundant because the staging environment mirrors the production one and has fully tested the content. However, in a single-stage setup, these pipelines might be necessary to help test the code before the repository goes public.

4 Creating your own software content

This section explains the procedure and best practices for going about the creation of software content. It is recommended to follow the steps explained in the upcoming sections so that the content developed works seamlessly with ModusToolbox™. All content should be developed using the latest version of ModusToolbox™ for best results but the steps are applicable for all content developed using ModusToolbox™ 2.2 and later.

Note: *This section assumes that you are familiar with creating applications in ModusToolbox™ and now want to create your own content. See the ModusToolbox™ user guide if this is not the case. Additional training material can be found here.*

4.1 Creating a code example

All current ModusToolbox™ code examples can be found through the GitHub code example page. There you will find links to examples for the Bluetooth® SDK, PSoC™ 6 MCU, and PSoC™ 4 device among others. You can also create any of the released code examples using the ModusToolbox™ Project Creator tool. Let's look at the steps to create your own code example.

1.   Choosing a starter application
2.   Choosing the name
3.   Choosing the title
4.   Adding the source files
5.   Adding middleware
6.   Adding the End User License Agreement (EULA)
7.   Create a Git repository
8.   Create a topic branch
9.   Testing the code example
10. Merging into mainline
11. Creating the release package

4.1.1 Choosing a starter application

Based on the application you want to develop and the device you want to target, open Project Creator and choose a BSP and code example that provides a good starting point. You can always use the empty application if you do not know where to begin. Before you click Create, choose an appropriate name for the code example by following the instructions provided in section Choosing the name.

4.1.2 Choosing the name

The name of the code example should be unique and should conform to the following best practices:

•   Should be short (avoid long file paths)

•   All lower-case, with words separated with hyphens '-'

•   Prefixed with `<partner>-mtb-example-`

Some examples include `mycompany-mtb-example-psoc6-object-detection-using-ml` or `mycompany-mtb-example-psoc6-face-recognition`.

Add this name in the text-box under "New Application Name" as illustrated below and click **Create**.
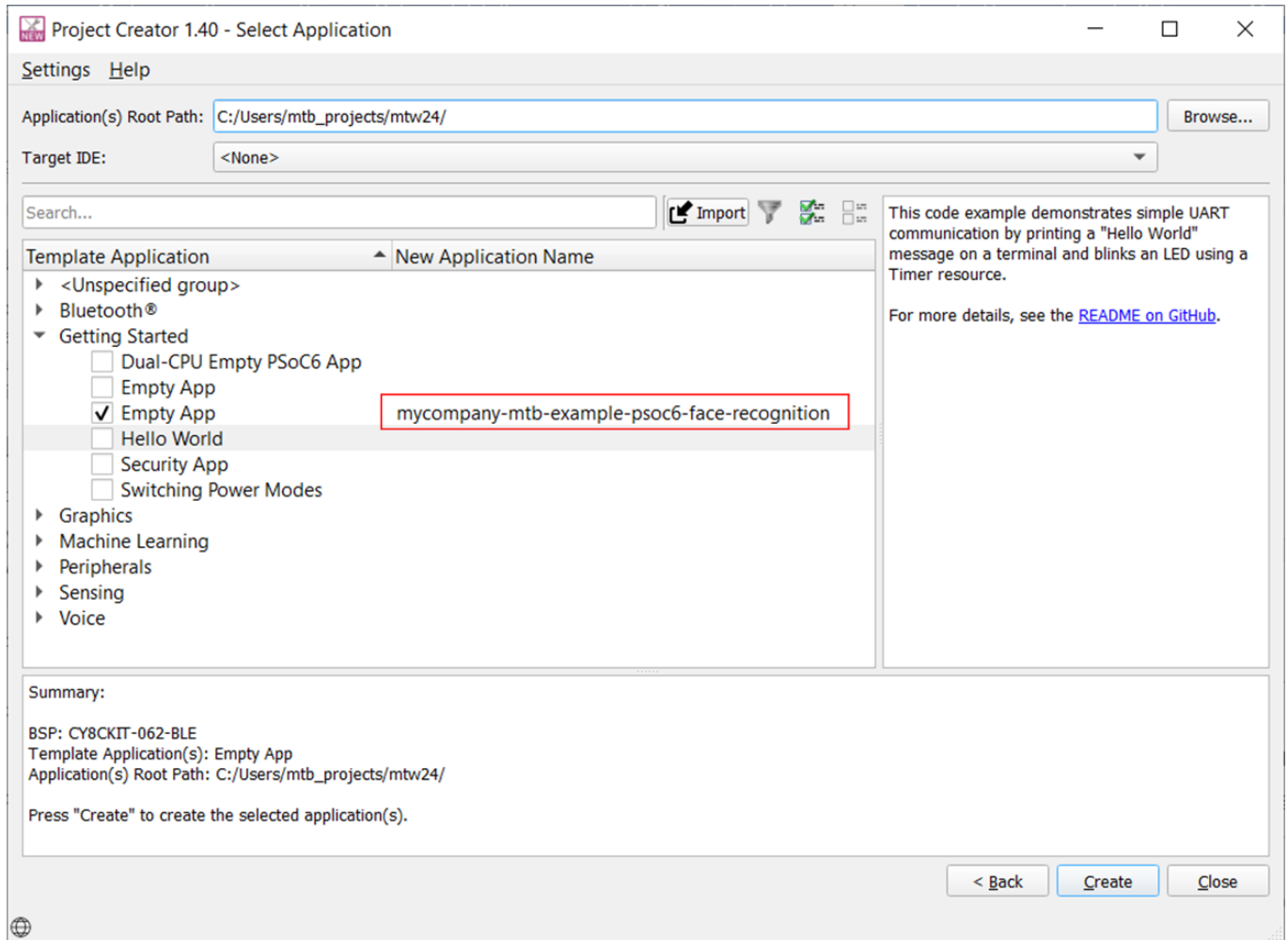
Once the code example generation is complete, update the Makefile to set the APPNAME make variable to the name of the code example. For example,

```
APPNAME=mycompany-mtb-example-psoc6-face-recognition
```

4.1.3   Choosing the title

The code example title should be a short descriptive header text and should conform to the following best practices:

- Informs the user of the scope and content of the code example
- It can contain the device family or platform name
- It does not use trademark symbols, superscripts, or subscripts

Some examples include "MyCompany PSoC 6 Face Recognition with ML" or "MyCompany PSoC 6 Object Detection" or "PSoC 6 Face Recognition with ML using MyCompany".

The `Readme.md` file in the code example should be updated with this title.

4.1.4   Adding the source files

The Project Creator tool is responsible for creating the fully functional application. During project creation, it looks at the deps folder to bring in all the dependent libraries to create the fully functional application. Additionally, based on the IDE of choice, other IDE-specific folders/files may be created.

All ModusToolbox™ code examples will contain the following folder/files:

- `deps` – Contains all the dependencies needed for the code example to run
- `Makefile` – Contains all the configuration variables to control the build flow
- `main.c` – Contains the source code

Add your source files that will interact with Infineon's device driver libraries or other middleware libraries. Here are some of the best practices when developing your source code:

- Use hardware abstraction layer (HAL) APIs if available for the hardware to allow the code example to be portable across multiple device families.
- Use the pin-aliased names available in the BSPs rather than actual pin numbers.
- Use the abstraction- rtos library if possible to make the code modular and reusable by users.
- Update the Readme.md file to describe the scope, purpose, working, design, and implementation of the code example. See available code example Readme.md files for reference.
- Follow other general coding practices like documentation of functions and meaningful variable naming.

If your source files need to interact with your proprietary software, we will cover how to add your proprietary software as a middleware in the next section.

4.1.5 Adding middleware

This section assumes that you have already created your middleware by following the steps in the Creating a middleware library section. The `deps` folder in an application contains a number of `.mtb` files. 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#latestv2.X#$$ASSET_REPO$$/capsense/latest-
v2.X
```

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

Adding your middleware to the code example is fairly simple:

1. Create a `.mtb` file inside the `deps` folder with the following value:

```
http://**github**.com/**<partner_name>**/**<my_middleware>**#**<commit>**#$$ASSET_REPO$$/**<my_middleware>**/
**<commit>**
```

2. Edit the details highlighted in bold to specify the path to your middleware on your Git hosting platform and also the corresponding commit or release tag to be used. Save the file and close it.
3. Run Library Manager and click **Update**. This runs `make getlibs` in the background which sees the new `.mtb` file and clones your middleware into the `mtb_shared` folder. You can also use the command line and run `make getlibs` for the same purpose.
4. Use the make variables in the application Makefile to specify the source files, header files, and paths from the middleware to be included by the build system. This is typically the path to the porting layer (see Scenario #2 in Middleware) in the middleware that supports Infineon chipsets. See the ModusToolbox™ user guide for more information on available make targets.

You have successfully added support for your middleware in your code example.

Note: *Make sure to copy these changes to the local copy of the repository and push these changes to the upstream Git server.*

4.1.6   Adding the End User License Agreement (EULA)

Once the code development is complete, add the EULA provided by Infineon as part of the partner agreement into your source directory. The source directory is the one that contains the application Makefile.

4.1.7   Create a Git repository

Create a repository with the code example name in your Git hosting platform:

*   This will be the staging server for the dual-stage workflow.
*   This will be the production server with the repository set to Private for the single-stage workflow.

Clone the repository using the git clone command. A local copy of this repository will now be available.

4.1.8   Create a topic branch

It is up to the partner to create a separate topic branch or use the main branch to work on the changes. It is recommended to use a topic branch to prevent any accidental updates to the main branch that might break the code for everyone. Use the `git checkout -b <branch_name>` command to create a topic branch.

As a rule of thumb, copy the project files and add it to your local copy of your Git repository you created previously whenever you make significant progress. Push all these changes to the upstream Git server so that you can always track your work and revert to a previously working configuration if needed. See the gitignore file for reference to understand what files should be pushed and what should be ignored.

4.1.9   Testing the code example

The code example must be tested to ensure that there are no issues with the build. It must also be tested on the hardware to verify the functionality. Once both the tests have passed, push the code example to the Git repository for automated test CI/CD pipelines to test it against multiple toolchains, tools versions, build configurations, cross-platform support, etc.

4.1.10   Merging into mainline

Once the code passes all the test pipelines, merge the topic branch into the main branch. Any merge conflicts that may arise should be resolved. If using a dual-stage setup, deploy the contents into production.

4.1.11   Creating the release package

Create a release tag from the main branch with the following naming scheme:

```
release-<major_version>-<minor_version>-<patch_version>
```

For example, the first release version will be `release-v1.0.0`. See release tags of an existing code example to understand what this looks like.

Here's how to decide the version numbers when creating a release package:

*   **major version:** Should be incremented only for code or makefile changes that break backward compatibility. This happens when the code no longer works with previous major versions of the tool or ecosystem. Reset the minor and patch version numbers to '0' when this is incremented.

- **minor version:** Should be incremented for code-related changes like changes to source code, file structure or makefile, that DO NOT break backward compatibility. Reset the patch version number to '0' when this is incremented.
- **patch version:** Should be incremented for document or code comment changes only.

An example of the version history for a code example that underwent different types of changes:

Scenario	Category of change	Version number
New content	New	**1.0.0**
Documentation fix, no changes to source code	Increment patch number	**1.0.**1
Cosmetic changes to source code, doesn't affect functionality	Increment patch number	**1.0.**2
Files / folders restructured	Increment minor number	**1.**1.**0**
Source code or Makefile changes that do not break backward compatibility	Increment minor number	**1.**2.**0**
Source code or Makefile changes that break backward compatibility	Increment major number	2.**0.0**

In addition to release tags, latest version tags need to be generated whenever there is a change in the major version number. They follow the naming scheme: `latest-v<major_version>.X`

The latest version tags are used to point to latest versions of the major version of the content available. This helps users to get the latest version of the content. For example, if there are two release tags for a code example such as `release-v1.0.0` and `release-v1.1.0`, the `latest-v1.x` tag will be generated to point to the latest release of the major version library, i.e., `release-v1.1.0`. See the tags generated here for reference.

Here's how you can create a latest version tag:

1. Create a latest version tag from the main branch with the naming scheme specified above and filling in the major version available.
2. Whenever there is an update to the content, the latest version tag must be moved to point to the latest version. For example, during initial deployment of content, `latest-v1.x` will point to the same commit as `release-v1.0.0` tag. Whenever the content is updated to `release-v1.1.0`, the `latest-v1.x` should be updated to point to the same commit as `release-v1.1.0` tag.

4.2 Creating a middleware library

There is no generic way to define what a middleware library should look like. It could be a firmware SDK, a binary or `.a` static library that users can link in their project. In any case, there are certain general guidelines to be followed.

To create a middleware library that is supported by ModusToolbox™, the following should be considered:

- It should follow any of the naming schemes listed below if the middleware library is developed specifically to support ModusToolbox™:
  - `<partner_name>-middleware` (For example, `mycompany-middleware`)
  - `<partner_name>-middleware-<feature_name>` (For example, `mycompany-middleware-remote-debugging`)
  - `<partner_name>-middleware-<tool_name>` (For example, `mycompany-middleware-graphics-wizard`)
  - `<partner_name>-middleware-<application>` (For example, `mycompany-middleware-image-processing`)
- If the middleware library is not developed to support ModusToolbox™ specifically, the naming scheme is left to the discretion of the partners.
- It should be hosted as a Git repository on any Git hosting platform. The name of the Git repository should be the same as the name of the middleware library.

- It should have a valid release and latest tag to prevent applications that use it from breaking because the contents of the middleware might change.
- It should have appropriate documentation to explain the APIs and how to add and use the middleware in an application.
- It can be easily integrated with the make-based flow in ModusToolbox™.
- Constraints should be specified in the documentation such as capabilities of the hardware required, supported operating systems, toolchains, and ModusToolbox™ tools versions.

Once the middleware is developed, push the changes to the upstream Git server and create a release tag similar to the one described in section Creating the release package.

If the middleware has dependencies on other libraries (i.e., it needs other libraries for it to work), then see Adding middleware dependencies to understand how to specify these dependencies. For example, a partner that offers cloud-based services could require wifi-connection-manager and http-client libraries to establish a connection to the cloud. This can be specified as a dependency to their middleware to make sure the right dependent libraries are brought in whenever the middleware is used.

4.3 Creating a BSP

The BSPs have a specific folder structure, files, and Makefiles to allow ModusToolbox™ applications to build correctly. To create your own BSP, ModusToolbox™ 3.x provides a graphical tool called "BSP Assistant". To launch it in standalone mode, run the `make bsp-assistant` make command in your application or you can search for `bsp-assistant` in the search bar and launch it. The steps to create your own custom BSP is provided in the BSP Assistant User Guide.

On ModusToolbox™ 2.x versions, the custom BSP is created using the command line. It supports the `make bsp` make command that creates a custom BSP based on a particular MCU device and optional connectivity module. See the Creating a Custom BSP User Guide for the steps to create your own custom BSP.

Once your BSP is created, create a repository on your Git hosting platform with the following best practices:

- Repo name must be prefixed with `TARGET_`
- The BSP name should be all uppercase, with words separated by hyphens "-".
- Include documentation for the BSP with kit features, contents, default configuration, etc.

For example, `TARGET_MY-KIT-062S2-43012` or `TARGET_MY-SENSOR-SHIELD`.

The BSP contains the design.modus file inside the config folder in the newer generation BSPs (BSP Gen 4) and inside the `COMPONENT_BSP_DESIGN_MODUS` folder for older generation BSPs (BSP Gen3 and older). Use the Device Configurator to open this file and define all the pin aliases, peripherals, clocks, and any other default settings of the kit.

Once the BSP is developed, push the changes to the upstream Git server and create a release tag similar to the one described in the Creating the release package section.

5 Creating your own manifest

By default, ModusToolbox™ tools look for Infineon's manifest files maintained on Infineon's GitHub server. So, the initial list of BSPs, code examples, and middleware available to use are limited to our manifest files. Once your content is completely integrated into ModusToolbox™, your manifest files will be included in the default Infineon super manifest. However, during development, you can create your own manifest files on your servers or locally on your machine, and you can override or add to where ModusToolbox™ tools look for manifest files. To do that, you first need to create manifest files for your BSPs, code examples, and middleware. You will then create a super manifest that points to these manifest files.

Let's go through the steps to create these manifests in the following sections.

5.1 Creating repositories

Create repositories on your Git hosting platform for each manifest. Each manifest file should reside in its own individual repository to allow the developer to update, maintain, and revert the manifests over time. That is, you should have one repository for a BSP manifest, one for a code example manifest, and one for a middleware manifest. You only need a repository for the types of content that you will be creating. For example, if you are not creating any BSPs, you don't need a BSP manifest.

The repositories need to use the following naming scheme based on the manifest type:

- Super manifest: `mtb-<partner_name>-super-manifest`
- App manifest: `mtb-<partner_name>-ce-manifest`
- Middleware manifest: `mtb-<partner_name>-mw-manifest`
- BSP manifest: `mtb-<partner_name>-bsp-manifest`

For example, the super manifest repository can be named `mtb-mycompany-super-manifest`.

The following repositories can be used for reference on how to create them and understand what goes in them:

- Super manifest: mtb -partner-super-manifest
- App manifest: mtb-partner-ce-manifest
- Middleware manifest: mtb-partner-mw-manifest
- BSP manifest: mtb-partner-bsp-manifest

Note: ModusToolbox™ tools does not work directly with private repositories and requires the setup of tokens as explained in the "Access to Private Repositories" section in the ModusToolbox™ user guide to get access to them. This extra step of setting up the tokens is just a one-time effort.

5.2 Creating your code example manifest

The code example manifest is used to point to URIs of code examples. Inside the `mtb-partner-ce-manifest` repository, create a file named `mtb-partner-ce-manifest-fv2.xml`.

Open the `mtb-partner-ce-manifest-fv2.xml` in an editor of your choice. The code example manifest is essentially an XML with the following base structure:

5 Creating your own manifest

Code Listing 1

```
<apps version="2.0">
  <app keywords="for keywords">
    <name>code example name</name>
    <category>code example category</category>
    <id>code-example-id-without-spaces</id>
    <uri>code example URL</uri>
    <description>Hi I am a code example</description>
    <req_capabilities>code example capabilities</req_capabilities>
      <versions>
        <version flow_version="2.0" tools_min_version="MTB tools minimum version"
req_capabilities_per_version="bsp_gen4">
          <num>Name of the branch</num>
          <commit>exact branch of the repo to be used</commit>
        </version>
      </versions>
  </app>
</apps>
```

The file starts with `<apps>...</apps>` as the root labels. Each code example is described within an `<app></app>` section. Multiple code examples can thus be added using the `<app>` labels within the root `<apps>` section.

Code Listing 2

```
<apps version="2.0">
  <app keywords="psoc6,partner,demo">
    // body of the code example 1
  </app>
  <app keywords="psoc6">
    // body of the code example 2
  </app>
</apps>
```

Update the body of the code example using the guidance for the fields described below to add details of your CE:

Field / attribute	Description	Example
app:keywords	List of labels which helps identify the given code example using the Project Creator search feature. ✔ Multiple keywords are supported as a comma-delimited list. ✔ Allowed: chars, nums, spaces, hyphen, underscore, period ✔ Best practice: include keywords used in `CE title`, `req_capabilities` attribute.	<app keywords="psoc6,led,starter,hello world,mtb-flow">

5 Creating your own manifest

Field / attribute	Description	Example

\<name\>	Name of the CE as it would show up in the Project Creator.
	
	✔ Use a short descriptive text. The CE title is a good starting point.
	
	   Avoid including device family or SDK name (example: "PSoC 6") as part of this field.
	
	   Do not use special characters like hyphens, colons, underscores, bracket or slashes in this field. Spaces are allowed.
		\<name\>Hello World\</name\>

\<category\>	Value which enables showing CEs in categorized (grouped) list. Check existing categories to determine where the CE fits best.	\<category\>Voice\</category\>
\<id\>	Unique ID of the CE.	\<id\>mtb-example-psoc6-hello-world\</id\>
\<uri\>	Path to the CE's GitHub repo. No trailing or leading spaces.	\<uri\>https://github.com/Infineon/mtb-example-psoc6-hello-world\</uri\>
\<description\>	Short description of the CE. This text shows up in the description area of the Project Creator while selecting the CE. Do not use special characters like ™ or ® or smart (curly) braces. ✔ This can be used to include a link to company website and sales.	\<description\>This code example demonstrates the implementation of simple UART communication and blinks an LED using a Timer resource using PSoC 6 MCU.\</description\>
\<req_capabilities\>	Capabilities required by the CE which must be provided by the board (BSP). CEs will be listed in the Project Creator only for those BSPs which provide these capabilities using the \<prov_capabilities\> field in the BSP manifest. Use this to ensure that the CE is listed only for supported kits. See Adding BSP capabilities for a list of available capabilities and their descriptions. ✔ Enter a space-separated list of capabilities the CE requires. All the required capabilities must be satisfied by a BSP for the CE to show up in Project Creator for that BSP.	psoc6

5 Creating your own manifest

Field / attribute	Description	Example
version:flow_version	Specifies flow version, required/ identified by tools > MTB 2.1 For the -fv2 manifest, this should be 2.0.	<version flow_version="2.0"
version:num version:commit	These indicate the aliases for the commit hash or branch/tag to fetch. CE should mandatorily contain a release tag i.e., release-v1.0.0 and a latest tag i.e., latest-v1.X. See section Creating the release package for more details. Although the default "main" branch can be used here, it might keep receiving updates and commits might move. It is always better to create a release tag to make the content static.	``` <versions> <version> <num>Latest v1.X release</num> <commit>latest-v1.X</commit> </version> <version> <num>1.0.0 release</num> <commit>release-v1.0.0</commit> </version> </versions> ```
version:tools_min_version version:tools_max_version	[Optional] Specifies min and max ModusToolbox™ tool/patch versions on which the particular <version> of application is displayed. tools_min_version should match the minimum required tools or patch version. For example, 2.2.0 if ModusToolbox™ 2.2 is required without any patches, or 2.2.1 if the 2.2.1 patch is required. All entries in fv2 manifest file MUST contain tools_min_version="2.2.0" or a higher version number. In general, leave tools_max_version blank as examples are expected to work with latest versions of tools. *Note:* *The ModusToolbox™ version in the Readme requirements section should match these values.*	<version tools_min_version="2.2.0" tools_max_version="2.4.1">

5 Creating your own manifest

Field / attribute	Description	Example
version:req_capabilities_per_version	[Optional] A list of required version-specific capabilities for the given application. This list is treated as an "AND" list in addition to the global `req_capabilities`. The list is whitespace-delimited. If this attribute is missing or empty, it means that this application has no version-specific capability.	`<version req_capabilities_per_version="bsp_gen4 std_crypto">`
app:req_capabilities_v2	[Optional] A list used to restrict the scope of the code example. See Specifying requirements for restricted scope for more information.	`req_capabilities_v2="[cy8ckit_062-wifi_bt,cy8ckit_062s2_43012]"`

An example of what the manifest file will look with all the details filled in is shown below:

Code Listing 3

```
<apps version="2.0">
  <app keywords="psoc6,partner,my,cool,demo">
    <name>My Cool Demo</name>
    <category>Voice</category>
    <id>partner-mtb-example-my-cool-demo</id>
    <uri>https://github.com/partner/partner-mtb-example-my-cool-demo</uri>
    <description>
      <![CDATA[This code example demonstrates this XYZ application PSoC™ 6 MCU with AIROC™
CYW43xxx Wi-Fi & Bluetooth® combo chips.
        <br><br>For more details, see the <a href="https://github.com/partner/partner-mtb-
example-my-cool-demo/blob/master/README.md">README on GitHub</a>.
        <br><br><a href="https://partner-webpage.com/">Partner</a> is a company that does ABC
and XYZ.
        <br> > <a href="https://docs.partner-doc-page.com/">Get Started</a><br> > <a
href="https://partner-webpage.com/contact/">Talk to Sales</a>]]>
    </description>
    <req_capabilities>psoc6</req_capabilities>
    <versions>
      <version flow_version="2.0" tools_min_version="3.0.0"
req_capabilities_per_version="bsp_gen4">
        <num>Latest 1.X release</num>
        <commit>latest-v1.X</commit>
      </version>
      <version flow_version="2.0" tools_min_version="3.0.0"
req_capabilities_per_version="bsp_gen4">
        <num>1.0.0 release</num>
        <commit>release-v1.0.0</commit>
      </version>
    </versions>
  </app>
</apps>
```

Note: *The latest tag is used to bring the latest version of the content available in the repository. This tag must be used to point to the latest version of the library. All the tags mentioned in the manifest must exist in the repository hosting the content. See* Creating the release package *to learn more about creating these tags if not already done.*

An illustration of how these fields affect the behavior in project-creator is shown below:
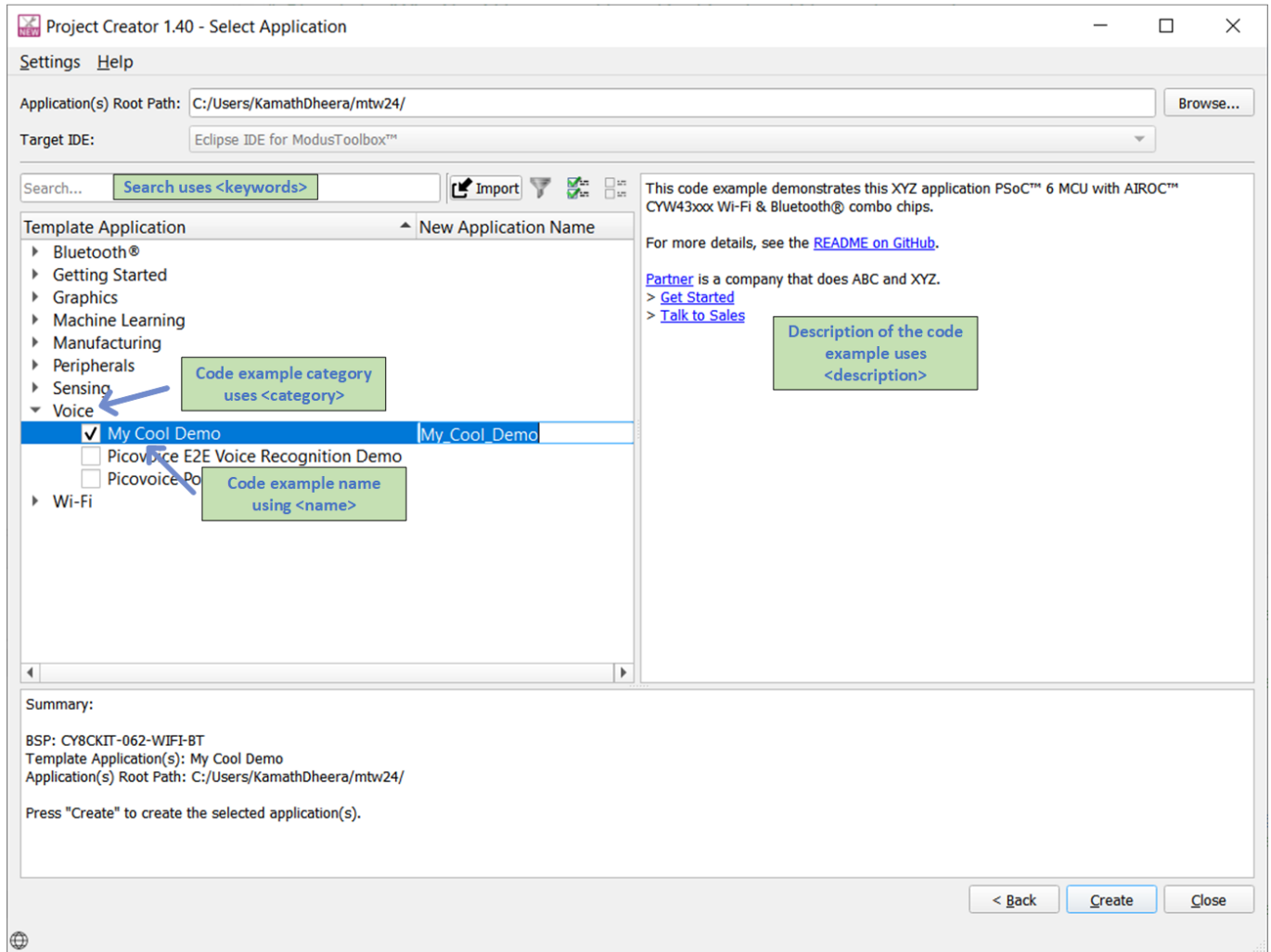
5.2.1   Adding BSP capabilities

ModusToolbox™ relies on a set of capabilities provided by the BSP and required by the code examples to determine what is compatible. This ensures that code examples and BSPs can be created and released completely independently of the software tools that present them to customers.

Capabilities	Description
psoc4	This board includes a PSoC™ 4 (e.g., PSoC™ 4200, PSoC™ 4100S Max, , …) MCU that can be used to develop firmware on.
psoc6	This board includes a PSoC™ 6 (e.g., PSoC™ 62, PSoC™ 63, …) MCU that can be used to develop firmware on.
pmg1	This board includes a EZ-PD™ PMG1 (e.g., CYPM1xxx, …) MCU that can be used to develop firmware on.

5 Creating your own manifest

Capabilities	Description
cat1	This board includes an MCU that is part of CAT1A (or PSoC™ 6) and CAT1B family of devices.
cat2	This board includes an MCU that is part of CAT2 (PSoC™ 4 and PMG1) family of devices.
cat3	This board includes an MCU based on the Infineon XMC™ platform (XMC1xxx, XMC4xxx).
cat4	This board includes an MCU based on the 43907 platform (43907, 54907).
xmc	This board includes an XMC™ MCU that can be used to develop firmware on.
xmc1000	This board includes an Arm® Cortex® M0 (CM0) XMC1xxx MCU that can be used to develop firmware on.
xmc4000	This board includes an Arm® Cortex® M4 (CM4) XMC4xxx MCU that can be used to develop firmware on.
xmc7000	This board includes a CM0+ and up to two an Arm® Cortex® M7 (CM7) XMC7xxx MCU that can be used to develop firmware on.
cyw20xxx	This board includes a CYW20xxx (e.g., CYW20735, CYW20820, ...) MCU that can be used to develop firmware on.
cyw20<###>	This board includes the specific connectivity chip (e.g., CYW4343W, CYW43012). This should only be used when the generic 'cyw20xxx' capability is also provided.
cyw43xxx	This board includes a CYW43xxx (e.g., CYW4343W, CYW43012, ...) chip providing Bluetooth® and Wi-Fi functionalities.
Chip capabilities	
adc	The MCU on the board contains at least one programmable analog-to-digital converter (ADC) block.
can	The MCU on the board contains at least one Controller Area Network (CAN) block.
comp	The MCU on the board contains at least one analog comparator block.
cyw20xxx_controller	The MCU on the board includes a CYW20xxx (e.g., CYW20819, CYW20706, ...) chip providing Bluetooth® controller functionality (standard HCI, WICED-HCI). A separate MCU is used for the host functionality.
dac	The MCU on the board contains at least one programmable digital-to-analog converter (DAC) block.
dma	The MCU on the board contains at least one programmable DMA block.
flash_<X>k	The MCU on the board contains X kilobytes of internal flash memory, e.g., for a device with 256 KB of flash, the capability would be: flash_256k.
i2c	The MCU on the board contains at least one programmable I2C block.
i2s	The MCU on the board contains at least one programmable I2S block.
lin	The MCU on the board contains at least one programmable Local Interconnect Network (LN) block.
low_power	The MCU and board are capable of running in low-power modes.

5 Creating your own manifest

Capabilities	Description
lptimer	The MCU on the board contains at least one programmable low-power timer (LPT) block.
mcu_gp	The MCU on the board has a programmable general-purpose MCU (e.g., PSoC™, CYW43907), NOT: 20xxx.
multi_core	The MCU on the board has multiple user-configurable cores (e.g., most PSoC™ 6 MCUs have a CM0+ and a CM4 core)
opamp	The MCU on the board contains at least one programmable operational amplifier block.
qspi	The MCU on the board contains at least one programmable Quad SPI block.
rtc	The MCU on the board contains at least one programmable real-time clock (RTC) block.
secure_boot	The MCU on the board is capable of booting into a secured environment.
smart_io	The board contains provisions to access smart I/O capable pins, even if the pins are multiplexed with other peripherals like CAPSENSE™.
std_crypto	The MCU on the board supports standard cryptography APIs.
uart	The MCU on the board contains at least one programmable UART block.
udb	The MCU on the board has a device that supports configurable Universal Digital Blocks (UDB).
sram_<X>k	The MCU on the board contains X kilobytes of internal SRAM, e.g., for a device with 16 KB of SRAM, the capability would be: sram_16k.
ble	This chip and board are capable of performing Bluetooth® LE communication. DEPRECATED: With the AIROC™ stack unification, going forward the "bt" capability covers all devices with Bluetooth® radios.
bt	This chip and board are capable of performing full Bluetooth® communication.
capsense	This chip supports CAPSENSE™ and the board contains CMOD and/or CINT capacitors to support CAPSENSE™ design. It may not have any widget present though.
capsense_button	The chip supports CAPSENSE™ and the board contains at least one CAPSENSE™ button.
capsense_linear_slider	The chip supports CAPSENSE™ and the board contains at least one CAPSENSE™ linear slider.
capsense_radial_slider	The chip supports CAPSENSE™ and the board contains at least one CAPSENSE™ radial slider.
capsense_touchpad	The chip supports CAPSENSE™ and the board contains at least one CAPSENSE™ touchpad/trackpad.
sdhc	This chip supports SDHC communication and the board contains an SDHC-compatible card port.
usb_device	This chip supports USB device operation and the board contains a USB header.
usb_host	This chip supports USB host operation and the board contains a USB header.
usbpd	This chip supports USB Power Delivery and the board contains a USB header.

5 Creating your own manifest

Capabilities	Description
wifi	This chip and board are capable of performing full Wi-Fi communication.
Board elements	
arduino	This board provides a pinout compatible with Arduino.
enclosure	This board is sealed in an enclosure to showcase liquid tolerance by immersing in liquids.
fram	This board contains at least one memory device that is a F-RAM.
j2	This board contains the an extended J2 pin header beyond the regular Arduino pins (used by some shields). This is only expected to be used if 'arduino' capability is also provided.
led	This board contains at least one user-controllable LED.
memory	This board contains a memory device external to the main processor that can be accessed via SPI, QSPI, …
nor_flash	This board contains at least one memory device that is a NOR flash.
pot	This board contains an analog potentiometer that can be accessed via a GPIO.
rgb_led	This board contains at least one RGB LED.
serial_led	This board contains at least one RGB LED driven serially using SPI interface.
switch	This board contains at least one user-controllable switch (aka button).
Others	
bsp_gen1, bsp_gen2, bsp_gen3, bsp_gen4	Anytime a new major version of a BSP is created, you need to switch the bsp_gen[X] capability to a higher number. This ends up being used by Project Creator in the way the major version of the asset should be used. bsp_gen1 - CAT1A v1 bsp_gen2 - CAT1A v2, CAT2 v1, CAT3 v1 bsp_gen3 - CAT1A v3, CAT2 v2 bsp_gen4 - CAT1A v4, CAT1B v1, CAT1C v1, CAT2 v3, CAT3 v2

Note: *You don't necessarily have to add all the capabilities. At a basic level just Infineon chip-level capabilities like "psoc6" or "psoc4" can be used. Other constraints that the code example may have are typically documented in the Readme file.*

A code example manifest template has been created with details filled out. You can use this directly or modify the fields as required.

5.2.2   Specifying requirements for restricted scope

The code example manifest lists the capabilities it requires to work using options `<req_capabilities>` or `<req_capabilities_per_version>` described previously. ModusToolbox™ looks at BSPs that provide these capabilities (specified using `<prov_capabilities>` as described in Adding BSP dependencies) and displays the code examples against only the supported BSPs.

For those special cases when the code example works for specific BSP(s), it is not possible to use `<req_capabilities>` or `<req_capabilities_per_version>` because several BSPs could have the same capabilities. Hence, for a code example with restricted scope where it works only for specific BSPs or devices with a minimum flash size, the manifests provide an additional option `<req_capabilities_v2>` to specify such requirement.

5 Creating your own manifest

Use `req_capabilities_v2` as per guidance and example provided below:

- `req_capabilities_v2` uses the format "[tag1] [tag2, tag3]" which translates to the app requiring tag1 AND (tag2 OR tag3) Reminder: req_capabilities and `req_capabilities_per_version` use the format "tag4 tag5" which translates to the app requiring tag4 AND tag5.
- Overall required capabilities for the app for a particular version is an AND condition of individual lists: `req_capabilities_v2` AND `req_capabilities` AND `req_capabilities_per_version`.

Example 1: For displaying a code example that is supported only by two specific BSPs:

Code Listing 4

```
<apps version="2.0">
  <app keywords="psoc6,partner,my,cool,demo" req_capabilities_v2="[cy8ckit_062-
wifi_bt,cy8ckit_062s2_43012]">
    <name>My Cool Demo</name>
    <category>Voice</category>
    <id>partner-mtb-example-my-cool-demo</id>
    <uri>https://github.com/partner/partner-mtb-example-my-cool-demo</uri>
    <description><![CDATA[This code example demonstrates this XYZ application PSoC™ 6 MCU with
AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips.
      <br><br>For more details, see the <a href="https://github.com/partner/partner-mtb-example-
my-cool-demo/blob/master/README.md">README on GitHub</a>.
      <br><br><a href="https://partner-webpage.com/">Partner</a> is a company that does ABC and
XYZ.
      <br> > <a href="https://docs.partner-doc-page.com/">Get Started</a>
      <br> > <a href="https://partner-webpage.com/contact/">Talk to Sales</a>]]>
    </description>
    <req_capabilities>psoc6</req_capabilities>
    <versions>
      <version flow_version="2.0" tools_min_version="3.0.0"
req_capabilities_per_version="bsp_gen4">
        <num>Latest 1.X release</num>
        <commit>latest-v1.X</commit>
      </version>
      <version flow_version="2.0" tools_min_version="3.0.0"
req_capabilities_per_version="bsp_gen4">
        <num>1.0.0 release</num>
        <commit>release-v1.0.0</commit>
      </version>
    </versions>
  </app>
</apps>
```

Example 2: For displaying a code example that requires a minimum flash size

Code Listing 5

```
<apps version="2.0">
  <app keywords="psoc6,partner,my,cool,demo" req_capabilities_v2="[flash_2048k,flash_1024k]">
    <name>My Cool Demo</name>
    <category>Voice</category>
    <id>partner-mtb-example-my-cool-demo</id>
    <uri>https://github.com/partner/partner-mtb-example-my-cool-demo</uri>
    <description><![CDATA[This code example demonstrates this XYZ application PSoC™ 6 MCU with
AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips.
      <br><br>For more details, see the <a href="https://github.com/partner/partner-mtb-example-
my-cool-demo/blob/master/README.md">README on GitHub</a>.
      <br><br><a href="https://partner-webpage.com/">Partner</a> is a company that does ABC and
XYZ.<br> > <a href="https://docs.partner-doc-page.com/">Get Started</a>
      <br> > <a href="https://partner-webpage.com/contact/">Talk to Sales</a>]]<
    </description>
    <req_capabilities>psoc6</req_capabilities>
    <versions>
      <version flow_version="2.0" tools_min_version="3.0.0"
req_capabilities_per_version="bsp_gen4">
        <num>Latest 1.X release</num>
        <commit>latest-v1.X</commit>
      </version>
      <version flow_version="2.0" tools_min_version="3.0.0"
req_capabilities_per_version="bsp_gen4">
        <num>1.0.0 release</num>
        <commit>release-v1.0.0</commit>
      </version>
    </versions>
  </app>
</apps>
```

5.3　　　Creating your middleware manifest

The middleware manifest is used to point to URIs of middleware. Inside the `mtb-partner-mw-manifest` repository, create a file named mtb-partner-mw-manifest-fv2.xml.

The middleware manifest file has the following base structure:

5 Creating your own manifest

Code Listing 6

```
<middleware>
    <middleware>
        <name>Middleware Name</name>
        <id>middleware-id-without-spaces</id>
        <uri>Middleware URL</uri>
        <desc>Middleware Description</desc>
        <category>Middleware</category>
        <req_capabilities>capabilities required for the middleware to work</req_capabilities>
        <versions>
        <version flow_version="1.0,2.0" tools_min_version="3.0.0">
            <num>Version number</num>
            <commit>release tag of middleware</commit>
            <desc>description for release tag</desc>
        </version>
        </versions>
    </middleware>
</middleware>
```

The root of the XML contains the `<middleware>` … `</middleware>` section. Details of the middleware are written within another `<middleware>` section within the root `<middleware>` node as shown below:

Code Listing 7

```
<middleware>
    <middleware>
        // body of middleware 1
    </middleware>
    <middleware>
        // body of middleware 2
    </middleware>
</middleware>
```

Update the body of the middleware using the guidance for the fields described below to add details of your middleware:

| Element	Description	Example
<name>	A user-friendly name for the middleware. This is what is displayed in the UI. Typically, the name is the same as the GitHub repository name.	partner-middleware
<id>	A unique identifier for the middleware. The manifest processing code will give an error if multiple middleware items have the same ID. Typically, the name is the same as the GitHub repository name.	partner-middleware

5 Creating your own manifest

Element	Description	Example
<uri>	The URI for the Git repository holding the middleware.	https://github.com/partner/partner-middleware
<desc>	A user-friendly text description of the middleware item. This is meant to be displayed in the UI. Typically, this is one or two sentences that match the GitHub repository "About" text or README.md first paragraph. Note: Some asset descriptions apply advanced formatting like bold/inline text or line separators. In such situations, enclose the text into CDATA section.	<![CDATA[Block device drivers for use with littlefs filesystem. License Disclaimer: Adding this library will also download and add littlefs to your project. It is your responsibility to understand and accept the littlefs license.]]>
<category>	A user-friendly text string that specifies the category for displaying this middleware item in a GUI. It is expected that all middleware in the same category will be shown together in the library management GUI.	Middleware
<req_capabilities>	A list of capabilities that this middleware requires. This list is treated as an "AND" list. That is, all capabilities must be met. The list is whitespace-delimited; each item in the list must be a valid C identifier. If this element is missing or empty, it means that this middleware has no capability requirements. That is, it works with all boards. For the complete list of the provided capabilities that can be "required" by the middleware, see Adding BSP capabilities.	psoc6

All libraries and middleware are shown in the Library Manager tool. An illustration of how these fields affect the behavior in Library Manager is shown below:

5 Creating your own manifest
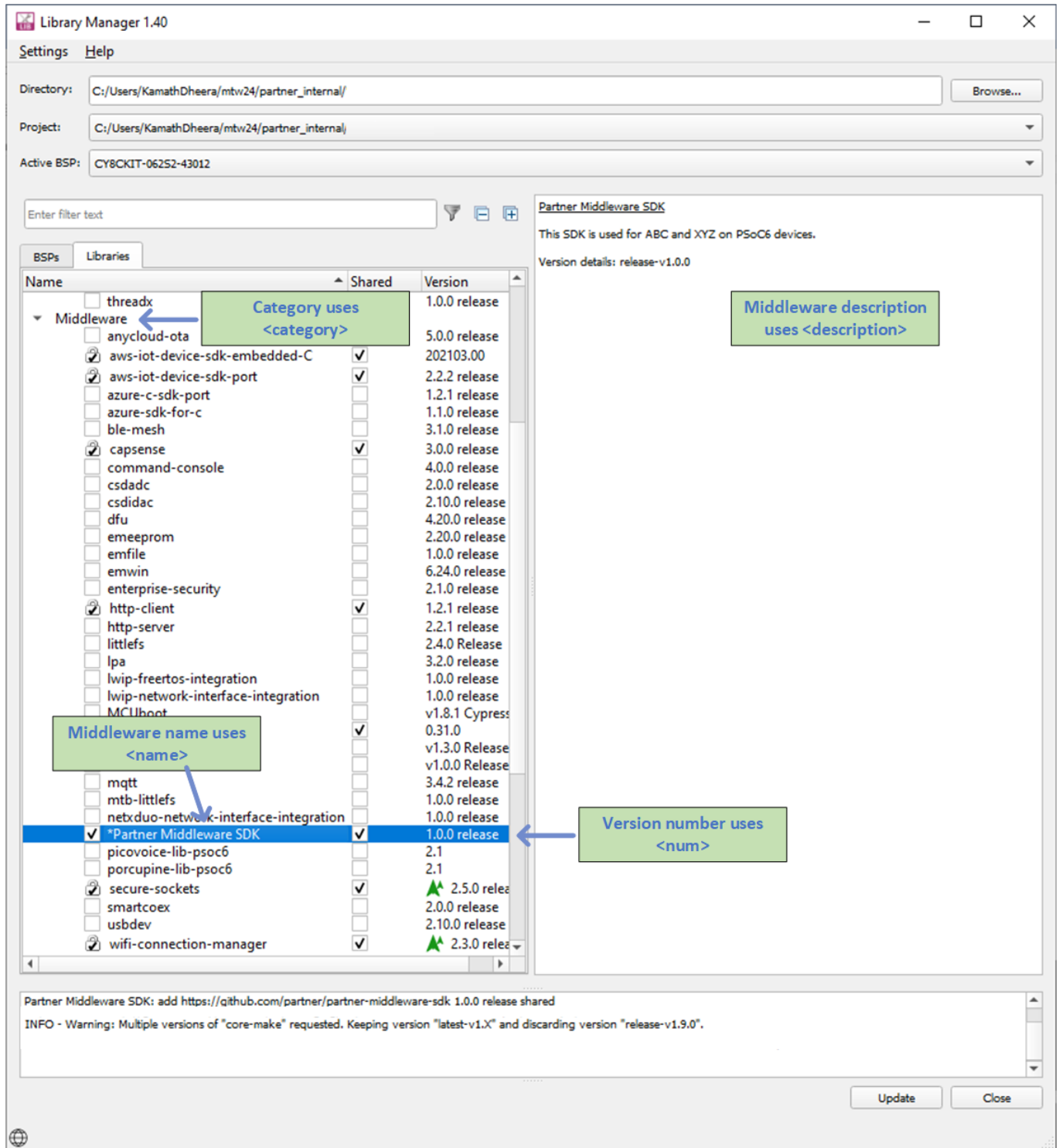
An example of what the middleware manifest file will look like with all the details filled is shown below:

Code Listing 8

```
<middleware>
    <middleware>
        <name>partner-middleware-sdk</name>
        <id>partner-middleware-sdk</id>
        <uri>https://github.com/partner/partner-middleware-sdk</uri>
        <desc>This SDK is used for ABC and XYZ on PSoC6 devices.</desc>
        <category>Middleware</category>
        <req_capabilities>cat1</req_capabilities>
        <versions>
         <version flow_version="1.0,2.0">
            <num>Latest v1.X release</num>
            <commit>latest-v1.X</commit>
            <desc>latest-v1.X</desc>
         </version>
         <version flow_version="1.0,2.0">
            <num>1.0.0 release</num>
            <commit>release-v1.0.0</commit>
            <desc>release-v1.0.0</desc>
         </version>
        </versions>
    </middleware>
</middleware>
```

Note: *The latest tag is used to bring the latest version of the content available in the repository. This tag must be used to point to the latest version of the library. All tags mentioned in the manifest must exist in the repository hosting the content. See* Creating the release package *to learn more about creating these tags if not already done.*

5.3.1   Adding middleware dependencies

Middleware libraries can work standalone or require other middleware or other high-level libraries to work correctly. A middleware dependencies manifest is used to specify the dependencies a middleware may have on other middleware or high-level libraries.

Note: *The dependency manifest defines only dependencies between high-level middleware libraries. There is no need to define the dependencies on the low-level base libraries like HAL, PDL, or Core-Lib. Instead, the BSP dependencies manifest will define such dependencies separately for each BSP.*

For example, an MQTT middleware might have dependencies on Wi-Fi and HTTP libraries to function correctly. ModusToolbox™ offers a number of middleware libraries that can be added dependencies. These middleware libraries are divided into three categories:

- General middleware libraries (mtb -mw-manifest)
- Bluetooth ® middleware libraries (mtb - bt -mw-manifest)
- Wi-Fi middleware libraries (mtb - wifi -mw-manifest)

All the above links point to the manifest repositories used by ModusToolbox™ for organizing the middleware libraries. If your library has a dependency on a Wi-Fi library, use the mtb-wifi-mw-manifest repository to find the information required for your dependency manifest. Similarly, if your middleware has a dependency on a general library, use the mtb-mw-manifest repository to find the information required for your dependency manifest.

In each of these repositories, you will find manifest files that have the same typical structure:

Code Listing 9

```
<middleware>
    <name>bmm150</name>
    <id>bmm150</id>
    <uri>https://github.com/BoschSensortec/BMM150-Sensor-API</uri>
    <desc>Bosch BMM-150 Sensor Driver.</desc>
    <category>Peripheral</category>
    <req_capabilities>mcu_gp</req_capabilities>
    <versions>
      <version flow_version="1.0,2.0">
        <num>2.0.0 release</num>
        <commit>bmm150_v2.0.0</commit>
        <desc>2.0.0 release</desc>
      </version>
    </versions>
</middleware>
```

Based on which middleware library should be added as a dependency, locate the name of the middleware using the `<name>` label in the manifest file. For example, if your middleware has a dependency on the BMM150 sensor library, because this is a non-Bluetooth® or a non-Wi-Fi library, you will use the mtb-mw-manifest repository to search for it.

Once you have found the library, note the ID of the middleware in the <id> label. This will be used to specify the dependency.

Create a file in the `mtb-partner-mw-manifest` repository named `mtb-partner-mw-dependencies-manifest.xml`. The structure of the middleware dependencies manifest file is as follows:

Code Listing 10

```
<dependencies>
  <depender>
    <id>my-middleware-id</id>
    <versions>
      <version>
        <commit>2.0.0</commit>
        <dependees>
          <dependee>
            <id>dependent-library1-id</id>
            <commit>1.1.0</commit>
          </dependee>
          <dependee>
            <id>dependent-library2-id</id>
            <commit>1.2.0</commit>
          </dependee>
        </dependees>
      </version>
      <version>
        <commit>1.0.0</commit>
        <dependees>
          <dependee>
            <id>dependent-library1-id</id>
            <commit>1.0.0</commit>
          </dependee>
          <dependee>
            <id>dependent-library2-id</id>
            <commit>1.1.0</commit>
          </dependee>
        </dependees>
      </version>
    </versions>
  </depender>
</dependencies>
```

All the middleware dependencies are specified within the root `<dependencies>…</dependencies>` section. The dependencies of each middleware library are specified within the `<depender>` section. Because each middleware library can have dependencies on different versions of other libraries for a particular release, the `<version>` section is used to differentiate the dependencies across versions.

Update the body of the middleware dependencies file using the guidance for the fields described below to add details of your middleware:

Element	Description	Example
<id>	A unique identifier for the middleware. The manifest processing code will give an error if multiple middleware items have the same ID. Typically, the name is the same as the GitHub repository name.	*partner-middleware*

5 Creating your own manifest

Element	Description	Example
<commit>	Specifies the tag/branch used for fetching the library	*1.0.0*
<dependee><id>	Unique identifier of the dependent library	*bmm150*
<dependee><commit>	Specifies the tag/branch used for fetching the dependent library for a specific version of the depender library.	*bmm150_v2.0.0*

For example, if your middleware has a dependency on `wifi-connection-manager` and `http-client` to function correctly, the completed middleware dependencies file should look like this:

Code Listing 11

```
<dependencies>
  <depender>
    <id>partner-middleware</id>
    <versions>
      <version>
        <commit>latest-v1.X</commit>
        <dependees>
          <dependee>
            <id>wifi-connection-manager</id>
            <commit>latest-v2.X</commit>
          </dependee>
          <dependee>
            <id>http-client</id>
            <commit>latest-v1.X</commit>
          </dependee>
        </dependees>
      </version>

      <version>
        <commit>release-v1.0.0</commit>
        <dependees>
          <dependee>
            <id>wifi-connection-manager</id>
            <commit>latest-v2.X</commit>
          </dependee>
          <dependee>
            <id>http-client</id>
            <commit>latest-v1.X</commit>
          </dependee>
        </dependees>
      </version>
    </versions>
  </depender>
</dependencies>
```

The middleware manifest dependencies file reference can be found here.

5.4 Creating your BSP manifest

The BSP manifest is used to point to URIs of BSPs. Inside the `mtb-partner-bsp-manifest` repository, create a file named `mtb-partner-bsp-manifest-fv2.xml`.

The BSP manifest file has the following base structure:

Code Listing 12

```xml
<boards>
  <board default_location="local">
    <id>CUSTOM-BSP-NAME</id>
    <category>BSP Category</category>
    <board_uri>URL of the BSP</board_uri>
    <chips>
      <mcu>Name of the MCU chip used</mcu>
      <radio>Name of optional radio chip used</radio>
    </chips>
    <name>Name of the kit</name>
    <summary>Summary of the BSP</summary>
    <prov_capabilities>Capabilities of the BSP </prov_capabilities>
    <description> Detailed description of the BSP </description>
    <documentation_url>URL to BSP documentation </documentation_url>
    <versions>
      <version tools_min_version="3.0.0" flow_version="1.0,2.0"
prov_capabilities_per_version="bsp_gen4">
        <num>Latest 0.X release</num>
        <commit>latest-v0.X</commit>
      </version>
      <version tools_min_version="3.0.0" flow_version="1.0,2.0"
prov_capabilities_per_version="bsp_gen4">
        <num>0.5.0 release</num>
        <commit>release-v0.5.0</commit>
      </version>
    </versions>
  </board>
</boards>
```

The root of the XML contains the `<boards>` … `</boards>` section. Details of the BSPs are written inside the `<board>` section within the root `<boards>` node as shown below:

Code Listing 13

```xml
<boards>
    <board>
        // body of BSP 1
    </board>
    <board>
        // body of BSP 2
    </board>
</boards>
```
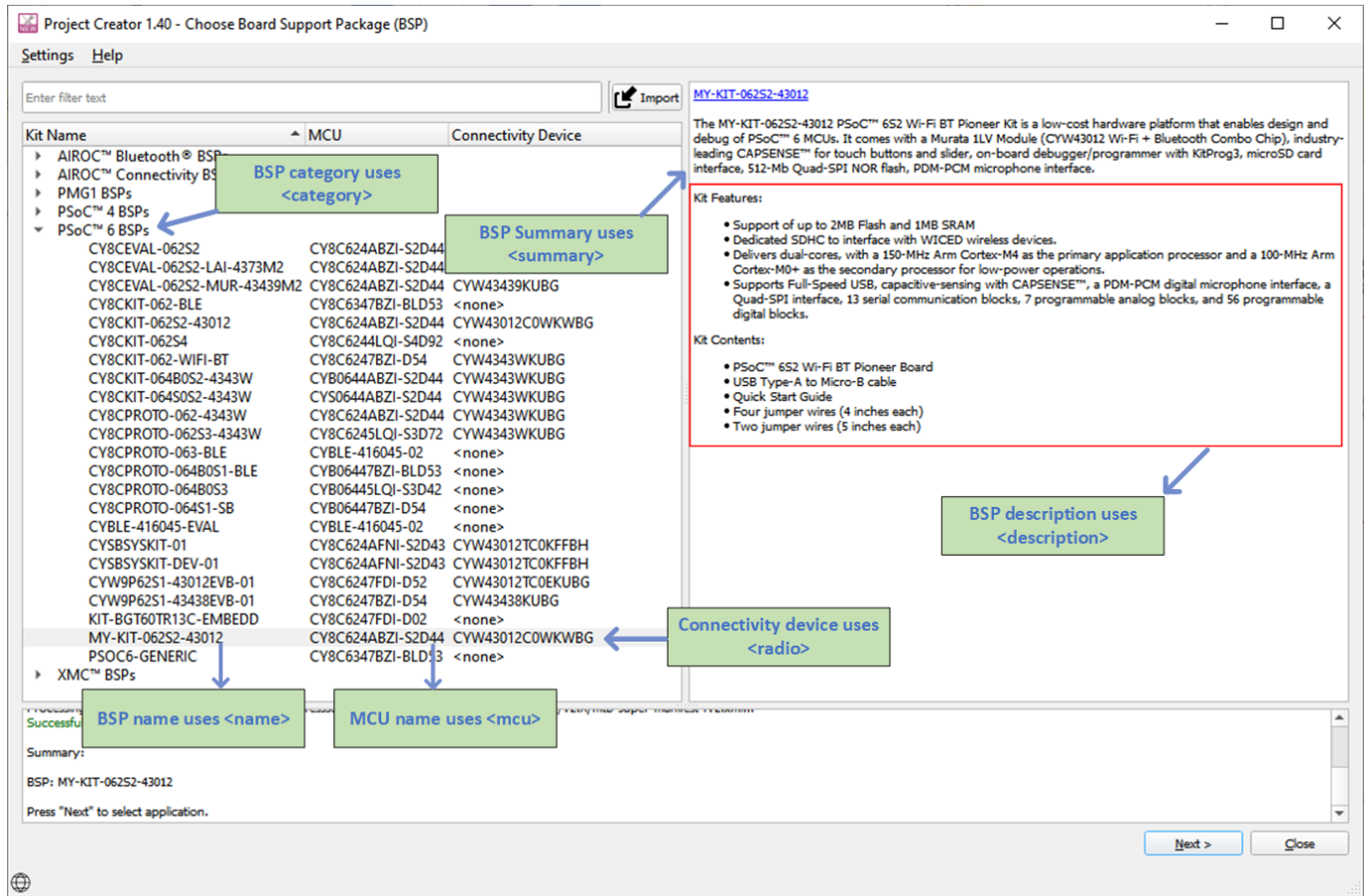
5 Creating your own manifest

Update the body of the board section using the guidance for the fields described below to add details of your middleware:

Element	Description	Example
board:default_location (optional)	Describes whether the BSP should be placed in the shared or local application folder	\<board default_location="local"> or \<board default_location="shared">
\<id>	A unique identifier for the custom BSP.The manifest processing code will give an error if multiple BSP items have the same ID. Typically, the name is the same as the GitHub repository name.	MY-CUSTOM-BSP
\<board_uri>	The URI for the Git repository holding the custom BSP	https://github.com/partner/TARGET_MY-CUSTOM-BSP
\<description>	A user-friendly text description of the BSP item. This is meant to be displayed in the UI. Typically, this is a few sentences that match the GitHub repository "About" text or the first paragraph in README.md. Note: Some asset descriptions apply advanced formatting like bold/inline text or line separators. In such situations, enclose the text into CDATA section.	\<![CDATA[My Custom BSP is an evaluation kit with the following features.]]>
\<category>	A user-friendly text string that specifies the category for displaying this BSP item in a GUI. It is expected that all BSP in the same category will be shown together in the Project Creator GUI.	PSoC™ 6 BSPs. Note that ™ is the special code representation of the trademark symbol.
\<prov_capabilities>	A list of capabilities that this BSP provides. This list is treated as an "AND" list. That is, all the capabilities that are provided. The list is whitespace-delimited; each item in the list must be a valid C identifier. If this element is missing or empty, it means that this BSP has no capability. For the complete list of the provided capabilities by the BSP, see Creating your code example manifest Adding BSP capabilities.	psoc6

5 Creating your own manifest

All BSPs are shown in the Project Creator tool. An illustration of how these fields affect the behavior in Project Creator is shown below:

An example of what the BSP manifest file will look like with all the details filled in is shown below:

5 Creating your own manifest

Code Listing 14

```
<board default_location="local">
    <id>MY-KIT-062S2-43012</id>
    <category>PSoC&#8482; 6 BSPs</category>
    <board_uri>https://github.com/partner/MY-KIT-062S2-43012</board_uri>
    <chips>
        <mcu>CY8C624ABZI-S2D44</mcu>
    </chips>
    <name>MY-KIT-062S2-43012</name>
    <summary>The BSP MY-KIT-062S2-43012 is the next generation board for XYZ applications</summary>
    <prov_capabilities>adc arduino capsense capsense_button capsense_linear_slider cat1 comp
dma flash_2048k hal i2c i2s j2 led low_power lptimer mcu_gp memory memory_qspi multi_core
nor_flash pdm psoc6 qspi rgb_led rtc sdhc smart_io spi sram_1024k std_crypto switch uart</prov_capabilities>
    <description><![CDATA[
        <div class="category">Kit Features:</div><ul>
        <li>Ready-to-Use CAPSENSE&#8482; Trackpad</li>
        <li>EZ-BLE PRoC module</li>
        <li>Potentiometer</li>
        <li>Rechargeable coin-cell battery</li>
        </ul><p/>
        <div class="category">Kit Contents:</div><ul>
        <li>MY-KIT-062S2-43012</li>
        <li>USB Standard-A to Micro-B cable</li>
        <li>ABC Sensor</li>
        <li>Four press-fit connectors (for Arduino headers)</li>
        <li>Four jumper wires</li>
        <li>Quick Start Guide</li>
        </ul>
    ]]></description>
    <documentation_url>https://www.partner.com/documentation/development-kitsboards/MY-KIT-062S2-43012</documentation_url>
    <versions>
        <version tools_min_version="3.0.0" flow_version="1.0,2.0"
prov_capabilities_per_version="bsp_gen4">
            <num>Latest v1.X release</num>
            <commit>latest-v1.X</commit>
        </version>
        <version tools_min_version="3.0.0" flow_version="1.0,2.0"
prov_capabilities_per_version="bsp_gen4">
            <num>1.1.0 release</num>
            <commit>release-v1.1.0</commit>
        </version>
        <version tools_min_version="3.0.0" flow_version="1.0,2.0"
prov_capabilities_per_version="bsp_gen4">
            <num>1.0.0 release</num>
            <commit>release-v1.0.0</commit>
        </version>
    </versions>
</board>
```

The BSP manifest file reference can be found here.

5.4.1 Adding BSP dependencies

All BSPs rely on low-level device-specific libraries to work correctly. A BSP dependencies manifest is used in to specify the dependencies a BSP has on the lower-level libraries.

For example, a PSoC™ 6 BSP has dependencies on PSoC™ 6 Peripheral Driver Library (mtb-pdl-cat1), PSoC™ 6 Hardware Abstraction Layer library (mtb-hal-cat1), core libraries (core-lib), etc. ModusToolbox™ offers a number of libraries that can be added as a dependency. These libraries are divided into three categories:

- General middleware libraries (mtb -mw-manifest)
- Bluetooth ® middleware libraries (mtb - bt -mw-manifest)
- Wi-Fi middleware libraries (mtb - wifi -mw-manifest)

All the above links point to the manifest repositories used by ModusToolbox™ for organizing the middleware libraries. If your library has a dependency on a Wi-Fi library, use the mtb-wifi-mw-manifest repository to find the information required for your dependency manifest. Similarly, if your middleware has a dependency on a general library, use the mtb-mw-manifest repository to find the information required for your dependency manifest.

In each of these repositories, you will find manifest files that have the same typical structure:

Code Listing 15

```
<middleware>
    <name>mtb-pdl-cat1</name>
    <id>mtb-pdl-cat1</id>
    <uri>https://github.com/cypresssemiconductorco/mtb-pdl-cat1</uri>
    <desc>The Peripheral Driver Library (PDL) integrates device header files, startup code, and
low-level peripheral drivers into a single package.</desc>
    <category>Core</category>
    <req_capabilities>cat1</req_capabilities>
    <versions>
      <version flow_version="1.0,2.0" tools_min_version="3.0.0">
        <num>3.0.0 release</num>
        <commit>release-v3.0.0</commit>
        <desc>3.0.0 release</desc>
      </version>
      <version flow_version="1.0,2.0" tools_min_version="2.4.1">
        <num>2.4.1 release</num>
        <commit>release-v2.4.1</commit>
        <desc>2.4.1 release</desc>
      </version>
      </version>
    </versions>
  </middleware>
```

Based on the middleware library that should be added as a dependency, locate the name of the middleware using the `<name>` label in the manifest file. For example, if your BSP has a dependency on the PSoC™ 6 PDL, because this is a non-Bluetooth® or a non-Wi-Fi library, you will use the `mtb-mw-manifest` repository to search for it.

Once you have found the library, note the ID of the middleware in the `<id>` label. This will be used to specify the dependency.

5 Creating your own manifest

Create a file in the `mtb-partner-bsp-manifest` repository named `mtb-partner-bsp-dependencies-manifest.xml`. The structure of the BSP dependencies manifest file is as follows:

Code Listing 16

```xml
<dependencies version="2.0">
  <depender>
    <id>my-bsp-id</id>
    <versions>
      <version>
        <commit>2.0.0</commit>
        <dependees>
          <dependee>
            <id>dependent-library1-id</id>
            <commit>1.1.0</commit>
          </dependee>
          <dependee>
            <id>dependent-library2-id</id>
            <commit>1.2.0</commit>
          </dependee>
        </dependees>
      </version>
      <version>
        <commit>1.0.0</commit>
        <dependees>
          <dependee>
            <id>dependent-library1-id</id>
            <commit>1.0.0</commit>
          </dependee>
          <dependee>
            <id>dependent-library2-id</id>
            <commit>1.1.0</commit>
          </dependee>
        </dependees>
      </version>
    </versions>
  </depender>
</dependencies>
```

All BSP dependencies are specified within the root `<dependencies>…</dependencies>` section. The dependencies of each BSP library are specified within the `<depender>` section. Because each BSP library can have dependencies on different versions of other libraries for a particular release, the `<version>` section is used to differentiate the dependencies across versions.

Update the body of the BSP dependencies file using the guidance for the fields described below to add details of your middleware:

5 Creating your own manifest

Element	Description	Example
<id>	A unique identifier for the BSP. The manifest processing code will give an error if multiple middleware items have the same ID. Typically, the name is the same as the GitHub repository name.	PARTNER-MY-KIT-062S2-43012
<commit>	Specifies the tag/branch used for fetching the library	1.0.0
<dependee><id>	Unique identifier of the dependent library	mtb-pdl-cat1
<dependee><commit>	Specifies the tag/branch used for fetching the dependent library for a specific version of the depender library	release_v2.0.0

For example, if your BSP has a dependency on PSoC™ 6 PDL and PSoC™ 6 HAL to function correctly, the completed BSP dependencies file should look like this:

Code Listing 17

```xml
<dependencies>
  <depender>
    <id>PARTNER-MY-KIT-062S2-43012</id>
    <versions>
      <version>
        <commit>latest-v1.X</commit>
        <dependees>
          <dependee>
            <id>mtb-pdl-cat1</id>
            <commit>latest-v3.X</commit>
          </dependee>
          <dependee>
            <id>mtb-hal-cat1</id>
            <commit>latest-v2.X</commit>
          </dependee>
        </dependees>
      </version>
      <version>
        <commit>release-v1.0.0</commit>
        <dependees>
          <dependee>
            <id>mtb-pdl-cat1</id>
            <commit>latest-v3.X</commit>
          </dependee>
          <dependee>
            <id>mtb-hal-cat1</id>
            <commit>latest-v2.X</commit>
          </dependee>
        </dependees>
      </version>
    </versions>
  </depender>
</dependencies>
```

The BSP manifest dependencies file reference can be found here.

5.5   Creating your super manifest

The super manifest is used to point to the URIs of your `code example`, `middleware`, and `BSP manifest` files. Inside the `mtb-partner-super-manifest` repository, create a file named `mtb-partner-super-manifest-fv2.xml`.

The super manifest has the following base format:

_____

5 Creating your own manifest

Code Listing 18

```
<super-manifest version="2.0">
  <board-manifest-list>
    <board-manifest>
      <uri>https://github.com/partner/mtb-partner-bsp-manifest/raw/main/mtb-partner-bsp-
manifest.xml</uri>
    </board-manifest>
  </board-manifest-list>
  <app-manifest-list>
    <app-manifest>
      <uri>https://github.com/partner/mtb-partner-ce-manifest/raw/main/mtb-partner-ce-manifest-
fv2.xml</uri>
    </app-manifest>
  </app-manifest-list>
  <middleware-manifest-list>
    <middleware-manifest>
      <uri>https://github.com/partner/mtb-partner-mw-manifest/raw/main/mtb-partner-mw-
manifest.xml</uri>
    </middleware-manifest>
  </middleware-manifest-list>
</super-manifest>
```

At the root, the `<super-manifest>` label is used to identify that is a super manifest type and flow version of the manifest is v2.0. It then uses `<board-manifest-list>`, `<app-manifest-list>`, and `<middleware-manifest-list>` sections for listing the BSPs, code examples, and middleware libraries to be brought in.

The URL to the manifest XMLs must be in raw format. To create this link, use the following scheme:

`<link to repo>/raw/<branch_name>/<manifest_name>.xml`

For example, a repository named mtb -partner- ce -manifest and having the manifest file mtb-partner-ce-manifest-fv2.xml on branch "master" will have the following link:

https://github.com/Infineon/mtb-partner-ce-manifest/raw/master/mtb-partner-ce-manifest-fv2.xml

To specify the BSP manifests, each BSP manifest URIs can be added using the `<board-manifest>` sections within the `<board-manifest-list>` using the `<uri>` field. For example, if you have two different BSP manifests you want to point to, it can be done this way:

Code Listing 19

```
<board-manifest-list>
    <board-manifest>
      <uri>https://github/partner/mtb-partner-bsp-manifest/raw/main/mtb-partner-bsp-
manifest.xml</uri>
    </board-manifest>
    <board-manifest>
      <uri>https://github/partner/mtb-partner-bsp-manifest/raw/main/other-bsp-manifest.xml</uri>
    </board-manifest>
  </board-manifest-list>
```

Similarly, to specify the code example manifests, each of the code example manifest URIs can be added using the `<app-manifest>` sections within the `<app-manifest-list>` using the `<uri>` field. For example, if you have two different code example manifests that you want to point to, it can be done this way:

5 Creating your own manifest

Code Listing 20

```
<app-manifest-list>
    <app-manifest>
       <uri>https://github.com/partner/mtb-partner-ce-manifest/raw/main/mtb-partner-ce-manifest-
fv2.xml</uri>
    </app-manifest>
    <app-manifest>
       <uri>https://github.com/partner/mtb-partner-ce-manifest/raw/main/other-ce-manifest-
fv2.xml</uri>
    </app-manifest>
  </app-manifest-list>
```

Similarly, to specify the middleware manifests, each middleware manifest URI can be added using the `<middleware-manifest>` sections within the `<middleware-manifest-list>` using the `<uri>` field. For example, if you have two different code example manifests that you want to point to, it can be done this way:

Code Listing 21

```
<middleware-manifest-list>
    <middleware-manifest>
       <uri>https://github.com/partner/mtb-partner-mw-manifest/raw/main/mtb-partner-mw-
manifest.xml</uri>
    </middleware-manifest>
    <middleware-manifest>
       <uri>https://github.com/partner/mtb-partner-mw-manifest/raw/main/other-mw-manifest.xml</
uri>
    </middleware-manifest>
  </middleware-manifest-list>
```

The super manifest should finally look like this once all the details have been filled in:

Code Listing 22

```
<super-manifest version="2.0">
  <board-manifest-list>
    <board-manifest>
      <uri>https://github.com/partner/mtb-partner-bsp-manifest/raw/main/mtb-partner-bsp-
manifest.xml</uri>
    </board-manifest>
  </board-manifest-list>
  <app-manifest-list>
    <app-manifest>
      <uri>https://github.com/partner/mtb-partner-ce-manifest/raw/main/mtb-partner-ce-manifest-
fv2.xml</uri>
    </app-manifest>
  </app-manifest-list>
  <middleware-manifest-list>
    <middleware-manifest>
      <uri>https://github.com/partner/mtb-partner-mw-manifest/raw/main/mtb-partner-mw-
manifest.xml</uri>
    </middleware-manifest>
  </middleware-manifest-list>
</super-manifest>
```

The super manifest file reference can be found here.

Note: You will probably only have a single manifest each for BSP, code examples and middleware unless you need to bifurcate the manifests for organizing content a certain way.

Note: In cases when you don't deal with a particular manifest, that entire list can be omitted from the super manifest. For example, if you don't have a BSP manifest, the `<board-manifest-list> .. </board-manifest-list>` section can be removed entirely.

5.6   Specifying the dependency manifests

If the middleware or BSP manifests have a dependency manifest attached to them, they should be specified in the super manifest using the `dependency-url` field in the following manner:

Code Listing 23

```
<super-manifest version="2.0">
  <board-manifest-list>
    <board-manifest dependency-url="https://github.com/Infineon/mtb-partner-bsp-manifest/raw/
master/mtb-partner-bsp-dependencies-manifest.xml">
      <uri>https://github.com/Infineon/mtb-partner-bsp-manifest/raw/master/mtb-partner-bsp-
manifest-fv2.xml</uri>
    </board-manifest>
  </board-manifest-list>
  <app-manifest-list>
    <app-manifest>
      <uri>https://github.com/Infineon/mtb-partner-ce-manifest/raw/master/mtb-partner-ce-
manifest-fv2.xml</uri>
    </app-manifest>
  </app-manifest-list>
  <middleware-manifest-list>
    <middleware-manifest dependency-url="https://github.com/Infineon/mtb-partner-mw-
manifest/raw/master/mtb-partner-mw-dependencies-manifest.xml">
      <uri>https://github.com/Infineon/mtb-partner-mw-manifest/raw/master/mtb-partner-mw-
manifest-fv2.xml</uri>
    </middleware-manifest>
  </middleware-manifest-list>
</super-manifest>
```

This field can be skipped if there are no dependencies attached to the manifests.

5.7   Validating your manifests

As a final check, all the manifests must be validated using the manifest checker tool. The manifest checker tool runs a suite of tests to check for issues in format, schema, syntax, and whether the URIs pointed to contain valid assets.

Follow these steps to check-in valid manifests into your repository:

1. Clone the mtb -manifest-checker repo into your local machine.

2. On Windows, run `modus-shell` by typing `modus-shell` in the Windows search bar. On Linux and MacOS, run any shell application.

3. Navigate to the folder where the repo was cloned and create the folders `apps`, `bsp`, and `mw`. Copy all the `app manifests`, `bsp manifests`, and `middleware manifests` into the `apps`, `bsp`, and `mw` folders respectively.

4. Run the following command to test your manifest for format, schema and syntax. You can just test for format or schema or syntax as well.

```
>> ./mtb_manifest_checker.sh --syntax --format --schema apps/mtb-partner-ce-manifest-fv2.xml
```

```
/cygdrive/c/mtb-manifest-checker                                          —   □   ×

                              /cygdrive/c/mtb-manifest-checker
$ mkdir apps bsp mw

                              /cygdrive/c/mtb-manifest-checker
$ ./mtb_manifest_checker.sh --syntax --format --schema apps/partner-ce-manifest-fv2.xml
```
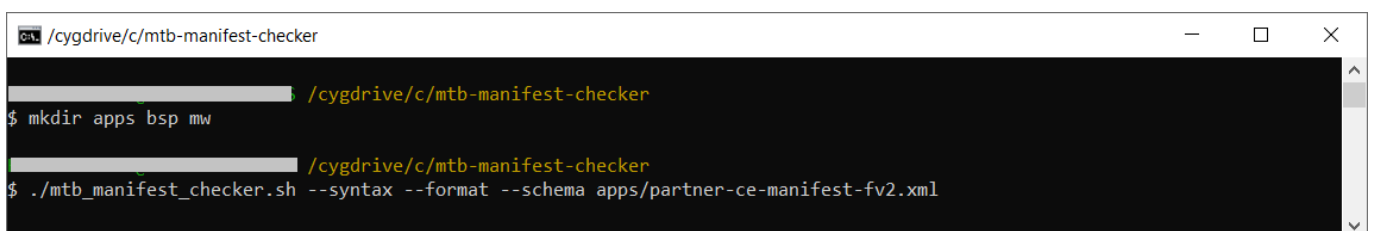
5 Creating your own manifest

Refer to the Readme documentation for each of the tests to understand how to resolve the errors.

5. If you have multiple manifests in the `local` folder, you can use the "*" wildcard to test all the manifests at once.

```
>> ./mtb_manifest_checker.sh --syntax --format --schema apps/*.xml bsp/*.xml mw/*.xml
```

6. After all the manifests have been tested, push the manifests upstream into their respective git repository.

7. Test the manifests checked into the repositories by running the tests on the super-manifest chain to ensure nothing is broken.

```
>> ./mtb_manifest_checker.sh --syntax --format --schema --assets https://github.com/Infineon/
mtb-partner-super-manifest/raw/<topic_branch>/mtb-partner-super-manifest-fv2.xml
```

For more information, see the Readme of the manifest checker tool.

8. Once all the tests have passed, you can integrate the manifest and test the content directly in the tools like Project Creator and Library Manager as explained in Testing the manifest integration.

6   Testing the manifest integration

Once the content and manifests are created, the integration can be tested in ModusToolbox™ using the following steps:

1. Navigate to the `.modustoolbox` folder (note the dot at the beginning of the folder name) which is located in your user profiles directory where ModusToolbox™ is installed by default. Sometimes this folder may be hidden, so unhide it in the file explorer view options to find it. For example, `C:/Users/Username/.modustoolbox`

2. Add a file to this folder named `manifest.loc`.
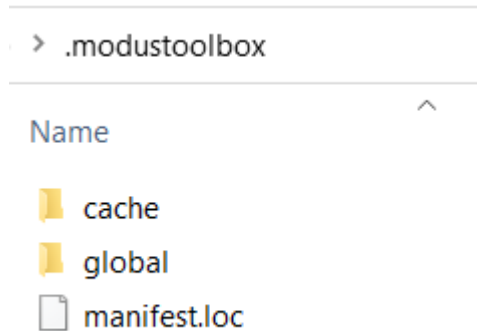
Use an editor of your choice to edit the manifest.loc file to add the link to your super manifest such as https://github.com/Infineon/mtb-partner-super-manifest/raw/master/mtb-partner-super-manifest-fv2.xml.

3. Run the Project Creator tool. It will list all the super manifests it is processing to bring the content in as shown below:
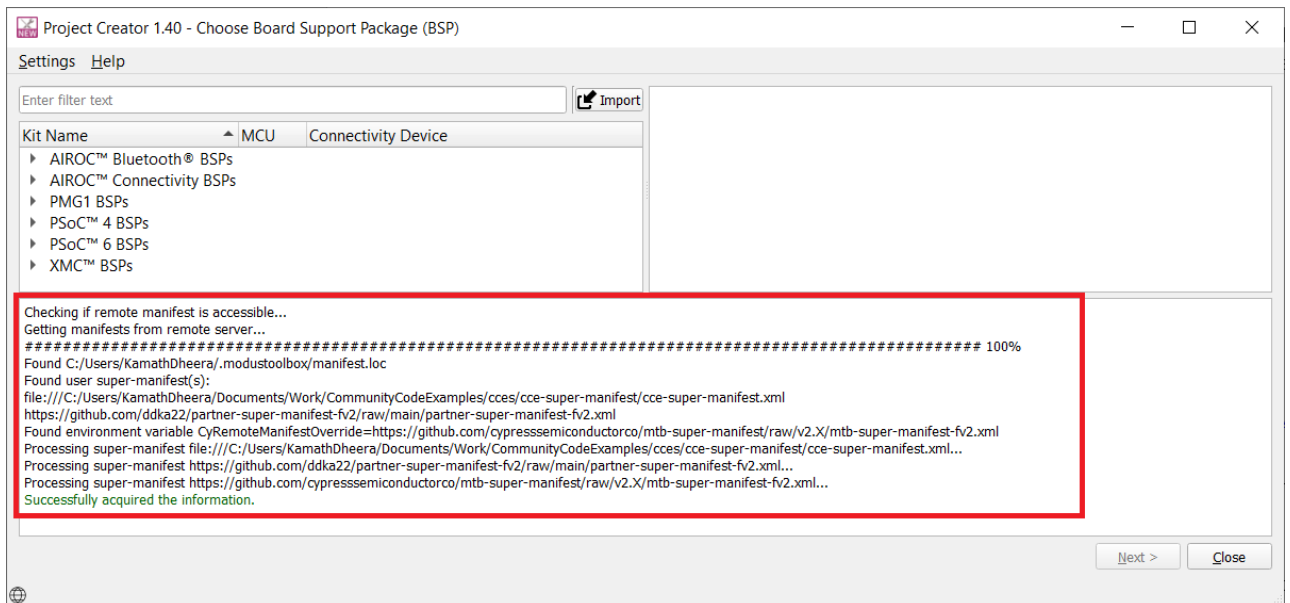

Note: If the tools were open previously, you will have to close and open them again for the manifest.loc changes to take effect. If any issues are encountered, Project Creator will display an error indicating the loading of the manifests failed as shown below:
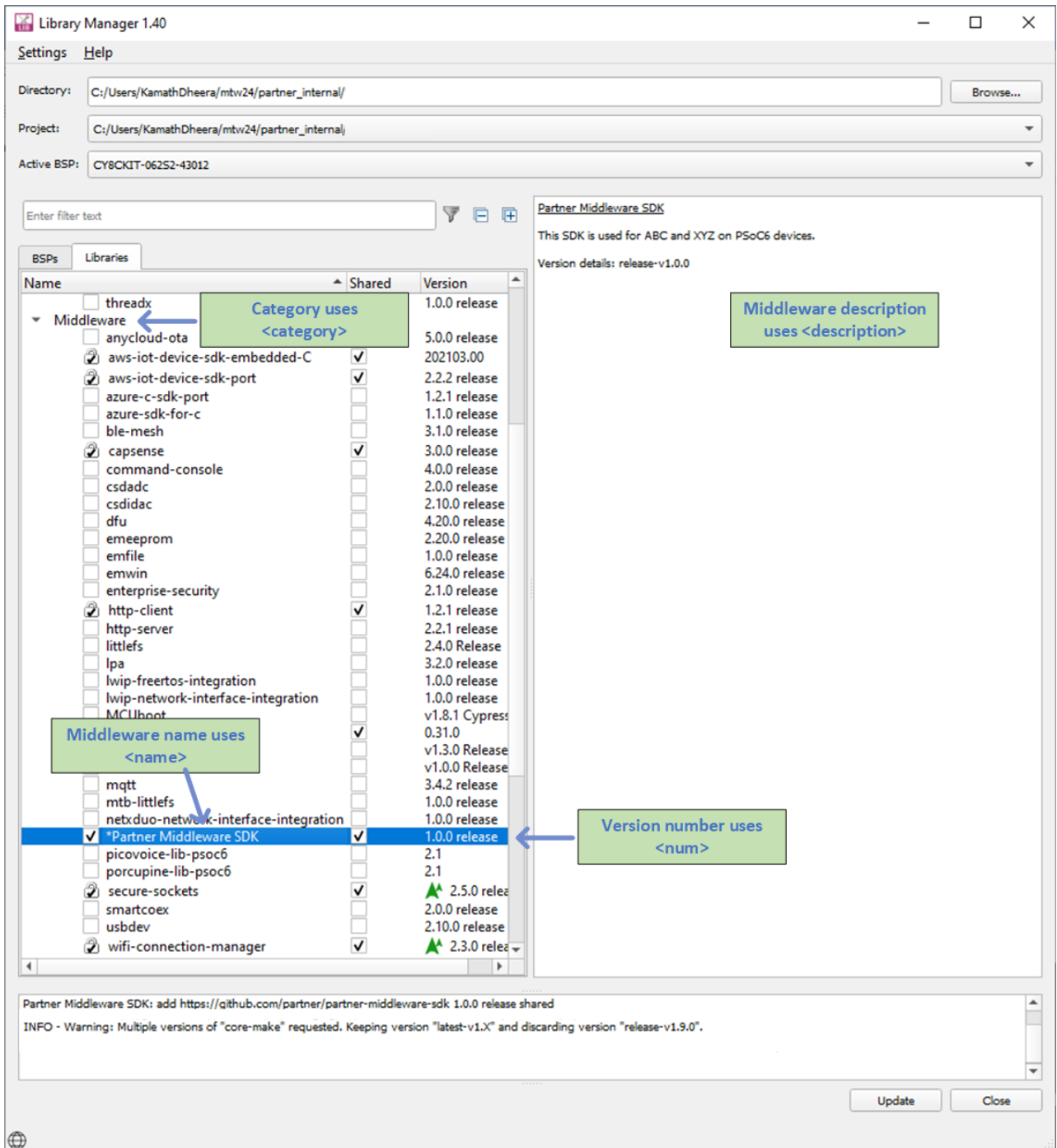
6 Testing the manifest integration
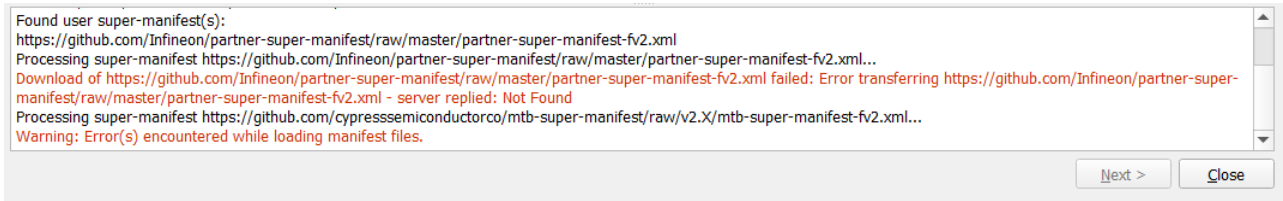
You should be able to see the BSPs and code examples that you added being displayed in their respective tab and category.
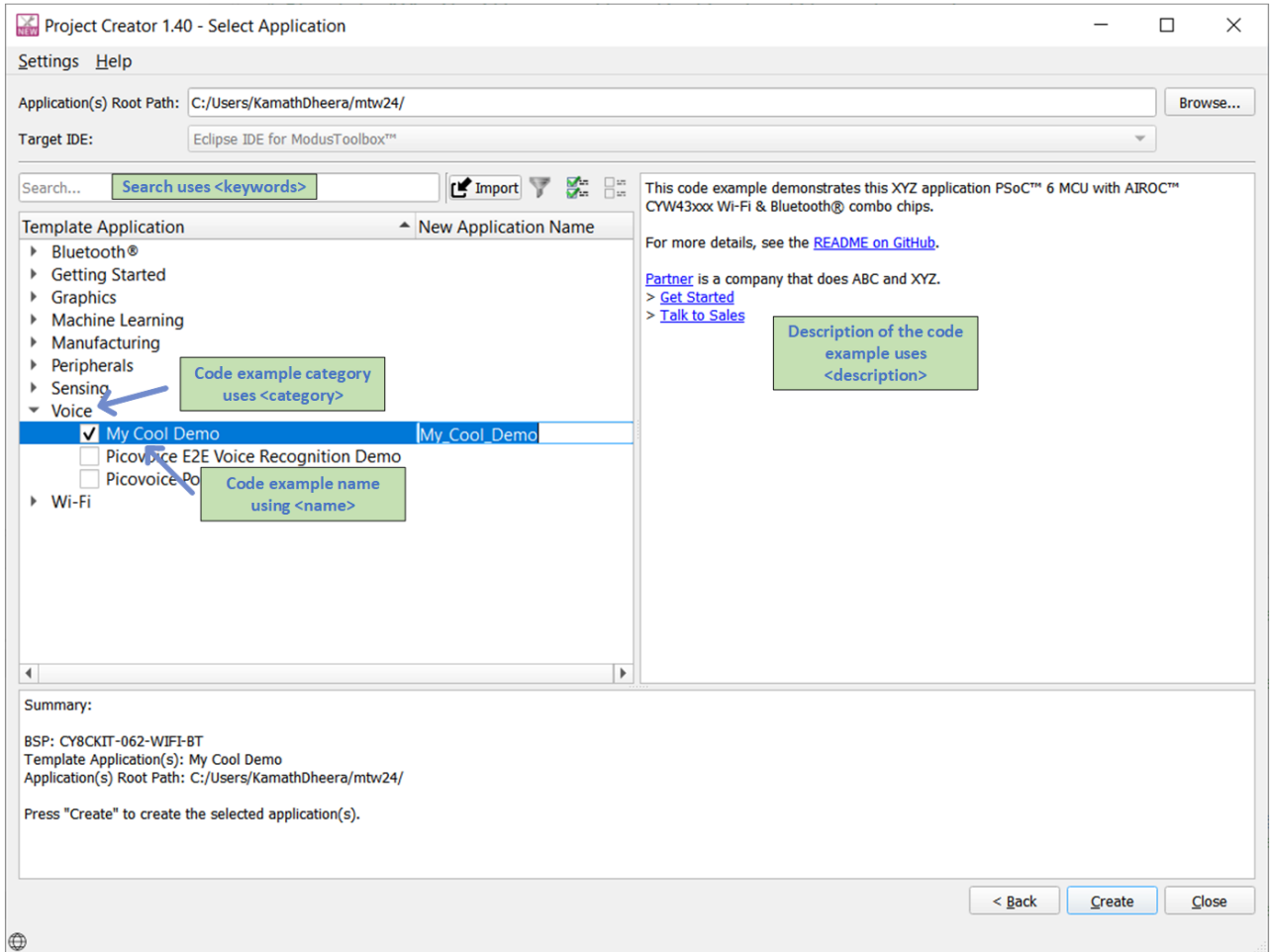
4. Create an application that uses your solution.
5. Run the Library Manager tool.
6. You should be able to see the middleware that you added being displayed in its respective tab and category.

If you are able to see all the content displayed, you now have a successful setup in place to update your software content and make it available to ModusToolbox™ users seamlessly.

6.1   Testing the dependency manifest integration

As discussed previously, the middleware and BSPs can have dependencies on other libraries that are specified using the dependencies manifest. The expectation is that whenever any code example uses the .mtb file of the middleware or BSP, the corresponding dependent libraries are brought in automatically by ModusToolbox™ tools without the users needing to add them manually.

Follow these steps to check if the dependent libraries are being brought in:

1.   In your code example, add the .mtb file to the deps folder that points to your middleware or BSP that has dependent libraries.

2. Push these changes to the upstream server and create a release tag. Update the code example manifest to add support for this new release.

3. Open Project Creator and select the code example. Click **Create**.

In the Project Creator log, you should see the following sets of lines that indicate the resolution of the dependencies:

```
Resolving dependencies...
Checking if remote manifest is accessible...
Getting manifests from remote server...
Found C:/Users/User/.modustoolbox/manifest.loc
Processing super-manifest https://github.com/Infineon/mtb-partner-super-manifest/raw/main/
mtb-partner-super-manifest-fv2.xml...
Successfully acquired the information.

INFO - Warning: Multiple versions of "core-make" requested. Keeping version "latest-v1.X"
and discarding version "release-v1.9.0".
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/abstraction-rtos.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/capsense.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/clib-support.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/connectivity-utilities.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/core-lib.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/core-make.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/cy-mbedtls-acceleration.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/freertos.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/lwip.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/mbedtls.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/mtb-hal-cat1.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/mtb-pdl-cat1.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/psoc6cm0p.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/recipe-make-cat1a.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/secure-sockets.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/whd-bsp-integration.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/wifi-connection-manager.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/wifi-host-driver.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/wifi-mw-core.mtb was added
C:/Users/User/mtw/partner_internal/Partner_Demo/libs/wpa3-external-supplicant.mtb was
added
Dependencies resolved.
```

4. Once the project is created, check if all the dependent libraries are available in the mtb_shared directory. If all the libraries are available, the dependencies integration is a success.

6.2 Out-of-the-box testing

Now that the content is visible in ModusToolbox™ tools, test if the content is being downloaded correctly and works as expected. During the creation of the code example, Project Creator should import all the necessary libraries and the middleware automatically. The code example should build out-of-the-box. To test this, do the following (it is assumed you have added your super manifest file to `manifest.loc` file):

1. Run Project Creator.

2. Select the code example you added in your code example manifest, and click **Create**.

3. The code example will be cloned from your Git repository and all the dependencies will be brought in.

4. Now, build your code example and make sure it works as expected. If there are modifications to be made for the build to be successful, document the steps in the ReadMe.md file.

5. Program the code on the hardware to verify the functionality.

7   Integrating into ModusToolbox™

Once the integration of the content and manifest has been tested, do the following to get the content integrated formally with ModusToolbox™:

1. Reach out to Infineon by creating a ticket in the case system as documented in Technical Support. Provide a brief description indicating that the ticket is a request for your content to be integrated along with links to your manifests.

2. Infineon technical support will get in touch with you and support you with the integration. The manifests will be reviewed and content will be tested to ensure that everything works as expected in ModusToolbox™.

3. Any concerns with the manifest or the content will be raised in the ticket. Partners must address all the concerns and provide an update of the resolution in the ticket.

4. Once the technical support team finds the content and manifest ready to be integrated, the ModusToolbox™ super manifest will be updated to point to the partner manifests.

5. The update pertaining to the integration will be provided via the ticket. Partners can validate the integration and close the ticket if everything looks good.

Partners can continue to manage their content via their own manifests. The partners are free to manage, update, and create new content using their manifests without any need to contact Infineon. However, care should be taken while updating manifests to prevent errors. Any update to the content or manifests should follow the guidelines described in section Updating your content.

Steps 1 through 5 should be repeated only when new manifest files need to be integrated. For further queries, contact Technical Support by creating a ticket (see Technical Support).

8 Updating your content

Now that the initial version of the content and manifest is already integrated into ModusToolbox™, any update should be handled with care. Infineon reserves the right to temporarily pull down any content or manifest that does not work as intended or affects the user experience in ModusToolbox™ until it is rectified.

When updating your content, the following steps should be followed.

1. Clone repositories
2. Create a topic branch
3. Update and test your content
4. Merge into mainline
5. Create a release package
6. Update the corresponding manifest
7. Update the dependency manifest
8. Testing the integration

Note: *Partners must strictly follow the guidelines to update the content and manifests to ensure it doesn't break the user experience in ModusToolbox™. Infineon will provide an indication via proper communication channels to partners to rectify their content when such cases arise. If the guidelines are not followed and these problems keep recurring, the content and manifests will be permanently taken down after three such occurrences.*

8.1 Cloning the repositories

Clone the repository of the content you want to update. Use `git clone` command to clone your repository.

8.2 Creating a topic branch

Create a new topic branch to work on the updates. This prevents any accidental code being pushed to the mainline that can affect your users. Use `git checkout -b <branch_name>` command to create a new topic branch.

8.3 Updating and testing your content

Update the source code and documentation. Make sure that the changes are tested thoroughly. If the source code is updated, the functional tests should be conducted to verify it is working on the hardware.

For content that has dependencies on others, the asset that has the dependencies should also be tested. If the asset breaks, the asset should be updated to work with the updated dependencies. For example, a code example has dependencies on middleware. If the middleware is updated, the code example should also be tested to verify if it works with the updated middleware. If not, the code example should be updated to work with the updated middleware.

8.4 Merging into mainline

Merge the fully tested content into the main branch. Any merge conflicts that may arise as a result of this action should be resolved. If this is a dual-stage setup, deploy the contents into production.

8.5 Creating the release package

The release tags should be created for any new update to the content. See the table in Creating the release package to understand how to update the version number.

8.6 Updating the corresponding manifest

Once the release tag is created, the corresponding manifest should be updated based on the content being updated. For example, if the code example is updated and a new release tag "release-v2.0.0" is created, the CE manifest should be updated to reflect this change as highlighted in bold below:

Code Listing 24

```
<apps version="2.0">
  <app keywords="psoc6,partner,my,cool,demo">
    <name>My Cool Demo</name>
    <category>Voice</category>
    <id>partner-mtb-example-my-cool-demo</id>
    <uri>https://github.com/partner/partner-mtb-example-my-cool-demo</uri>
    <description><![CDATA[This code example demonstrates this XYZ application PSoC™ 6 MCU with
AIROC™ CYW43xxx Wi-Fi & Bluetooth® combo chips.
    <br><br>For more details, see the <a href="https://github.com/partner/partner-
mtb-example-my-cool-demo/blob/master/README.md">README on GitHub</a>.<br><br><a href="https://
partner-webpage.com/">Partner</a> is a company that does ABC and XYZ.<br> > <a
href="https://docs.partner-doc-page.com/">Get Started</a><br> > <a href="https://partner-
webpage.com/contact/">Talk to Sales</a>]]></description>
    <req_capabilities>psoc6</req_capabilities>
    <versions>
<version flow_version="2.0" tools_min_version="3.0.0" req_capabilities_per_version="bsp_gen4">
        <num>2.0.0 release</num>
        <commit>release-v2.0.0</commit>
      </version>
      <version flow_version="2.0" tools_min_version="2.4.0"
req_capabilities_per_version="bsp_gen3">
        <num>1.0.0 release</num>
        <commit>release-v1.0.0</commit>
      </version>
    </versions>
  </app>
</apps>
```

Note: *Only the CE, BSP, MW, and corresponding dependency manifests will be updated whenever there is an update. The super manifest will never be modified unless there is a new manifest file that should be pointed to. In that case, you must follow the steps in* Integrating into ModusToolbox™ *to get your super manifest changes integrated into the ModusToolbox™ super manifest.*

8.7 Updating the dependency manifest

If the update to the content requires any newer versions of the dependent libraries, the dependency manifest should be updated to reflect this change. For example, if the middleware is updated to `release-v2.0.0` and it uses newer version `latest-v3.x` of the `wifi-mw-manager` library, the middleware dependency manifest should reflect that change as highlighted in bold below:

Code Listing 25

```
<dependencies>
  <depender>
    <id>partner-middleware-sdk</id>
    <versions>
      <version>
                    <commit>release-v2.0.0</commit>
                    <dependees>
                  <dependee>
                  <id>wifi-connection-manager</id>
                  <commit>latest-v3.X</commit>
                  </dependee>
                  <dependee>
                  <id>http-client</id>
                  <commit>latest-v1.X</commit>
                  </dependee>
                </dependees>
            </version>
        <version>
          <commit>release-v1.0.0</commit>
          <dependees>
            <dependee>
              <id>wifi-connection-manager</id>
              <commit>latest-v2.X</commit>
            </dependee>
            <dependee>
              <id>http-client</id>
              <commit>latest-v1.X</commit>
            </dependee>
          </dependees>
        </version>
      </versions>
    </depender>
  </dependencies>
```

8.8   Testing the integration

Follow the steps described in Validating your manifests and Testing the manifest integration to validate your manifests and check if the content is working as expected post integration.

Note that since you created a new branch for manifest updates, you need to pass the super manifest file from this new branch into the manifest checker tool.

```
>> ./mtb_manifest_checker.sh --syntax --format --schema --assets https://github.com/Infineon/
mtb-partner-super-manifest/raw/<topic_branch>/mtb-partner-super-manifest-fv2.xml
```
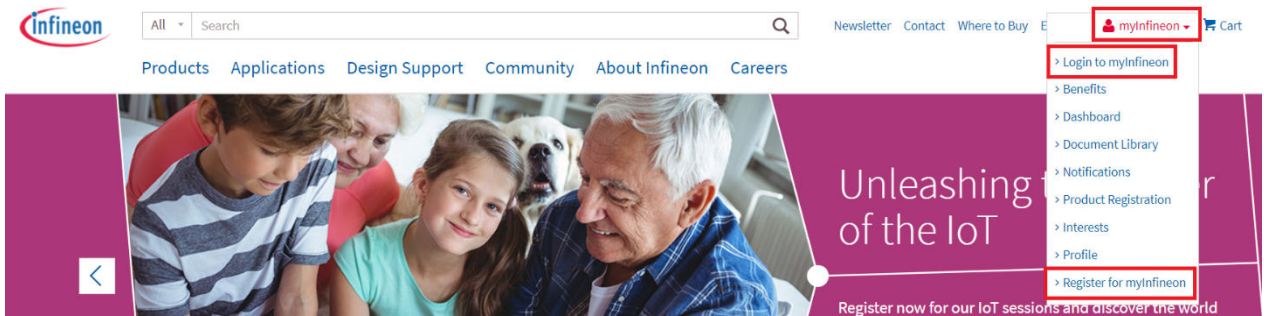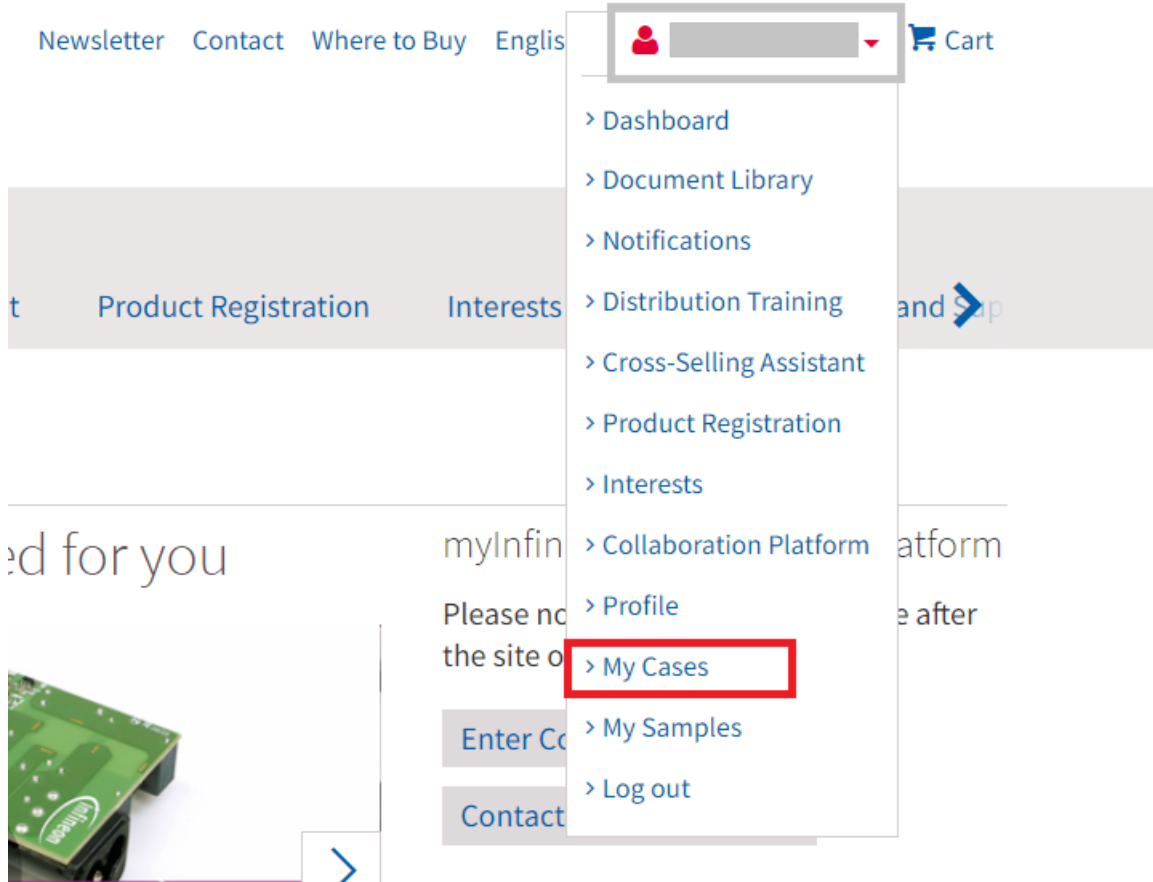
9 Technical Support

This section documents how you can contact Technical Support when you face any issues or have questions about your implementation. Infineon offers a case system to create tickets to seek technical support. Additionally, once you have successfully tested your manifest implementation, reach out to Infineon via the technical support channel to request the manifests to be integrated into ModusToolbox™.

Here are the steps to contact technical support:

1. Navigate to https://www.infineon.com/.
2. In the right-top corner, select the drop-down that reads **myInfineon** and click **Login to myInfineon**. For new users, click **Register for myInfineon** to create your account and then login. The steps are very straightforward.
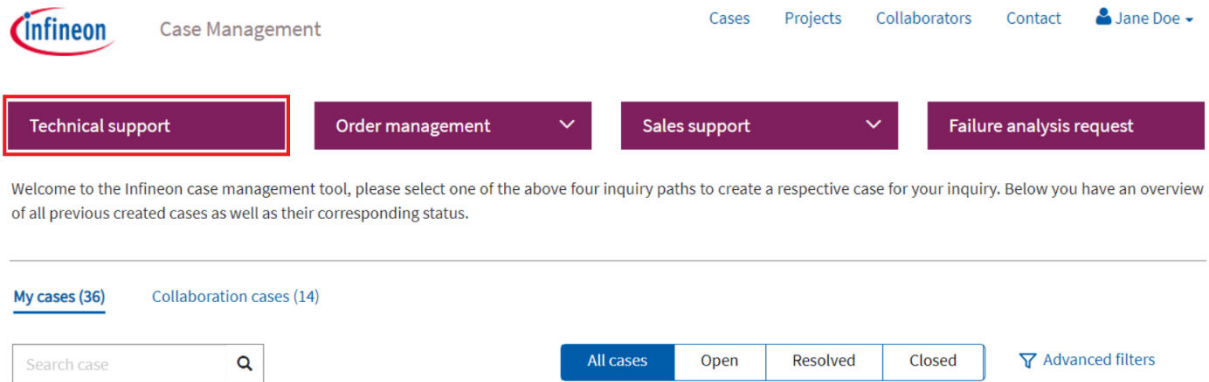

3. Once logged in, click **MyCases** in the drop-down under your profile.

4. On the **MyCases** webpage, choose **Technical Support** option as highlighted below:

9 Technical Support

You will be directed to a new webpage with a form to fill out the details of your query. The table below lists the various fields and values that need to be entered.

| Field	Description	Example
Subject	Specify a summary of the issue or query.	Issue with super manifest
Inquiry type	Select the type of query. Choose between different options available in the drop-down. Use "CY Software Tools" for queries related to this document.	CY Software Tools
Priority	Specify the priority of the request. Choose between low, medium, and critical based on how much the request affects your work and how soon you want the resolution.	Low
Due date	Provide the deadline/final date.	-
Product ID	Select the Product ID. Choose between different options available in the drop-down.	SP005672751
Other product name	Specify alternate product name.	-
Application	Provide the application you were working on which led to the query.	MODUSTOOLBOX & FRIENDS
Final Customer/OEM	Specify the account associated with the customer creating the ticket. Use the partner account issued to you as part of the partner program.	CompanyName
Project	Specify the project related to the issue. Use "MODUSTOOLBOX & FRIENDS" category for all issues related to this document.	MODUSTOOLBOX & FRIENDS

---

9 Technical Support

Field	Description	Example
Description	Describe the issue in detail. Make sure that you provide relevant screenshots under "Attachment" to illustrate your issue better.	Hi, I have problems implementing the super manifest. Here is what I implemented. Please see the attached file. I see the following issues.

Note: *If you have successfully implemented the manifests, mention the same in the description and request the Technical Support team to have them integrated into ModusToolbox™. The Technical Support team will review the manifests and have them integrated. Post integration, updates will be provided on the ticket.*

5. Click **Submit** to submit your request. After submitting the form, a case summary will be shown. Click **Add** files to add attachments up to 50 MB to the case.

Attachments

Disclaimer: Please note that once you have uploaded a document to the system you will not be able to delete/remove it!

Add files
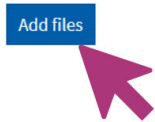
6. A ticket will be generated and the status can be viewed under **My Cases** section.

My cases (36)   Collaboration cases (14)

Case number	Case type	Subject	Status	Created on	Last modified on ↓
IFX-220818-020500	Product pricing	Availability of Product	New	18/08/2022	31/08/2022
IFX-220830-020979	Failure analysis request	Ref2387234890	New	30/08/2022	30/08/2022

A technical support engineer will be assigned to the thread within the next 24 hours. They will reach out to you via the ticket by adding their comments.

7. Use the **Discussion Board** box to interact with the technical support engineers. If your issue is resolved, use the **Resolve** button and click **Close Case**.

Quick Create: Discussion Board Messages

⚠ The external comment will be visible to the customers with access to this case.

Interaction type*

External

Description*

Save and Close ∨   Cancel

8. Use steps 1 through 8 to create a new case whenever you face any issues in the future.

10 Summary

This application note has shown how partners can get onboarded into the "ModusToolbox™ & Friends" program and contribute content to the ModusToolbox™ ecosystem. As mentioned earlier, this provides partners the ability to showcase content directly to developers using ModusToolbox™ to create interesting applications.

Appendix A – Partners integrated into ModusToolbox™ in this document features a list of partners who have been successfully integrated as part of this program and reference links to their content.

11 Appendix A – Partners integrated into ModusToolbox™

This appendix features a representative partner successfully integrated into ModusToolbox™ as part of the "ModusToolbox™ & Friends" program.

11.1 Memfault

Memfault is the first cloud-based observability platform for connected device debugging, monitoring, and updating, which brings the efficiencies and innovation of software development to hardware processes.

Memfault, as part of "ModusToolbox™ & Friends", leveraged the benefits of the program and developed content to make remote debugging and monitoring accessible to developers using ModusToolbox™. With the combination of Infineon's powerful PSoC™ 6 MCUs and their proprietary firmware SDK, they demonstrated an innovate solution on how to approach debugging.

Here are some quick links to the content and manifests developed by Memfault (for reference only):

- Memfault Super Manifest
- Memfault Middleware Manifest
- Memfault Code Example Manifest
- Memfault Middleware
- Memfault Code Example

Revision history

Document version	Date of release	Description of changes
**	2022-07-29	Initial release
*A	2023-02-14	Update for ModusToolbox™ 3.x
*B	2023-06-09	Update naming scheme for manifests Update all manifest URIs with new naming scheme

Trademarks

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

Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. 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) with respect to any and all information given in this application note.

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.