1. PURPOSE/SCOPE

1.1 Purpose

This document defines the design of the ModusToolbox Query APIs (MTBQueryAPIs) as well as the console mode applications that are part of this project.

1.2 Scope

This document covers just the design of the MTBQueryAPIs as well as mtbsearch, mtbquery, mtbgetlibs, mtblaunch, and lcs-manager-cli. This document does not cover the integration of the MTBQueryAPIs into various applications that are part of ModusToolbox. This document also does not cover the integration of the console mode executables into the ModusToolbox build environment. Since the overall architecture of the MTBQueryAPIs are covered in the SAS, this document is more focused on the implementation of the APIs and associated console mode programs.

Note, this document does not and should not provide the API documentation for the APIs. The specific API information is available through the doxygen generated API documentation.

1.3 Definitions

Term	**Meaning**
SAS	This term refers to the specification 002-30759 ModusToolbox 3.x Software Architecture Specification. The SAS contains additional definitions that are used by this document. It is assumed that anyone reading this document is familiar with the SAS.
API	Application Programming Interface
Client	When discussing the use of the MTBQueryAPIs any reference to the platform application (i.e. Windows Application, Mac OS Application, or Linux Application) will be made using the term “client”. The term “application” is reserved to its SAS context which is a customer “project” within the ModusToolbox environment.
Application	In the MTBQueryAPI context, the term “application” refers to a customer application within the ModusToolbox environment. This term, combined with the term project described below, captures the structure of a ModusToolbox customer effort. Prior to ModusToolbox 3.0, there was not separate application and project directory, but rather both of these entities were stored in the same directory. Therefore the term application will be used to mean the top level directory. This directory will always include application information and may also contain project information.

Term	Meaning
Project	In the MTBQueryAPI context, the term “project” refers to a specific project within a ModusToolbox application. There are two distinct types of application/project structures. The first of these stems from ModusToolbox 2.x where the application directory and the project directory are the same directory. ModusToolbox 3.x introduces multi-core applications where each project is a subdirectory of the application directory.
FlowVersion	This is a well defined ModusToolbox term defining the ModusToolbox flow used. FlowVersion 1 was used for ModusToolbox 2.0 and 2.1. Since ModusToolbox 2.2 FlowVersion 2 is used. While the MTBQueryAPI can detect FlowVersion 1 applications, it cannot perform any operations on these projects and any information returned via AppInfo or ProjectInfo is not guarenteed to be valid.
AppStructure	The MTBQueryAPIs support various versions of ModusToolbox applications
SharedDirectory	This is a directory where assets may be stored so that the contents of the asset can be shared by multiple applications. This directory is generally parallel to a set of application directories and is named mtb_shared.
AppSearchPath	A set of directories that are part of the build process for a given project. This generally includes the project directory itself and any middleware libraries that are stored in the SharedDirectory.
AppIngorePath	A set of directories that are found via the AppSearchPath that should be ignored when doing any file searching within a project.

2. RESPONSIBILITIES

The ICW Software Organization is responsible for:

- Creating and maintaining this document for projects.
- Keeping this document current as the project progresses.

3. REFERENCED DOCUMENTS

3.1 00-00064 – Cypress Record Retention Policy

3.2 001-98379 – Universal Device Database

3.3 002-30759 – ModusToolbox 3.x Software Architecture Specification

3.4 69-00028 – Software Best Practices

4. MATERIALS: N/A

5. EQUIPMENT: N/A

6. SAFETY: N/A

7. CRITICAL REQUIREMENTS SUMMARY

7.1 MTBQueryAPIs

The MTBQueryAPIs (also referred to as the QueryAPI) are intended to move the knowledge of ModusToolbox applications, manifests, tools, and device support files from Makefiles and bash scripts to a C++ implementation. One goal is to reduce the implementation in Makefiles to the activities that are typical for make, namely build activities. A further goal of the MTBQueryAPIs is to provide a uniform view of manifests, applications, tools, and device support files to the various C++ clients that are part of ModusToolbox, including the make based build system, project-creator, library-manager, BSPAssistant, and the various configurators.

7.1.1 Usage

The public interface for the MTBQueryAPI is a set of classes. The MTBQueryAPI classes are in the infineon::mtb::mtbqueryapi namespace. The header files for the classes that are considered public are stored in the top level include directory. When these APIs are distributed within Infineon, the complete source will be made available for debug purposes, but it is required that only the classes in the top level include directory are used by clients.

The MTBQueryAPIs include a set of read only data models exposing information about ModusToolbox and, if desired, a specific application. These data models create a unified view of the ModusToolbox environment that incorporates the online manifest database, any user specified manifest database entries, ModusToolbox packs, installed tool versions, and discovered external tools.

The MTBQueryAPIs also include classes that can create a project (Application Creator), change a project (Project Change Manager), or change a BSP (BSP Data Model).

7.1.2 Read Only Data Models

7.1.2.1 Tools Data

The MTBQueryAPIs must provide information about the tools discovered as part of ModusToolbox. This information is provided in the form of a tool database (***IToolDb***) object. The tool database can be used to query a tool by name or by id. The tool database identifies a subset of the tools discovered as configurators. Both the list of tools (including configurators) and the list of configurators can be provided.

This is described in detail in section 8.1.13.

7.1.2.2 Manifest Data

The manifest data is contained in a set of XML files. At the top level there are a set of super-manifest files. These super-manifest files contain references to manifest files. The manifest files contain data about board support packages (BSPs), applications, and middleware. Each manifest file will contain only data about one of these three types of items. Each of these manifest files contains data structures describing the attributes of a set of assets that are available to ModusToolbox. Each of these XML data structures has a corresponding data structure in the MTBQueryAPI.

Dependency files can be associated with each manifest file. Dependency files provide information about whether any application or middleware item is dependent on any BSP or other middleware. The dependencies are declared at the version level between items. They are used to ensure that all compatible versions of BSPs, applications, and libraries are matched up and that middleware required for a given application is present in the application.

The manifest database is described in detail in section 8.1.15

7.1.2.3 Device Support Files

The universal device database (also known as the DeviceDb) is described in the specification 001-98379 – Universal Device Database. Also, some assets (e.g., many versions of PDL middleware) provide crucial information about devices. There are many different situations that require the use of a DeviceDb and device support files. The MTBQueryAPI provides APIs that help clients locate the correct versions of these files.

Support for device support files is decribed in detail in section 8.1.27.

7.1.2.4 Application Data

The application data is information about the application and projects contained in the application. Information available includes important directories, direct dependencies, target BSP, and much more.

The application data is described in section 8.1.16.

7.1.2.5 ModusToolbox Packs

ModusToolbox packs are described in in detail the SAS. This data model provides information about the ModusToolbox packs that have been loaded by the MTBQueryAPIs. The data loaded from ModusToolbox Packs is fully integrated into the other data models provided by the MTBQueryAPIs. In general, clients should not need to know if specific data comes from a ModusToolbox pack rather than from some other source. However, this information is made available through the API for debugging or to provide information to the user about packs that are loaded.

ModusToolbox packs are made available to the user as either Technology Packs or Early Access Packs. Both of these use cases are a set of conventions at the business process level for developing and deploying these packs. Within the MTBQueryAPIs the only distinction between these two types of packs is how they are enabled. In all other respects they are treated the same.

The ModusToolbox Packs data model is described in detail in section 8.1.17.

7.1.3 Read/Write Data Models

7.1.3.1 ApplicationCreator

The application creator is the backend for the project-creator. This object allows for creation of applications using manifest-based Apps (i.e., code examples) and BSPs as well as custom Apps and BSPs. It supports all combinations.

This class is described more in detail in section 8.1.21.

7.1.3.2 ProjectChangeMgr

The project change manager provides capabilities to alter the middleware in a project, to alter the BSPs in a project, or change the target BSP for a project. This is the only way to modify projects in the MTBQueryAPI.

This class is described in more detail in section 8.1.22.

7.1.3.3 BSP Data Model

The BSP Data Model is a read/write data model that provides information about a standalone BSP. This data model can also be used to make changes to a BSP and write those changes out to the BSP on disk.

The BSP Data Model is described in more detail in section 8.1.23.

7.2 Executables

The MTBQueryAPI project provides executables that are used to integrate the MTBQueryAPI data with the ModusToolbox build environment.

7.2.1 mtbsearch

This executable implements the search for source files that are part of a given project. It follows the ModusToolbox conventions for shared assets and ignored content. The tool generates two types of data: a list of files to use in a build, a list of include directories.

This executable is described in more detail in section 8.2.1.

7.2.2 mtbquery

This executable provides a general-purpose query tool to provide information about the ModusToolbox installation and a given application or project. Most of the output modes for this program are intended to be human readable but there are a few that are meant to be consumed by the ModusToolbox make system.

This executable is described in more detail in section 8.2.3.

7.2.3 mtbgetlibs

This executable provides 100% of the functionality of the “make getlibs” target in ModusToolbox. It is essentially a command line wrapper for ProjectChangeMgr::commit().

This executable is described in more detail in section 0.

7.2.4 mtblaunch

This executable provides a method to launch configurators from the build environment. It also provides information to support quick launch panels in support IDEs.

This executable is described in more detail in section 8.2.4.

7.2.5 lcs-manager-cli

This executable is used by users to create and manage their local content.

This executable is described in more detail in section 8.2.5.

8. DESIGN DETAILS

8.1 MTBQueryAPIs

8.1.1 Overview

The MTBQueryAPI contains five major subsystem that load and store application information, manifest information, tool information, BSP data, and ModusToolbox pack information.

It is assumed that all of the MTBQueryAPIs are called from the same thread or are protected from being called from multiple threads by the client. The MTBQueryAPIs are NOT thread safe.

8.1.2 IModusToolboxEnv

The MTBQueryAPIs make information available via a top level ***IModusToolboxEnv*** object. This object is an interface to an underlying object that is not exposed via the MTBQueryAPIs. This is very similar to the COM model used by Microsoft Windows and provides the ability to upgrade the APIs in the future while retaining compatibility.

The ***IModusToolboxEnv*** object is obtained by calling the static method *getModusToolboxEnvironment* and it is released by calling the static method *freeModusToolboxEnvironment*. This approach was taken to allow for extension in the future when a new interface could be created by deriving from ***IModusToolboxEnv***. This new interface would add additional methods to the API. The ***ModusToolboxEnv*** class would be changed to be derived from the new class and would implement the new methods. In this way, old code would work as is, but new code could cast the ***IModusToolboxEnv*** interface to the new interface and have access to the new APIs.

All information about the ModusToolbox environment is obtained by querying the ***IModusToolboxEnv*** object via its various methods.

8.1.3 ModusToolboxEnv

The ***ModusToolboxEnv*** object is derived from the ***IModusToolboxEnv*** object and contains the implementation of the APIs.

8.1.4 Operating Mode

The MTBQueryAPI operates in one of two modes. The mode is set via the ***IModusToolboxEnv::setOperatingMode*** API.

Direct – requires access to the internet and all fulfillment of asset requests are always taken directly from servers on the internet.

Local – content from the LCS is used.

The mode is set by the user using one of the GUI tools (in the settings dialog) or by setting the MTB_USE_LOCAL_CONTENT environment variable to true.

8.1.5 Background Operations

Some of the operations performed by the MTBQueryAPIs are background operations. These are operations that may request download of data from the internet or make calls to external applications like “make” or “git”. Some of these operations are:

- ***IModusToolboxEnv::loadEnvironmentData***
- ***ProjectChangeMgr::commit***
- ***ApplicationCreator::createApplication***.

These operations are requested via methods that return almost immediately but complete in the background and emit a signal when the operation is finished. These operations are not running on a separate thread, but rather take advantage of background activities that can be executed in the Qt event loop.

There may be Qt modules that use additional threads to complete their tasks but the MTBQueryAPIs run within the thread from which they are called and all signals are emitted on the thread that requested the operation.

For background operations to complete, the Qt event loop must be running. This means that there must be periodic calls to the ***QCoreApplication::processEvents***. If this is a GUI application, the main GUI event loop performs this task.

Each of these operations that execute in the background have arguments to force the operation to be synchronous to the call requesting the operation. In this case, there is an internal loop that calls ***QCoreApplication::processEvents*** until the operation is complete.

As stated above, the background actions are not happening on a separate thread. It is important to understand the Qt event model and to realize that for a synchronous operation (one of the APIs called with the synchronous flag set to true), the handlers connected to the signals will be executed BEFORE the API call returns.

8.1.5.1 Cancelable operations

The following public operations are cancelable:

- ***IModusToolboxEnv::loadEnvironmentData*** (APP_INFO, MANIFEST_DATA, ASSET_REFS only)
- ***ProjectChangeMgr::commit***
- ***IApplicationCreator::createApplication***

The following internal operations are cancelable:

- ***AppEnvironment::init***
- ***AppLoader::load***
- ***AppSource::getFromAppSrc***
- ***AssetMgr::fetchAsset***
- ***AssetJob***
- ***AssetRefLoader::loadReferences***
- ***DownloadManager::download***
- ***GitJob***
- ***ManifestLoader::load***

Cancelable operations accept a ***MtbCancelToken*** object. Each ***MtbCancelToken*** is connected to a ***MtbCancelSource***. The client creates a ***MtbCancelSource*** and passes the corresponding ***MtbCancelToken*** to the cancelable function. The client may use the ***MtbCancelSource*** to request cancellation. Cancelable functions may use the ***MtbCancelToken*** to poll for cancellation requests or to connect a Qt signal to receive notifications of cancellation requests.

MtbCancelToken is not copyable but it is intended to be passed using shared_ptr. Cancelable operations may pass the pointer to sub-operations. For example, ***AssetMgr*** passes the cancellation token to ***AssetDirectJob***.

Cancelable operations should report canceled operations using the same mechanism(s) they use to report successful or unsuccessful completion. For example, ***DownloadManager*** reports canceled downloads using the existing ***downloadError*** signal.

Requesting cancellation does not guarantee that the operation will be canceled. The response to cancellation requests depends on the implementor. For example, a cancellation request might not be effective if the implementation does not support cancellation or if the operation is already finished.

8.1.5.2 Canceling subprocesses

The ***Runner*** class provides a way to cancel subprocesses but the details are platform-dependent.

The API client should set the cancellation mode using ***IModusToolboxEnv***:: ***setSubprocessCancellationMode***.

- CLI mode: The query API does not directly pass cancellation requests to subprocesses. Instead, it allows signals (SIGINT/SIGTERM on Unix or CTRL_C_EVENT/CTRL_BREAK_EVENT on Windows) to propagate normally to subprocesses. This mode is recommended for CLI applications.

- GUI mode: The query API assigns subprocesses to a separate process group. This means that signals sent to the main application process will not propagate to subprocesses. Cancellation requests will cause the Runner to send SIGINT or CTRL_BREAK_EVENT to the subprocess group. This mode is recommended for GUI applications.

8.1.6 Error Reporting

Most operations performed by the MTBQueryAPI can produce a variety of warning and error messages. When The ***IModusToolboxEnv*** object is created, it creates an internal ***QueryAPIErrorList*** object As warnings and errors are encountered, ***QueryAPIError*** objects are created and added to this error list.

Clients can access this error list by calling a method on ***IModusToolboxEnv***. Clients can also call a method that will clear the internal error list.

When the ***IModusToolboxEnv*** object is being created, it is possible for errors to occur. Since the object isn't yet created, it's not possible to register errors on the internal error object (since it doesn't exist). To account for this, the call to ***IModusToolboxEnv::getModusToolboxEnvironment()*** accepts a ***QueryAPIErrorList*** argument. This is a temporary object used only for holding errors generated during the object creation. The ***IModusToolboxEnv*** object does not hold onto this error list.

8.1.7 Message Signals

Since several of the core operations of the QueryAPIs are performed as background operations, care has to be taken in how messages are sent to the client.

The QueryAPIs provide the following signals to allow clients to monitor and report on QueryAPI operations.

- ***streamStdout*** – this signal streams the stdout data from underlying processes.
- ***streamStderr*** – this signal streams the stderr data from underlying processes.
- ***startingGitCommand*** – this signal is emitted every time a git job is started. Note that not all operations require git commands.
- ***finishedGitCommand*** – this signal is emitted every time a git job is finished. Note that not all operations require git commands.

QueryAPI operations are done in a hierarchical manner. For example, reading the manifest database requires reading the super-manifest file. Then for each manifest from that list, download the file and parse it. While parsing the whole file we are parsing individual groups and individual items and even individual XML elements. The following signals provide a "level" that roughly corresponds to the hierarchical level of processing that is happening when the signal is emitted. The higher the level, the lower into the hierarchical layers the signal is coming from. This allows clients to filter the level of detail that they want to show to their users.

- ***startStep*** – this signal is emitted every time a new operation is started. For example, "Acquiring code example" (during application

creation) or “Exporting middleware content” (during the ***ProjectChangeMgr::commit*** operation). This is meant to match a corresponding ***finishStep*** signal.

- ***finishStep*** – this signal is emitted every time an operation completes. The ***finishStep*** signal includes an argument that indicates if the operation that just finished was successful or not. This is meant to match a corresponding ***startStep*** signal.

- ***verboseOutput*** – this signal is emitted to provide more detail about what is happening during an operation. Frequently this output is at the same level, or higher, as the enclosing ***startStep*** and ***finishStep***.

8.1.8 Logging

The MTBQueryAPIs can output debug information into a log file. This information is intended to be consumed by the internal development teams and not the customer. It is useful, when debugging customer problems, to send the generated log files to Infineon.

The MTB_QUERY_API_LOG_FILE environment variable enables the generation of a log file. This value must be set to a valid path. If the file does not yet exist, it will be created. If the file does exist, new log messages will be appended to it. New log messages will **always** be appended to this file. It will never be deleted by the MTBQueryAPIs. If a user leaves this variable set, the log file can grow to a significant size.

The MTB_QUERY_API_LOG_LEVEL environment variable specifies the level of detail for the log file. The higher the level the more information is provided in the logfile.

8.1.9 MTBQueryAPI Initialization Data

Some clients using the MTBQueryAPIs ask the API to load application data. This requires that the API obtains certain information about the application or project and that information is contained in the Makefiles. The exact data that is required is documented in the SAS based on specific use cases. This data is made available using one of two methods.

The first method is for the MTBQueryAPI to call “make CY_PROTOCOL=2 MTB_QUERY=1 get_app_info”. The required data is output from “make” and processed by the MTBQueryAPIs. After this data is collected, it is written into a cache file ${PROJECT_DIR}/.mtbqueryapi. Subsequent calls

to “make” to get the application info will use this cache file as long as it is up to date.

The cache file becomes out of date if any of the Makefiles used in the project have a timestamp more recent than the timestamp of the cache file. Also, if the path to the cache file has changed (for example, because of a copy), then the cache is considered out of date.

Due to the way that make works, it is possible for users to assign values to make variables on the command line and these assigned values override what is in the Makefiles. In these situations the timestamps on the Makefiles hasn’t change but the data in the .mtbqueryapi cache is incorrect (i.e., it doesn’t match what make is seeing because of the user assigned value). To handle this, the make system will detect command line variables. There are a few command line variables that are routinely set but do not affect the cached data (e.g., CY_PROTOCOL). If any variables, except those on a whitelist, are set on the command line, the make system will set the environment variable MTB__GAI_CACHE_INVALID to a non-empty value. Inside the QueryAPI, this environment variable is checked. If it is set to a non-empty value, then the get_app_info cache is ignored entirely. It is not read, it is not modified, and it is not deleted.

Variables are on the whitelist either because they are used by our IDEs or because they are well known utility variables that have no impact on the get_app_info data.

IDE Variables:

Variable	**IDE(s)**	**Details**
CY_MAKE_IDE	Eclipse	Lets the build system know it's being built from Eclipse so it can adjust what it prints to STDOUT in order to support IntelliSense.
CY_IDE_TOOLS_DIR	Eclipse	It relays what the IDE thinks the CY_TOOLS_DIR is, but it turns out this is NOT USED by

		any part of our make system.
CY_IDE_BT_TOOLS_DIR	Eclipse	It relays what the IDE thinks the BT tools suppliental tools live, but it turns out this is NOT USED by any part of our make system.
MTB__JOB_BACKGROUND	Visual Studio Code	By default this expands to & (ie, to make a command run in the background). This is override by Visual Studio Code to force tools to run in the foreground (eg, device-configurator).
CY_IDE_PRJNAME	Eclipse	Used to relay project name (eg, because the user renames it) to the "make eclipse" process. It is not used by the "make build" process.

Utility variables:

Variable	**Details**
VERBOSE	Used to show verbose command-invocations during the build process.

CY_SECONDSTAGE	Used by make to indicate a recursive call after code generation
MTB_APPLICATION_SUBPROJECTS	Set by application.mk. All the project in the application
MTB_APPLICATION_NAME	Set by application.mk. The name of the application
MTB_APPLICATION_PROMOTE	Set by application.mk. Whether the project should copy its hex file to the application directory to generate combined hex.

The second method avoids calling make. With this approach a “response” file is generated that contains the required data. The response file is provided to the client using the MTBQueryAPIs through a command line argument. This data is processed as though it was returned from the get_app_info called described above. This method is generally used when calling one of the MTBQueryAPI executables from make. This avoids a recursive call back into make.

In addition to the name/value pairs described in the SAS, there are name/value pairs that can be used for testing. These are described in the table below.

Variable	**Purpose**
TEST_GIT_JOB_MGR_FAIL_COUNT	Will force the git job manager to fail a git job after the given number of git jobs have been launched.
TEST_IDC_ALL_DIR	The location to look for Infineon Developer Center installed content for all users.
TEST_IDC_USER_DIR	The location to look for Infineon Developer Center installed content for the current user.
TEST_TOOLS_DIR	The tools directory for the MTBQueryAPI to use. This will override any internal mechanism MTBQueryAPI has to detect tools directories.
TEST_APP_DIR	The directory to use for the current client application directory. Used to test the mode where the tools directory is discovered by going from the current client location up the path until a tools directory is found.
TEST_BYPASS_TOOLS_CHECKS	By default, if the MTBQueryAPIs find a tools directory and the tools directory does not

Variable	Purpose
	contain a modus-shell package with at least “make”, “git”, and “bash” then an error is returned. There are some test cases where this is not required and this value, if set to TRUE, will bypass these checks.

8.1.10 Deducing A Directory Type

If application data is loaded by the MTBQueryAPI, a directory is provided where the application data is located. For MTB2X projects, this directory is the combined application and project directory. For MTB3X projects, this directory could be the application directory or the project directory.

If MTB_TYPE is defined, this is an MTB3X application or project. The MTB_TYPE will either be APPLICATION, PROJECT, or COMBINED. In this case, the application type is AppStructure::MTB3PLUS.

If MTB_TYPE is not defined, this is a project from ModusToolbox 2.x. The tool-make does not output data for COMPONENTS, DISABLE_COMPONENTS. In addition, the TARGET_DEVICE is supplied but is always empty. In this case, the application type is AppStructure::MTB2X.

8.1.11 ModusToolbox Specific Use Cases for Make

This table lists the specific MTBQueryAPI use cases, and what is expected from the ModusToolbox make system.

Case	Directory	Core Make	get_app_info Output	MTB_TYPE In Makefile	AppStructure	Notes
1	App + Project Combined (e.g. MTB 2x app)	None	MTB_TYPE undefined MTB_TOOLS_MAKE_VAR=3.0	N	MTB2X	ModusToolbox 2.x code example being used as is in ModusToolbox 3.0+.
2	App Dir	None	MTB_TYPE=APPLICATION MTB_TOOLS_MAKE_VAR=3.0	Y	MTB3PLUS	ModusToolbox 3.0+ multi-core code example, app directory
3	Project Dir	None	MTB_TYPE=PROJECT MTB_TOOLS_MAKE_VAR =3.0	Y	MTB3PLUS	ModusToolbox 3.0+ multi-core code example, project directory
4	App + Project Combined (e.g. MTB 2x app)	< 3.0	MTB_TYPE undefined MTB_TOOLS_MAKE_VAR undefined	N	MTB2X	ModusToolbox 2.x code example being used as is in ModusToolbox 3.0+.
5	App + Project Combined	>= 3.0	MTB_TYPE=COMBINED MTB_TOOLS_MAKE_VAR undefined	Y	MTB3PLUS	ModusToolbox 3.0+ code example with application and project in a single directory.

	(e.g. MTB 3x app single core)					
6	App Dir	>= 3.0	MTB_TYPE=APPLICATION MTB_TOOLS_MAKE_VAR undefined	Y	MTB3PLUS	ModusToolbox 3.0+ multi-core code example, app directory
7	Project Dir	>= 3.0	MTB_TYPE=PROJECT MTB_TOOLS_MAKE_VAR undefined	Y	MTB3PLUS	ModusToolbox 3.0+ multi-core code example, project directory

8.1.12 TOOLSDIR

The TOOLSDIR is the directory that contains the standard set of tools that are installed as the tools package for ModusToolbox. The MTBQueryAPI requires that the modus-shell package be installed and available to function correctly.

The process of finding the correct tools directory depends on how the MTBQueryAPI object was created, what arguments were provided during creation, the location of the executable that is using the MTBQueryAPI, the presence and value of certain environment variables, and the presence of tools in the common, well known location. The algorithm checks each of the following conditions in order. Once one of the conditions matches, that is used to set the TOOLSDIR.

1. If the TEST_TOOLS_DIR command line option is set, use that value.

2. If the MTB_TOOLS_DIR command line option is set, use that value. Normally, this should only be used when invoking mtbquery from ‘make’ to break a dependency cycle of CY_TOOLS_PATHS and get_app_info.

3. If the environment was created with an application, use the tools directory specified in the application. This is obtained by running ***get_app_info***. Emit a warning if the application tools directory differs from the one from where the executable was run.

4. If the executable is located in a tools directory tree then use that location as the TOOLSDIR. Emit a warning if the executable tool directory differs from the value obtained from the CY_TOOLS_PATHS variable.

5. If the CY_TOOLS_PATHS variable is set and one of the directories on that variable exists, use that directory. If the CY_TOOLS_PATHS

variable is set but none of the directories exist, issue an error. In this case the tools directory remains unset.

6. Check the common, well-known location and if it exists, use that location for the TOOLSDIR (for example ~/ModusToolbox/tools_3.0).

If none of those conditions are met then the MTBQueryAPI is unable to find an appropriate tools directory. In this case, an error is generated and the tools directory remains unset.

8.1.13 Tools

Tools are discovered by searching a set of directories for sub-directories that meet a specific pattern. The set of directories searched is the TOOLSDIR as defined in section 8.1.12, ModusToolbox pack tools directories (section 8.1.17), and any external tool directories (section 8.1.13.3).

The tools discovery process is managed by the ***ToolsScanner*** object.

For a directory to be identified as a tool, the directory must have a props.json file with a valid “core.version” entry.

The props.json file is how tools can provide information to the MTBQueryAPIs. The contents of this file are defined in the SAS, but one specific property that must exist is the core.id property which should be a UUID. An example props.json file is shown below. The special variables in this example (e.g. $$FINDFILE) are processed by the mtbquery executable and are described in section 8.2.3.

```
{
  "core": {
    "id": "fde6ff8c-2f4e-4456-bf4a-abe39c98ff44",
    "name": "device-configurator",
    “version”: “1.1.0.1”
  },
  "opt": {
    "open": {
      "device_configurator_EXT": "modus",
      "device_configurator_FILE": "$$FINDFILE:design\\.modus$$",
      "device_configurator_TOOL": "$$FULLPATH:device-configurator$$",
      "device_configurator_TOOL_ARGS": "--library $$DEVICE_SUPPORT_LIB$$",
      "device_configurator_TOOL_FLAGS": "--design $$FINDFILE:design\\.modus$$ --check-mcu=$$DEVICE$$ --check-
coprocessors=$$ADDITIONAL_DEVICES:,$$",
      "device_configutator_TOOL_NEWCFG_FLAGS": "--library $$DEVICE_SUPPORT_LIB$$ --design
$$PROJECT_DIR$$/design.modus"
    },
    "tool" : {
      "device-configurator-cli_EXE":"$$FULLPATH:device-configurator-cli$$",
      "device-configurator_BASE": "$$TOOL_DIR$$",
      "device-configurator_EXE": "$$FULLPATH:device-configurator$$"
    }
  }
}
```

In addition, if the directory contains a valid configurator.xml file, this tool considered a configurator. The contents of the configurator.xml file provide information about the context for launching this tool as a configurator. The content of the configurator.xml file is defined in the SAS.

8.1.13.1 Loading the tools

The tools database is created by loading tools from the base installation, packs, and external directories. This processed is started when the user calls loadEnvironmentData() with the TOOLS load flag.

The loading process first tries the base installation. If there are errors during this part then the load operation aborts and no other tools are loaded. In this case, the loadEnvironmentData() method emits a loadDataError signal and stops doing any more environment loading.

If there are no errors while loading the base installation tools, then the process continues by loading tools from packs and then external locations. If there are any errors encountered while loading packs or external tools, those errors are added to an error list but the process is allowed to continue. In this case, the loadEnvironmentData() process continues as normal.

8.1.13.2 Program and ProgramRunner

Each tool can contain one or more programs. A program is an executable that can be run by the user or by some other part of the ModusToolbox environment.

When client code wants to run a program, the client must first obtain a ***ProgramRunner*** object from the ***Program*** object using one of the ***getProgramRunner*** overloaded methods. This method creates an object that is capable of running the program with the correct arguments. The correct arguments are created by using the data defined in the props.json file associated with the ***Tool*** object that contains the ***Program***.

The ***ProgramRunner*** class is essentially a wrapper for the ***Runner*** class that has hidden the ***Runner::start*** method. This method is hidden so that the ***ProgramRunner*** has full control over the command and arguments sent to ***Runner***.

ProgramRunner has three StartModes:

- SYNC: The start method does not return until the subprocess exits. Client may use signals to receive notifications.

- ASYNC: The start method returns immediately. Client may use signals to receive notifications.
- DETACHED: The start method returns immediately. The ProgramRunner will not emit signals.

8.1.13.3 IDC Tool Discovery (“external tools”)

Non-MTB tools can advertise themselves to MTB and then be included in the MTB tools database. This only applies to tools that are distributed via IDC.

When scanning for tools, the QueryAPI scans all the JSON files in the IDC installation directories and usesthe information in the IDC JSON file to find the root of the tool. Next, it looks for one or more JSON files from that root. If any exist then the tool is included in the tools database.

8.1.13.3.1 Finding the root

There is no specific field in the IDC JSON file that indicates the root of a tool but there are several fields that indicate paths to other things. The QueryAPI looks at the "path" and "exePath" fields to find the common ancestor for all those and then assumes that this is the root.

8.1.13.3.2 Finding JSON files

The QueryAPI looks in the root for either a props.json or mtbprops.json file in the root.

If mtbprops.json exists then it processes that file to get a list of additional props.json files to process. In this case, the QueryAPI ignores the props.json file at the root (unless it is listed in the mtbprops.json file). The mtbprops.json file is a file that lists other props.json files that are tools. The format is a simple JSON document as follows:

```
{
  "prop_files": [“./dir1/props.json”, “./dir2/props.json”]
}
```

Each entry in the array is a relative path (relative to the directory containing the mtbprops.json file) to a props.json file. This file is expected to describe a tool.

If the mtbprops.json file does not exist at the root then the QueryAPI looks for a props.json at the root. If that file exists then it is expected to describe a tool.

Note that if a mtbprops.json exists and it contains an empty array of prop_files, we will process no props.json files. This mechanism allows an IDC package to have a non-tools props.json file at the root without generating an error (but this seems like a strange case).

8.1.13.3.3 Parsing the props.json file(s)

For each props.json file (either the ONE at the root, or all of the files identified in mtbprops.json), the QueryAPI uses the standard parsing algorithm as is used for the tools distributed with MTB.

8.1.13.3.4 Multiple executables for a tool

A "tool" in MTB is not a "program". Instead, each "tool" contains zero or more "programs". This same mechanism applies to external tools. Any external tool can specify multiple programs in the single props.json.

8.1.13.4 Tool Properties

Properties are simple name=value pairs associated with a tool or middleware. There are three types of properties – core properties are required (type, location, and version), optional properties have well-defined meanings but may or may not be present, and custom properties have a tool or middleware specific meaning.

- Property-aware versions of Infineon packaged tool and middleware assets will contain a text file in their base directory called props.json.
- Most properies are specified explicitly in the appropriate properties file.
- Some properties come from pre-existing sources (e.g.,.mtb files).
- Critical (“core”, details below) properties for middleware have fallback sources so that ModusToolbox 3.X can consume already shipped middleware.

All tools are required to be property-aware. That is, all tools that ship as part of ModusToolbox 3.X, tools provided via a ModusToolbox pack, or tools provided as external tools are required to support properties as described below.

8.1.13.4.1 Core Properties

Core properties are required by all tool and middleware assets. Core properties start with the prefix “core.” E.g., core.name and core.version.

8.1.13.4.1.1 core.id

A unique immutable identifer of the tool or middleware. It MUST NOT change over time / across versions. For tools and 3.X-aware middleware this must be a UUID. For BWC with already released middleware it may be the “shortname” of the repository. This property MUST appear in all properties files.

Legacy or 3rd party middleware: The URL field of the .mtb file will be used extract the id.

```
{
    “core”: {
        “id”: “value”
    }
}
```

8.1.13.4.1.2 core.name

A user-visible name of the tool or middleware. This property MUST appear in all properties files.

Legacy or 3rd party middleware: The git repo name will be extracted from the URL field of the .mtb file.

```
{
    “core”: {
        “name”: “value”
    }
}
```

8.1.13.4.1.3 core.version

Version of the tool or middleware. This property MUST appear in all tool properties files.

For middleware, this is backed by the middeware’s version.xml file.

```
{
    “core”: {
        // standard ModusToolbox Major.Minor.Patch.Build numbering required.
        “version”: “A.B.C.D”
    }
}
```

NOTE: Version may be empty in the case of 3rd party middleware.

8.1.13.4.1.4 core.schema-version

An integer that indicates the “schema” version of the props.json file. For ModusToolbox 3.X this must be “1”. Changes to the props.json format in the future may require the introduction of new schema versions. This property should appear in all propery files and MUST appear if not “1”.

Schemas for the props.json files are stored in github. For tools, the schema is here. For libraries the schema is here.

```
{
    “core”: {
        // version of the props.json schema this file aligns to.
        “schema-version”: "1“
    }
}
```

8.1.13.4.2 Optional Properties

Properties that are purely optional and their inclusion is purely up to the asset itself. These options are expected to grow over time as new items are found to be useful. Consult the schema’s listed above for a complete set of available properties. A couple of the most relevant items are described below.

8.1.13.4.2.1 opt.tool

This section describes the tool generally (ie, not a specific program / executable within the tool folder). This entry is only relevant for tools properties files.

8.1.13.4.2.1.1 opt.tool.make-vars

Define make variables that need to be generated by the tool. This is almost always just the tools base directory.

```
“opt”: {
     “tool”: {
          // CY_TOOL_ make variables that apply generally to the tool (not program-specific).
          // This is typcially just the “_BASE” variable, though some (like OpenOCD) have more.
          “make-vars”: {
               “openocd_BASE”: “openocd”
               “openocd_scripts_SCRIPT”: “openocd/scripts”,
          }
     }
}
```

8.1.13.4.2.1.2 opt.tool.make-vars.*

A set of string key / value pairs. The key is a string that defines the name of the make variable that should be generated. The mtbquery executable will automatically prefix the key with “CY_TOOL_” when generating the file for make.

The value is an arbitrary string intended to be used by make. It has access to the standard variety of props.json magic macros (e.g., $$TOOL_DIR$$, $$FINDFILE:glob$$).

8.1.13.4.2.2 opt.programs

An optional part of props.json for tools that describes programs (executables) that are distributed as part of the tool. It is not used or valid for middleware props.json files. These are important to the invocation of programs, code generation, and are primarily used by mtbquery, mtbserch, and mtblaunch.

```
“opt”: {
     “tool”: {
          // CY_TOOL_ make variables that apply generally to the tool (not program-specific).
          // This is typcially just the “_BASE” variable, though some (like OpenOCD) have more.
          “make-vars”: {
               “openocd_BASE”: “openocd”
               “openocd_scripts_SCRIPT”: “openocd/scripts”,
          }
     },

     // self-contained set of programs owned by this tool.
     “programs”: [
     {
          // Required, a globally unique and stable id (GUID) that uniquely identifies this configurator.
          // Correlated against the (new) MTB_DEVICE_PROGRAM_IDS in the get_app_info so the query
          // APIs know which are valid.
          “id”: “xyzzy”,

          // Required, a globally unique short name that is used in human-readable contexts
          // (e.g., BSP makefiles) that identify a program. For MTB 2.x tools this name MUST
          // come from your CY_OPEN_<shortname>_ variables. For tools new to MTB 3.0 and later
          // choose a unique but human readable short name. This name MUST be a legal identifier.
          “short-name”: “your-name-here”,

          // Required, path relative to the tool’s base directory of the executable to run from
          // when requested by the user (e.g., via the Quick Launch Panel or via “make <target>”.
          “exe”: “path/to/program.exe”,

          // Optional, path relative to tool base directory to a PNG icon that represents the
          // program.
          “icon”: “xyzzy.png”,

          // Optional, name as presented to the user in GUIs. Defaults to name of executble with
          // the .exe suffix removed.
          “display-name”: “Xyzzy Editor”,

          // Optional, extension(s) for which this program is the preferred viewer/editor
          // e.g., device configurator for design.modus.
          “priority-extensions”: [ “.modus” ],

          // Optional, extensions of files supported by this configurator. E.g., used by programs that
          // support the given file type but are not the default viewer/editor.
          “extensions”: [ “.xyz” ],

          // Optional, args used when opening a file with one of the above extensions.
          // This field can use macros like $$FINDFILE$$.
          “open-file”: “<open file args>”,

          // Optional, args for creating a new file [e.g., bt, usb, capsense]
          // This field can use macros like $$FINDFILE$$.
          “new-file”: “<new file args>”,

          // Optional, args used to retrieve external memory data from a configurator
          // This field can use the reserved names $$CONFIG_FILE$$ and $$OUTPUT_FILE$$
          “export-memory”: “<export memory args>”

          // Required, must be either “bsp”, “project”, or “all”. E.g., device-configurator is a BSP program
          // because its source file lives in BSPs, or bt whose source file lives in a specific project is a
          // project type. The all type is for a tool program that is always applicable (for example, the
          // library-manager)
          “type”: “<bsp or project or all>”,

          // Optional, like opt.tool.make-vars – a collection of executable-specific variables to generate
          // for make, also in key-value format. This is generally to provide the path to the executable.
          //
          // “tool-name_EXE” : “$$TOOLPATH:exe-name:relative$$”,
          // “tool-name_EXE_ABS” : “$$TOOLPATH:exe-name:absolute$$”,
          “make-targets”: [ “xyzzy” ],
```

```
        // Optional, when mtbgetlibs detects a core-make 3.x based application it generates
        // targets into mtb.mk to provide CLI access to configurators. E.g., these provide
        // a data-driven replacement for the old "make config" style targets in core-make 3.0.
        //
        // targets look like the following:
        // xyzzy:
        //        <path to program.exe> <open-args>
        //
        // multiple targets are supported to enable BWC (e.g., modlibs / libmgr)
        "prj-make-targets": [ "xyzzy" ],

        // app make targets are for things like "config", which runs the device configurator.
        // These are generally BSP configurators. The implementation of the targets will
        // recursively invoke this target on the first project in the project list. It provides
        // the context necessary to perform autodiscovery / find required source files.
        "app-make-targets": [ "xyzzy" ],

        // Optional, if the program needs to generate code at build-time...
        "code-gen": [{
             // If mtbsearch runs all code generation immediately before autodiscovery. Sources
             // are the inputs, targets are the output file(s) used to detect stale generated source.
             // They should be a file or files that are always written. Can be a simple timestamp
             // file that doesn't particpate in autodiscovery, args are any arguments that need to be
             // passed to the code generation process.
             // These three fields can use macros like $$FINDFILE$$.
             "name": "<display name of generator>",
             "sources": [ "source file1", … ],
             "targets": [ "$$FINDDIR:design.xyz$$/GeneratedSource/timestamp.txt", … ],
             "args": "<code generation args>",
             "passes": ["default"]
        }],

        // The "compat" section is the old direct children of "opt". They're still needed
        // for BWC with projects based on core-make 1.x. All generated variables take the form:
        // CY_OPEN_<legacy-id>_<suffix>.
        "compat": {
             // List of suffixes to generate in makefiles.
             // All fields can use macros like $$FINDFILE$$ as necessary.
             "open": {
                  // E.g., Generates CY_OPEN_<legacy-id>_EXT
                  "EXT": ".xyz",

                  // value is dynamic – uses autodiscovery to find the file this to edits (QP launch)
                  "FILE": "$$FINDFILE:file.xyz$$",

                  // value is dynamic – path to tool executable to launch (from opt.paths).
                  "TOOL": "$$FULLPATH:foo$$",

                  // value is dynamic – path to files (found using autodiscovery)
                  "TOOL_ARGS": "",
                  "TOOL_FLAGS": "-x -z -z $$DEVICE_SUPPORT_LIB$$",
                  "TOOL_NEWCFG_FLAGS": "-x -z -z"
             } // end-open
        } // end-compat
    } // end this configurator
    ] // end list of configurators in this tool
}
```

GUI Programs (e.g., that participate in the QuickLaunch Panel) are required to have "open-file" arguments and/or "new-file" arguments.

Programs that participate in code generation are required to have a "code-gen" block.

- These may be dedicated CLI which would have a code-gen block and NO new-args or open-args.
- They may be GUI tools that have some combination of (new-file || open-file) && code-gen.

Programs that participate in the build, program, or debug processes (e.g., toolchains) are required to have a “make-vars” section.

“priority-extensions” is primarily for the Device Configurator, where there are multiple tools that work with design.modus but we need to know which has preference if the user double clicks one one.

- In general tools should use the vanilla “extensions”.
- If there is a conflict between two or more valid / relevant programs:
- Tool versioning handles Early Access packs. Latest tool’s programs will win.
    - If there are two tools with different tool IDs that both have programs with the same “priority-extension” we will pick a default and generate a non-fatal error.

“app-make-targets” is used to flag application level program targets (e.g., “make config”. In order to enable argument expansions like $$FINDFILE$$ these are simple trampolines. E.g.,

```
config:
        $(MAKE) -C $(firstword $(PROJECTS)) $@
```

These need to be processed by tools-make, as part of the same invocation of mtbquery –alltools.

Project level targets expand both project level targets from “prj-make-targets” AND the concrete application targets from “app-make-targets”. E.g., typing “make config” from a project does autodiscovery from that project’s point-of-view and runs device configurator.

8.1.13.4.2.3 opt.props

This value is an array of entries that describe properties. This value is optional in the props.json file. These paths are important to the invocation of the tool and are provided to applications like mtb-query to move into the make environment. Values may be simple scalars (apply to all platforms), or a map (supported on a subset of platforms or have different values on each platform).

```
{
    “opt”: {
        “props”: {
            “name1”: “value1”,
            “name2”: {
                “windows”: “value-win”,
                “linux”:   “value-lin”,
                “macos”:   “value-mac”
```

```
            },
            “name3”: {
                “windows”: “value-win”
            },
        }
    }
}
```

Tools: These properties are stored on the Tools object in the MTBQuery API and are available for use by applications.

Middleware: These properties are parsed and stored in the MTBQuery API and are available for use by applications.

8.1.13.4.2.4 opt.svd

An optional part of props.json for libraries that informs what svd file to use for specific devices. It is only relevant for device support libraries.

```
{
    “opt”: {
        “svd”: {
            "COMPONENT_PSOC6_01": "devices/COMPONENT_CAT1A/svd/psoc6_01.svd",
            "COMPONENT_PSOC6_02": "devices/COMPONENT_CAT1A/svd/psoc6_02.svd",
            "COMPONENT_PSOC6_03": "devices/COMPONENT_CAT1A/svd/psoc6_03.svd",
            "COMPONENT_PSOC6_04": "devices/COMPONENT_CAT1A/svd/psoc6_04.svd",
            "COMPONENT_TVIIBE1M": "devices/COMPONENT_CAT1A/svd/tviibe1m.svd",
            "COMPONENT_TVIIBE4M": "devices/COMPONENT_CAT1A/svd/tviibe4m.svd",
            "COMPONENT_CYW20829": "devices/COMPONENT_CAT1B/svd/cyw20829.svd",
            "COMPONENT_CAT1C4M": "devices/COMPONENT_CAT1C/svd/cat1c4m.svd",
            "COMPONENT_CAT1C8M": "devices/COMPONENT_CAT1C/svd/cat1c8m.svd"
        }
    }
}
```

8.1.13.4.2.5 opt.documentation

An optional part of props.json for libraries and tool programs that informs what documentation information it provides. See section 8.1.28 for individual attribute definitions, and information on declaring documentation for assets.

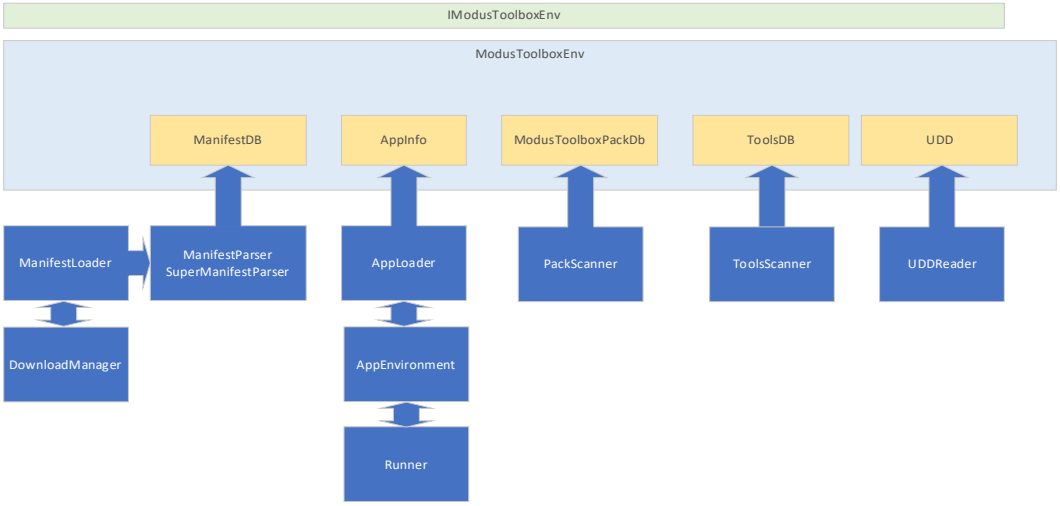
```
{
    “opt”: {
        “documentation”: [{
            // Optional list of filters for this document, such as project components
            "filters": {
                "components": []
            }
            // The type of documentation this provides (api-guide, readme, release)
            “type”: “api-guide”,
            // The name to show for this asset
            “title”: “PDL API Guide”,
            // The relative location in the asset to the documentation file
            “location”: “docs/pdl_api_reference_manual/index.html”,
            // An array defining the hieracy used to display this piece of documentation
            “path”: [ "Reference Guides”, “PSoC” ]
        }]
    }
}
```

8.1.13.4.2.6 opt.configurator_support

An optional part of props.json for libraries that informs tools that this library provides information that is relevant for configurators. This would be information such as device-db or personality files. Optionally, within configurator_support a license template file may specified. When present, this field specifies a file – which may include a path prefix, relative to the asset root directory. When present, this file is used in code generation to produce the software license text that gets inserted in a code comment at the top of every generated source file.

```
{
    // Whether or not this library provides contents (device-db or personalities) that are relevant
    // for device configurators
    “configurator_support”: {
        “enabled”: true,
        “codegen_license_template_file”: “device-info/codegen_license_template.txt”
}
```

8.1.14 Load Process



When ***IModusToolboxEnv::loadEnvironment*** is called, the data requested is loaded in a specific order: Application data, ModusToolbox pack data, tools data, asset references, manifest data, and UDD data. Some of these load operations are foreground operations and some of these are background operations. This is all managed by the ***ModusToolboxEnv::nextStep*** method.

After all requested data is loaded and there are no fatal errors detected, the ***dataReady*** signal is emitted.

If fatal errors were detected during the data loading process, the ***dataLoadError*** signal is emitted and the ***dataReady*** signal is not emitted.

There are some errors and warnings that are not considered fatal and, in these cases, the ***dataReady*** signal is emitted (and the errors have been added to the error list). Therefore, the following sequence must be followed by clients:

1. Register for the ***dataReady*** and ***dataLoadError*** signals.
2. Call the ***IModusToolboxEnv::loadEnvironmentData*** method.
3. If the ***dataLoadError*** signal is emitted, a fatal error occurred and the client should display an appropriate error message and exit.
4. If the ***dataReady*** signal is emitted, check the ***QueryAPIErrorList::getCount(ErrorType::ERROR)***. If that count is non-zero then check the ***LoadFlags*** argument of the ***dataReady*** signal to ensure the data you need is available or if you need to exit.
5. Decide how, or if, to display errors and warning found in the ***QueryAPIErrorList*** object.

The ***loadEnvironmentData*** method can be called multiple times. If the method is called requesting a new type of data that has not yet been loaded, that data is loaded. If the method is called requesting a type of data that was already loaded, then the new request is processed as a no-op. That is, the requested data is not reloaded. If the client code needs the requested data to be reloaded then the environment needs to be destroyed and a new environment obtained. There is one exception to this behavior. It is common to need to reload the ***AppInfo*** so there is a specific provision for this in the API. Client code can call ***flushAppInfo*** to expunge the current application info from the environment. A subsequent call to ***loadEnvironmentData*** with the ***AppInfo*** load flag will cause the new application to be loaded.

8.1.15 Manifest Database

8.1.15.1 Manifest Data

A central element of ModusToolbox is the “manifest database”. This is a collection of information about Applications, BSPs, and Libraries that are available to the MTB tools.

Nearly all the manifest data is located in XML files. These files can come from many different places: the Infineon webserver, ModusToolbox packs on the user’s disk, other local directories on the user’s disk, or a remote webserver owned by the user or a third party. When ModusToolbox creates the manifest database, it gathers the information from all the relevant locations and combines that data into a single, monolithic database. During this combining, the data from different sources is

merged according to well-known precedence rules and unresolvable conflicts are identified as errors.

The following sections describe the format and content of the XML files as well as the methods used for parsing and building the manifest database. They also describe the QueryAPI interfaces for accessing and using the manifest database.

8.1.15.2 Manifest Data Files

There are three types of XML files and one JSON file used for storing manifest database information. The XML files are super-manifest files, manifest files, and dependency files. The JSON files are used for storing capability description data (also known as the capabilities database). The schemas for all the files are located here: https://gitlab.intra.infineon.com/repo/manifest-schemas.

8.1.15.2.1 Super-manifest files

Super-manifests are XML files that contain pointers to manifest files, dependency files and the capabilities description files. The super-manifest XML is organized into three sections that list the board manifest files, code example manifest files, and middleware manifest files. Each manifest file can have attributes that point to dependency files and capability description files. There can be any number of manifest files entries for each section. The references to manifest files can be any URL or file path. If it is a relative file path then that is considered to be relative to the super-manifest file. If the super-manifest file is a remote resource then relative file paths are not allowed.

Super-manifest files come from three sources: the system super-manifest, super-manifest files provided by ModusToolbox packs, and super-manifest files provided by the user. Each of these sources has a priority and when there is conflicting data, the higher priority item wins. The system super-manifest, and all the manifest data reference by it, is labeled as priority level 1. The data loaded from ModusToolbox packs is labeled with priority level 2. The data loaded from the user provided super-manifests is labeled with priority level 3.

The system super-manifest file is the super-manifest file that contains all the assets supplied by Infineon as part of the base ModusToolbox package.

A description of super-manifests provided by ModusToolbox packs is found in section 8.1.17.

The super-manifest files provided by the user are specified by a set of URLs, one per line, located in a file known as the manifest location file. It is looked for in the following locations. Only a single manifest location file is read. The first one found wins.

- A file specified by the environment variable “CyManifestLocOverride”.
- A file located at ~/.modustoolbox/manifest.loc, where the ~ is the platform specific home directory for the user.
- A file located at ~/.ModusToolbox/manifest.loc, where the ~ is the platform specific home directory for the user.

8.1.15.2.2 Manifest files

Manifest files are XML files that contain lists of assets. There are three types of manifest files: app, bsp, and middleware. Most of the fields in these three different types of files are the same but there are minor differences.

8.1.15.2.2.1 Common fields

The following are the common fields that are used by more than one manifest type:

- id – A unique ID for this asset. Older versions of ModusToolbox used the ID as a unique key in internal data structures. Later versions of ModusToolbox (3.0 and later) have transitioned from using the ID field to using the reponame portion of the URL. It is expected that the ID will be completely deprecated in future versions. It is also expected that in existing data, the ID for an asset is identical to the reponame. For now, this field is required.
- board_uri or uri – The URL to the git repo of the asset. This URL must end in the reponame of the asset. The trailing “.git” suffix is optional. All versions of a particular asset must use the same URL. All URLs in the entire database must have a unique reponame. The reponame portion of the URL is how the QueryAPI maps from a .mtb file to an asset in the manifest database. This field is required.
- name – The name of the asset as displayed to the user as a part of a user interface. This field is required.

- summary – A short description of this asset (typically a single line) that can be displayed in a user interface. This field is optional.

- description or desc – A larger textual description of this asset that can be displayed in a user interface. The GUI display for this data will treat it as HTML. This field is optional.

- category – A category for displaying this hierarchically in a user interface. This field is optional.

- version.num – The user-facing name for the version of this asset for display in a user interface. There is no syntactical requirement for this user-facing version but standard ModusToolbox version naming policies should be followed. The field is required.

- version.commit – The git commit to use when checking out this specific version of this asset. This should conform to the ModusToolbox version naming policies for git commits. This field is required.

- prov_capabilities, prov_capabilities_per_version– A list of the capabilities that this asset provides. This is a string that conforms to the capability syntax. This data can be specified at the group level (for the entire asset) and/or the item level (for a specific version). It is optional.

- req_capabilities, req_capabilities_v2, req_capabilities_per_version, req_capabilities_per_version_v2 – The capabilities that this asset requires. This string conforms to the capability syntax for either version 1 or version 2. This data can be specified at the group level (for the entire asset) and/or the item level (for a specific version). It is optional.

- flow_version – The flow_version indicates what build flow this asset needs to use. Older versions of ModusToolbox (prior to MTB 2.2) were “1.0”. As of ModusToolbox 3.0, only “2.0” is supported. All versions of assets that only support flow version “1.0” are omitted from the database. This attribute is optional.

- tools_min_version – The minimum tool version that supports this asset. The version should be specified using only two digits. Earlier versions of the manifest database XML schema files required three digits for this version but this was a mistake since only two digits of the version are used for minimum tool version filtering. This field is optional. The default value is “0.0” which results in no filtering.

- tools_max_version – The maximum tool version that supports this asset. The version should be specified using only two digits. Earlier versions of the manifest database XML schema files required three digits for this version but this was a mistake since only two digits of the version are used for maximum tool version filtering. This field is optional. The default value is “0.0” which results in no filtering.

- not_for_locking – this attribute indicates if this asset is a valid target for latest-locking. See section 8.1.22.4.3 for more information on latest-locking. This attribute is optional.

- default_location, default_location_per_version – this indicates this asset should be created locally or shared. This attribute is optional.

8.1.15.2.2.2 App specific fields

- keywords – an app (code example) asset can specify a comma separated list of keywords. These are used when the user does a text-based search in the GUI. This attribute is optional.

8.1.15.2.2.3 BSP specific fields

- chips – this XML elements specifies the MCU and radio chips that are used by this BSP. This is used for the GUI only. The MCU field is required and the radio field is optional.

- documentation_url – this is a link to the online datasheet for the BSP. It is meant to be displayed by the GUI as a link that users can click on. This field is optional.

8.1.15.2.2.4 Middleware specific fields

- version.desc – This is version-specific description text. It is meant to be appended to the description text for the entire group when this particular version is selected in the GUI. This field is optional.

- prefix – this attribute indicates a “prefix” that will be prepended to the target path when this asset is cloned. This attribute is optional.

- mutex_group – this field is used to define a “mutually exclusive” group. Of all items within the same mutex group, only one can be selected. The GUI enforces this. This field is optional.

- type – this attribute indicates the type of library. There are two values for this attribute. The “device_data” value means that the

associated asset is a DeviceDB asset. The “device_data” value corresponds to the “devicedb” property in the asset’s props.json file. The “device_support” value means that the assets may provide additional device information and may provide personality information. The “device_support” value corresponds with the “configurator_support” property in the props.json file for an asset. This attribute is optional.

- hidden – when present and set to “true”, this attribute indicates that this library should not be shown in the user interface. This attribute is optional.

8.1.15.2.3 Dependency files

The dependency files indicate the dependencies between assets. Each dependency is shown as a one-way dependency from a “depender” to a “dependee”. The dependency declarations in the file use the ID and commit fields from the associated manifest XML files. Note that in future versions of ModusToolbox, we will try to deprecate the ID field entirely. At that time, we will expect the ID field in the dependency XML file to correspond to the reponame from the URL field in the manifest XML file.

8.1.15.2.4 Capability description files

The capability description files provide details about each of the capabilities that are provided or required in the manifest files. Each description has the following attributes:

- category – A user-facing category for display purposes
- description – A user-facing description for display purposes
- name – A user-facing name for display purposes
- token – The actual capability. This is what appears in the manifest files.
- types – A list of types that this capability has. This is used for capability filtering with a sensitivity list (see 8.1.15.7.3).

8.1.15.3 Assets in multiple files

Assets can appear in multiple manifest files. These can be manifest files coming from the same source or from different sources.

Here are the use cases where the same asset appears in multiple manifests.

- Flow version specific manifest files – Due to the need to maintain backwards compatibility and the fact that newer versions of the MTB tools (around version 2.2) needed to restructure the manifest XML files, it is necessary to split some assets between a “fv1” and “fv2” version of the manifest file. In these cases, the “fv1” file contains some versions of the asset and the “fv2” file contains the rest of the versions.

- ModusToolbox Packs – There are two types of packs: Tech Packs and Early Access Packs. Both of these can contain manifest data. That data can either be new assets that do not already exist anywhere or new versions of assets that already exist in the system manifest. This mechanism is used when Infineon needs to introduce a special version of an asset for the particular needs of the associated pack. The QueryAPI supports the ability for a pack to contain a replacement version of existing assets but this should not be used as a general practice. This ability to replace versions exists to allow for exceptional situations where an asset must be replaced. For example, to replace a completely broken asset version in a situation where it’s not possible to upgrade the asset. This should be a rarely used (if ever) feature of packs. Note that the ability for packs to replace versions of assets may disappear in the future.

- User-specified assets – The user can provide a manifest.loc file that contains additional manifest data. This data can include new assets that are not already in the manifest data, new versions of assets that already exist, or replacement copies of versions of assets that already exist. This mechanism is used when the user wants to add new content to their ModusToolbox environment or when they want to replace an Infineon supplied asset with their modified version of this asset (due to a bug fix or a desire for custom behavior).

The way that data from different manifests is merged is described in section 8.1.15.5.

8.1.15.4 Downloading files

The super-manifest files and manifest files referenced by the super-manifest files are XML files located on a remote server (typically GitHub, but any reachable server is fine) or on the local file system. These

references are expressed as URLs. The FILE://, HTTP://, and HTTPS:// protocols are supported.

Downloads are handled by the ***DownloadManager*** (see 8.1.24.1).

8.1.15.5 Parsing Manifest Files

There are two types of XML files that need to be parsed: super-manifest files and manifest files. The result from parsing a super-manifest file is a list of URLs to manifest files. The result from parsing a manifest file is a list of ***ManifestGroup*** objects or ***Dependencies***. Parsing is a foreground operation.

Parsing is DOM-based using ***QDomDocument***. The parser classes are ***SuperManParser*** and ***ManifestParser***.

Errors and warnings generated while loading and parsing manifest files are added to the environment's error list.

8.1.15.5.1 Data conflicts

Since data for a single asset can come from multiple sources, it is possible that there will be different values for the same field. For example, suppose that versions 1.0 and 1.1 of a code example were in one manifest file and the versions 1.2 and 1.3 were in a different manifest file. Now, it would be possible for the description text (which is defined at a group level) to differ between those two manifest files. This is called a "conflict". When conflicts are encountered during the construction of the ManifestDb they must be handled. There are two broad categories of conflicts: functional and non-functional. Some of the database fields (e.g., description) are for user display purposes only. These are considered non-functional fields since they do not alter the functional behavior of the user's project. Functional fields are the ones that can have an impact on the user's project. For example, capabilities are considered functional fields since they control what middleware is available for a project (based on the project's BSP). If these are wrong, the user can end up with code in their project that won't compile.

The rules for handling conflicts are these:

1) If two conflicting assets come from sources with different priorities, the one with a higher priority wins over the lower priority one. There is no error.

2) If two conflicting assets come from sources with the same priority and the conflicting data is non-functional, a warning is printed and the first value seen is kept.

3) If two conflicting assets come from sources with the same priority and the conflicting data is functional, ALL copies of that asset are discarded (including all future instances) and an error is printed. The reason that all versions are discarded is because we have no way of knowing which of the values is correct. It is better to omit all the data than to keep a wrong value and risk that the user didn't notice the error message.

8.1.15.6 Data Model

The MTBQueryAPI contains a data model that matches the manifest database structure fairly closely. The two primary concepts for the data model are a "group" and an "item". A "group" represents a particular App, BSP, or Library while an "item" represents a specific version for that group. For example, the group "PSoC 6 Empty App" represents a code example while the item "PSoC 6 Empty App version 1.0" represents the specific version (1.0) of that App.

These concepts are represented by the classes ***ManifestGroup*** and ***ManifestItem***. These classes are specialized by ***AppGroup***, ***BspGroup***, ***LibGroup***, ***AppItem***, ***BspItem***, and ***LibItem*** classes. The base classes contain data fields and behavior common to all groups or items. The derived classes contain data fields and behavior specific to the type of group or item that class represents.

Each ***ManifestGroup*** contains a list of one or more ***ManifestItems***. When items are created, the common data from the group is copied into the item so that each item is complete on its own.

The manifest database is returned to the client as a read-only data structure.

8.1.15.7 Capabilities

Capabilities are used to describe a relationship between two manifest database items. One of the items is considered the "provider" and the other item is considered the "consumer". Capabilities are used to match a provider and a consumer. For example, project-creator will use the capabilities matching mechanism to get a list of code examples that are compatible with the project's BSP.

In general, BSPs are considered to be providers and Apps and Libs are considered to be consumers. There is no restriction that prohibits an item from being both a provider and consumer.

8.1.15.7.1 Representing Capabilities in the Database

Capabilities are represented in the database using the "prov_capabilities" and "req_capabilities" XML elements or attributes. Capabilities can be declared at the group level (as an XML element) and the version level (as an XML attribute). If capabilities are specified at both levels, they are merged to create an “effective capabilities” set for each item. This effective capability set is used in the matching algorithm.

Capabilities (both provided and required) are specified as a space delimited string of tokens. The tokens in the string are considered to be ANDed together. That is if a provider specifies "a b c" that means that it provides "a and b and c". If a consumer specifies "a b c" that means that it requires "a and b and c".

As of MTB 2.3, required capabilities can be specified using a syntax that allows "or" groups. In this syntax, all items enclosed in "[]" are ORed together. For example: "[a,b] c d" means "a OR b AND c AND d". This is called the "v2" syntax and is specified in the XML files using "req_capabilities_v2" elements or "req_capabilities_per_version_v2" attributes. When merging group-level and version-level required capabilities to create the effective capability set, if both the v1 and v2 capabilities are specified at a level (group or version) then only the v2 version of the capabilities is taken. This allows for a global level v2 capability specification to be merged with a version level v1 capability specification.

The following table shows all combinations of global and version-specifc capabilities for both v1 and v1 syntaxes. The final colum of the table shows what the effective capability set will be.

Global	**Version**	**Effective**
V1	V1	Global V1 + Version V1
V1	V2	Global V1 + Version V2
V1	V1, V2	Global V1 + Version V2
V2	V1	Global V2 + Version V1

V2	V2	Global V2 + Version V2
V2	V1, V2	Global V2 + Version V2
V1, V2	V1	Global V2 + Version V1
V1, V2	V2	Global V2 + Version V2
V1, V2	V1, V2	Global V2 + Version V2

Each capability token must be a valid C identifier. That is, it must start with a letter followed by zero or more letters, digits, or underscores. Capabilities are case sensitive.

8.1.15.7.2 Capability Descriptions

Manifest file entries in the super-manifest may contain a “capability-url” attribute pointing to a file with human-readable information about capabilities. Capability descriptions are optional and do not necessarily represent the comprehensive set of capability names used in the associated manifest group.

The capability description file is a JSON document with the following fields:

```
{
    "capabilities": [
        {
            "name": "NAME",
            "token": "TOKEN",
            "category": "CATEGORY",
            "description": "DESCRIPTION"
            "types": ["TYPE", …]
        },
        …
    ]
}
```

The **name** (required) is the human-readable name of the capability.

The **token** (required) is the name of the capability as it appears in prov_capabilities and req_capabilities fields. The token name must be a valid C identifier and should be unique (if duplicate tokens appear, the query API will issue an error).

The **category** (required) is the human-readable category name of the capability.

The **description** (required) is the human-readable description of the capability.

The **types** (required) property contains the types of items associated with the capability. The currently available types are "chip" and "board" but others may be added in the future.

When manifest groups are merged, the merged group will contain the capability description data of the highest precedence group.

When manifest groups are merged with the same precedence level, the capability description data will be combined. If multiple capability descriptions have the same token, the QueryAPI will issue an error message.

8.1.15.7.3 Capability Matching

The primary purpose of capabilities is to allow the QueryAPI to algorithmically find compatible assets. This is called capability matching and it requires that one asset is a provider and another is a consumer.

The trivial cases for deciding of two assets match occurs when one or the other do not provide or require any capabilities. A provider that provides no capabilities can only match consumers that require no capabilities. A consumer that has no required capabilities is always satisfied by any provider.

In cases where the provider and consumer both have capabilities, a third data item is involved in the matching algorithm. This is called the “sensitivity list”. The sensitivity list is a list (possibly empty) of capability *types* that the matching algorithm is sensitive to. See 8.1.15.7.2 for information about capability types.

At a high level the rules are as follows:

- If the sensitivity list is empty, then every capability in the consumer must be satisfied by the provider.
- If the sensitivity list is non-empty, then every capability in the consumer that has a type listed in the sensitivity list and every capability in the consumer that has an unknown type (i.e., not in the capability description database), must be satisfied by the provider.

- A provider satisfies the consumers capabilities if all capabilities in an AND group are present in the provider and at least one capability from an OR group is present in the provider.

8.1.15.7.4 Examples

Provider specifies at the group level: a

Provider specifies at the version level: b c

Provider effective set: provides “a AND b AND c”

Consumer specifies at the group level: a b

Consumer specifies at the version level: c

Consumer effective set: required “a AND b AND c”

Result: satisfied

Provider specifies at the group level: a

Provider specifies at the version level: b c

Provider effective set: provides “a AND b AND c”

Consumer specifies at the group level: a b

Consumer specifies at the version level: c d

Consumer effective set: requires “a AND b AND c AND d”

Result: not satisfied (required capability "d" is not provided)

Provider specifies at the group level: a

Provider specifies at the version level: b c

Provider effective set: provides “a AND b AND c”

Consumer specifies at the group level: a

Consumer specifies at the version level: [c,d] (v2 specification)

Consumer effective set: requires “a AND c OR d”

Result: satisfied

Provider specifies at the group level: a

Provider specifies at the version level: b c

Provider effective set: provides “a AND b AND c”

Consumer specifies at the group level: a b

Consumer specifies at the version level: c d e (ignored because v2 exists at this level)

Consumer specifies at the version level: [c,d] (v2 specification)

Consumer effective set: requires “a AND b AND c OR D”

Result: satisfied

Provider specifies at the group level: a

Provider specifies at the version level: b c

Provider effective set: provides “a AND b AND c”

Consumer specifies at the group level: [a,b] (v2 specification)

Consumer specifies at the version level: c d

Consumer effective set: requires “a OR b AND c AND d”

Result: not satisfied

Provider specifies at the group level: a

Provider specifies at the version level: b c

Provider effective set: provides “a AND b AND c”

Consumer specifies at the group level: a b (ignored because v2 exists at this level)

Consumer specifies at the group level: [a,b] (v2 specification)

Consumer specifies at the version level: c d

Consumer effective set: requires “a OR b AND c AND d”

Result: not satisfied

Provider specifies at the group level: a

Provider specifies at the version level: b c

Provider effective set: provides “a AND b AND c”

Consumer specifies at the group level: a b (ignored because v2 exists at this level)

Consumer specifies at the group level: [a,b] (v2 specification)

Consumer specifies at the version level: c d e (ignored because v2 exists at this level)

Consumer specifies at the version level: [c,d] (v2 specification)

Consumer effective set: requires “a OR b AND c OR d”

Result: satisfied

8.1.15.8 Generating manifest files for local content (LCS)

The Local Content Manager tool needs to generate manifest files to store the metadata for local content repositories. ***IManifestWriter*** (available via ***IModusToolboxEnv::getManifestWriter***) provides this functionality.

8.1.15.8.1 Differences between LCS manifests and ordinary manifests

LCS manifest files have an additional attribute “precedence” for the following elements.

- <app>
- <board>
- <middleware>

- <version>

The value of the “precedence” attribute may be “system”, “pack”, or “local”, corresponding to the original source of the group or item. This allows readers to accurately prioritize LCS manifest entries with respect to other manifest sources.

URLs in the LCS super manifest are generated as relative paths assuming that the sub-manifests are in the same directory as the super manifest. The Git URLs of the items are unmodified.

8.1.16 Application Info

The application information is provided via an ***AppInfo*** object. This object provides information about the “tools” directory, the application structure (MTB2x vs MTB3x), and the set of projects that make up the application.

8.1.16.1 Loading the Application Info

Loading the application information is performed by two classes. The ***AppLoader*** class manages the loading process and the ***AppEnvironment*** class acquires the required data.

The ***AppEnvironment*** is responsible for retrieving data from the directory provided in the call to ***loadEnvironmentData***. This data is retrieved as described in section 8.1.9.

The ***AppEnvironment*** maps ModusToolbox 2.x data to the 3.x conventions as described in the SAS and checks that all required data is present. Since the ***AppEnvironment*** has access to all the raw data values from the project, it is responsible for determining the ***AppStructure*** and ***FlowVersion*** for a project. This operation may involve external calls to make to retrieve data depending on whether the data is provided via response or cache files. If a flow version 1 application is detected, a fatal error is issued since the QueryAPIs do not support flow version 1 applications.

Loading the application information starts in the ***ModusToolboxEnv*** class which creates an empty ***AppInfo*** object and passes it to the ***AppLoader***.

The ***AppLoader*** class instantiates the ***AppEnvironment*** class and calls the ***init*** method to read the get_app_info information. After this the ***AppLoader*** initializes the ***AppInfo*** object with that data.

Next, each project detected by the ***AppEnvironment*** is processed and a ***ProjectInfo*** object created for each and added to the ***AppInfo***. This

ProjectInfo object is initialized with the relevant information from the ***AppEnvironment*** (e.g. target, device, cache directory, etc.).

Finally, the ***ProjectInfo*** method ***initialize*** is called with the project ***AppSearchPath*** and ***AppIgnorePath*** as provided by the build environment data. This method does the following:

- Prepend the application directory to AppSearchPath.
- Reads project level “MTB” files to get all of the direct and indirect dependencies for the project
- Adds the direct and indirect asset directories to the ***AppSearchPath***
- Find BSP Instances in the project (section 8.1.16.3)
- Adds ignore paths, stop-at paths, and addme paths from props.json in each asset as well as the BSP directory (including local BSPs).
- Look at the top level of each search path as well as the BSP directory for a .cyignore file and add entries to the ***AppIgnorePath***. The ***AppIgnorePath*** does not affect the search for cyignore files.
- Difference from MTB 2.4: The older make-based implementation of cyignore would remove items from the search path if they are descendants of an ignored path. In MTB 3.0, ignore paths do not affect descendant search paths. For example, if the ignore path contains /A and the search path contains /A/B then MTB 2.4 would ignore /A/B but MTB 3.0 would not ignore /A/B.

The search method finds files using the ***AppSearchPath***, ***AppIgnorePath***, and addme lists.

- Search is performed depth-first from the ***AppSearchPath*** directories.
- The search will end if it encounters a directory that has already been entered (may be possible with overlapping ***AppSearchPath*** directories).
- The search will end if it exceeds the depth limit.
- The search will end if it encounters a directory that is in the ignore list unless the directory is also in the addme list. If a directory is

ignored, the search method will process descendant paths from the addme list.

- Subdirectories and files are processed in case-insensitive locale-insensitive lexographical order.

8.1.16.2 Public Interface

The ***IModusToolboxEnv*** interface always returns ***AppInfo*** objects as "const" objects to the client. The "const" methods provide information about the application. The most notable of these is the method ***getProjects*** that returns a collection of ***ProjectInfo*** objects.

There are many "non-const" methods on the ***AppInfo*** class. These are not available for use by clients and are intended to be used by the internal implementation of the MTBQueryAPI.

8.1.16.2.1 The ***AppInfo*** API

The ***AppInfo*** API provides information about the application as a whole and not detailed information about the projects contained in the application. The primary API methods are those for getting the application structure, the tools directory associated with the application, the root directory of the application, the list of projects, and the device support files associated with this application.

8.1.16.2.2 The ***ProjectInfo*** API

The ***ProjectInfo*** API provides access to all the details about a project. These details can be categorized into several different areas:

- Device information: there are several methods that return the project's TARGET, the list of MPNs for the project, the list of devices and additional devices, and information about the core of the devices.

- Associated directories and paths: methods to find important directories and paths. For example, the "libs" folder path.

- Project attributes: methods to find out the project and application names, the components, the toolchain, the configuration, the application structure, and the project type.

- Project assets: methods to get the list of asset requests and asset instances in a project. This includes the ability to get bsp instances

(a special form of an asset instance). This also includes the ability to get the dependencies of the BSP in the project.

- Search parameters: methods to retrieve user provided (via makefile) parameters used for searching. These include the search path, the “stop include” list, and the ignore path.

- Search: methods to search a project for files. There are several variations of the search method that allow the caller to filter based on a regular expression, filter based on folder naming conventions (COMPONENT_, TOOLCHAIN_, etc.), limit which directories in the project to look at, limit a search to just the BSP, and so on. All the searches are recursive given a starting set of directories and following the filters and limitations provided by the arguments.

- ***ProjectChangeMgr***: a method to return a ***ProjectChangeMgr*** for this project.

8.1.16.3 BSP Instances

Applications can contain BSP folders. For MTB 2.x applications, this happens when the project is created using an imported BSP (also known as a custom BSP for 2.x applications). For MTB 3.x applications, local BSPs are converted to “in-app BSPs” and these are folders in the application. Any BSP folder in the application is represented as a ***BSPInstance*** in the application.

Each ***BSPInstance*** contains a props.json file which contains a list of capabilities that this ***BSPInstance*** provides.

8.1.17 ModusToolbox Packs

ModusToolbox packs are content and tools that are added to the ModusToolbox environment. ModusToolbox packs are described completely in the SAS.

ModusToolbox packs are loaded using the ***PackScanner*** object. This object scans a specific directory for JSON files that describe content that has been installed by the Infineon Developer Center. This directory contains a set of JSON files that describe each installed package. If the JSON file contains a field name “type” and the value of this field is “content-pack” then this entry is considered a ModusToolbox pack. This JSON file is further processed and the following fields must exist.

Field	Purpose
guid	A unique identifier for the content-pack. This is expected to be a random GUID that remains consistent across all releases of this ModusToolbox pack.
featureId	A unique id of the content-pack to identify it across different versions (e.g. com.ifx.mtb.mlpack)
title	The name of the content-pack.
version	This is a human readable string that describes the version of this content.
versionNumeric	This is numeric version number that is well suited for processing by tool.
path	The path to the installed content-pack.
tags	A list of tags used to search for specific content installed by the Infineon Developer Center.
description	The description of this pack
exePath	Path to the launch-tool.exe.
toolImage	The path to an icon or image that represents this ModusToolbox path.
uninstallPath	The path to an executable that can uninstall this ModusToolbox pack.
help	The path to a help file describing the ModusToolbox pack and where additional information can be found.
licenses	Path to the local license text file
release	Path to the local release notes text file

In addition, if the JSON file describing the ModusToolbox pack contains an entry named “attributes”, the contents of this JSON object are copied directly to the QueryAPI ***PackInfo*** object to be retrieved by clients as needed.

Early-access packs are a type of ModusToolbox pack used to provide early access to new devices. These are identified by the attributes.pack-type property in the JSON file having a value of “early-access-pack”. ModusToolbox only loads early access packs if the featureId matches the value of the MTB_ENABLE_EARLY_ACCESS environment variable. Only one early access pack can be enabled at a time.

See the link below for more information about the Infineon Developer Center requirements for content that is installed via the Infineon Developer Center. There may be additional requirements or fields required by their design.

https://confluencewikiprod.intra.infineon.com/pages/viewpage.action?spaceKey=DP&title=Toolbox+Launcher+2.0+-+Environmenthttps://confluencewikiprod.intra.infineon.com/pages/viewpage.action?spaceKey=DP&title=Toolbox+Launcher+2.0+-+Environment

When the URLs in the manifest data in ModusToolbox packs refer to data in that pack, it is necessary to use the MTB protocol. See 8.1.20.2.1 for information about this protocol.

8.1.18 Placeholder Entry Support

Placeholder entry support is an advertisement feature related to content distributed by ModusToolbox Packs. Its primary purpose is to inform end users about additional features which are available in ModusToolbox Packs that the user has not installed or enabled.

The main idea of placeholder support is that public manifests can contain entries that refer to items that are distributed via ModusToolbox Packs. These are called placeholder entries. They can be entries for BSPs, code examples, or middleware libraries.

The format of a placeholder entry is mostly the same as the format for entries in Tech Packs. The only difference is that the ***description*** *(or* ***desc****)* field contains text that tells the user that the item is located in a ModusToolbox Pack and which pack that is.

Placeholder entries are not allowed to be 'split'. That is, some versions of an asset available in the public github and other versions of the same asset in a ModusToolbox Pack. In such a scenario, it is necessary to create a separate entry in the ModusToolbox Pack with a different name.

When the QueryAPI loads the Manifest Db, it will automatically detect placeholder entries. This is done when processing an asset that has a "techpack" (see 8.1.19.2.1) URL scheme. The database loader checks to see if the ModusToolbox Pack with the associated GUID is loaded or not. If a placeholder entry comes from an Early Access Pack, the system also checks to see if that Early Access Pack is enabled. It uses this information to set a "placeholder" value on the asset. This value is exposed via an ***getPlaceholderType()*** method on both the ***ManifestGroup*** and ***ManifestItem***. Clients should use these methods to determine placeholder entries for changing their behavior. They can only show placeholder entries in lists but can't operate them.

8.1.19 MTB File Requirements

The QueryAPIs use mtb files (.mtb or .mtbx) to store information about assets in an application. Each file contains three parts: URL, commit, location. A fourth part, the repo name, is derived from the location field or the URL (see section 8.1.20). Given these items, the QueryAPIs have the following requirements:

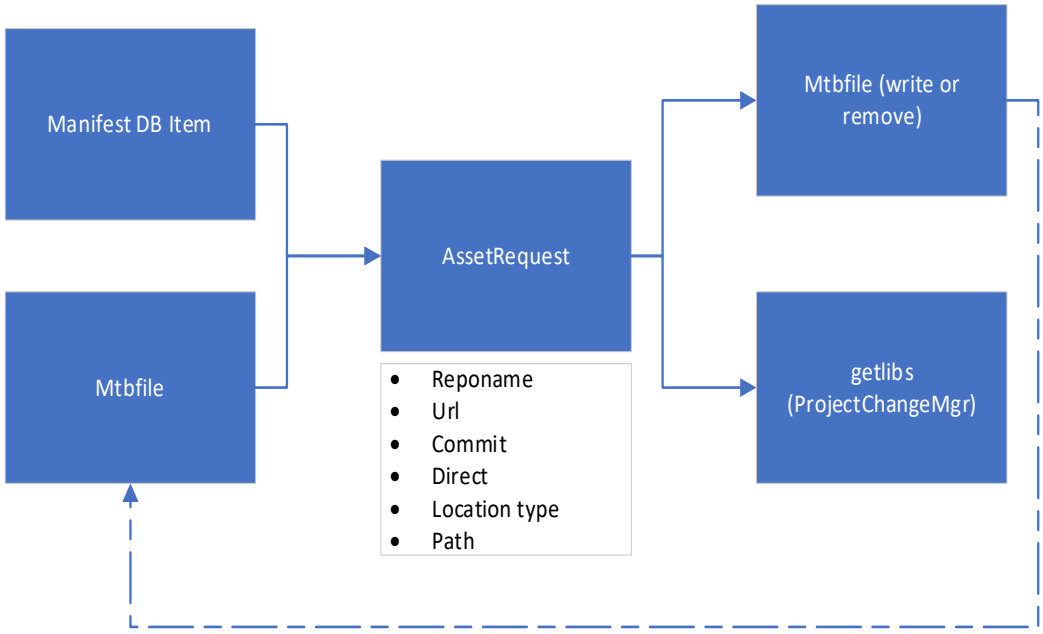
- The mtbfile must be named “reponame.mtb” or “reponame.mtbx”.
- For a local asset the location field is expected to be in the format “$$LOCAL$$/reponame”
- For a shared asset the location field is expected to be in the format “$$ASSET_REPO$$/reponame/commit”
- For a global asset the location field is expected to be in the format “$$GLOBAL$$/reponame/commit”
- For an absolute asset the location field is expected to be in the format “$$ABSOLUTE$$/path”.
- For a project-relative asset the location field is expected to be in the format “path”.

When the QueryAPIs need to correlate an asset represented by an mtbfile to an item in the Manifest DB, the reponame from the mtbfile is used to find the corresponding manifest group. The commit from the mtbfile is used to find the corresponding manifest item in that group. If either of these values (reponame or commit) are not available in the Manifest DB then the QueryAPI is not able to correlate the mtbfile with an item in the database. In this case the mtbfile is assumed to refer to a user-supplied asset and has limited treatment in the QueryAPIs.

8.1.20 Asset Request

Many parts of the QueryAPIs revolve around downloading and managing git-based resources. For example, the ***ApplicationCreator*** may download a code example or BSP from git as a part of creating an application. These git-based resources are represented by a class called ***AssetRequest***. The ***AssetRequest*** contains the information needed to allow the QueryAPIs to clone or update a resource. Also, the ***AssetRequest*** knows how to store itself in the application in the form of a “.mtb” or “.mtbx” file.

The following diagram shows an overview the ***AssetRequest*** class.



Internally, an ***AssetRequest*** has several key pieces of information:

- The repo name – this is the name of the git repo for this asset. If the ***AssetRequest*** is being created from an mtb file, the reponame is derived from the location field. If the ***AssetRequest*** is being created from a manifest item, the reponame is derived from the URL in the manifest.

- The URL – this is the URL for the git clone command. It is provided by the client during construction of an ***AssetRequest***.

- The commit – this is the commit for the git checkout command. It is provided by the client during the construction of an ***AssetRequest***.

- The “direct” flag – this indicates if the asset is a “direct” or “indirect” asset in the application. It is provided by the client during the construction of an ***AssetRequest***.

- The location type – this indicates if the asset is “local”, “shared”, etc. The location type is used to determine the “first half” of the final destination for cloning an asset. This information is provided by the client during the construction of an ***AssetRequest***.

- The path – this is the “second half” of the final destination for cloning an asset. It is derived from the repo name, the commit, the

location type, and “prefix” data from a manifest item during the construction of an ***AssetRequest***.

8.1.20.1 Creating an *AssetRequest*

As shown in the diagram above, an ***AssetRequest*** can be created starting with data taken from the Manifest DB or it can be created starting with data taken from a “.mtb” or “.mtbx” file.

When an ***AssetRequest*** is being constructed from a Manifest DB item the fields are set as follows:

```
reponame = item.url.reponame;
url = item.url;
commit = item.commit; // or passed in the constructor
direct = arg;
locType = arg; // or item.default or LOCAL
path = constructPath(locType,reponame,commit)
```

The path is constructed using the location type, repo name, and commit. If the location type is LOCAL then the path is just the “reponame”. If the location type is SHARED or GLOBAL then the path is “reponame/commit”. If the location type is ABSOLUTE then the path is explicitly provided by the client.

When an ***AssetRequest*** is being constructed from an “.mtb” or “.mtbx” file the fields are set as follows:

```
reponame = last_part_of_location_field;
url = mtbfile.url;      // or passed in constructor
commit = mtbfile.commit; // or passed in constructor
direct = arg;
locType = extractLocType(mtbfile.locationField);
path = removeSentinal(mtbfile.locationField);
```

Extracting the location type from the mtbfile location field is simply looking for the sentinel value (e.g. $$LOCAL$$). If there is no sentinel then the location type is set to PROJECT.

The ***AssetRequest*** class offers several useful methods for getting information about the ***AssetRequest***. In particular, there are methods to get the repo name, commit, URL, location type, and whether or not the asset is direct or indirect. In addition, the ***AssetRequest*** has methods that take several project related directories (e.g. “libs” for local assets) and uses those, along with the location type and path, to construct the absolute path to the directory that an asset should be cloned into.

8.1.20.2 Asset URL

The asset URL gives the location of the asset. This URL can specify a repository that is located anywhere on the internet and in this case, the

URL is generally an https protocol URL. Asset URLs can also be specified using the file protocol. This is common for user's custom manifest data referred to by the manifest.loc file. Since file-based URLs need to be specified using absolute paths, using file-based URLs as the Asset URL can be limiting (e.g., the Asset URL points to a directory in a user's home directory).

8.1.20.2.1 Custom Protocols in Asset URLs

ModusToolbox allows the use of two custom protocols for specifying the URL of an asset: mtb, techpack.

ModusToolbox packs introduce two new "protocols" within ModusToolbox. A URL request found in an "mtb" or "mtbx" file may be of the form "mtb:ID", where ID is the id of an asset. This is generally a UUID that is stored as the ID of that asset in the manifest file. This forces ModusToolbox to look up the asset in the manifest data based on the ID given.

When the item being referred to comes from a ModusToolbox pack, the URL for the location within the manifest file may be of the form "techpack:ID/relative_path". This gets translated to the install location of the pack with the given ID. The ID for the pack is expected to be a UUID.

Because of the limitations of using absolute paths in file-based asset URIs, ModusToolbox introduces two new "protocols": mtb, techpack.

The mtb protocol type is followed by the unique ID of the asset (e.g., mtb:ID_GOES_HERE). This unique ID can then be used to look up the location of the asset in the manifest database. This type of URL is useful for referencing content in a ModusToolbox pack. When a ModusToolbox pack is installed, it is generally installed in a different location on each user's machine. The URL with an mtb protocol allows the MTBQueryAPIs to look up the location of the assets in the ModusToolbox pack no matter where it is installed.

When the MTBQueryAPI encounters a URL with an mtb protocol, it uses the ID to find the item in the manifest database and then gets the URL from there.

URLs with the "mtb" protocol can be found only in .mtb or .mtbx files.

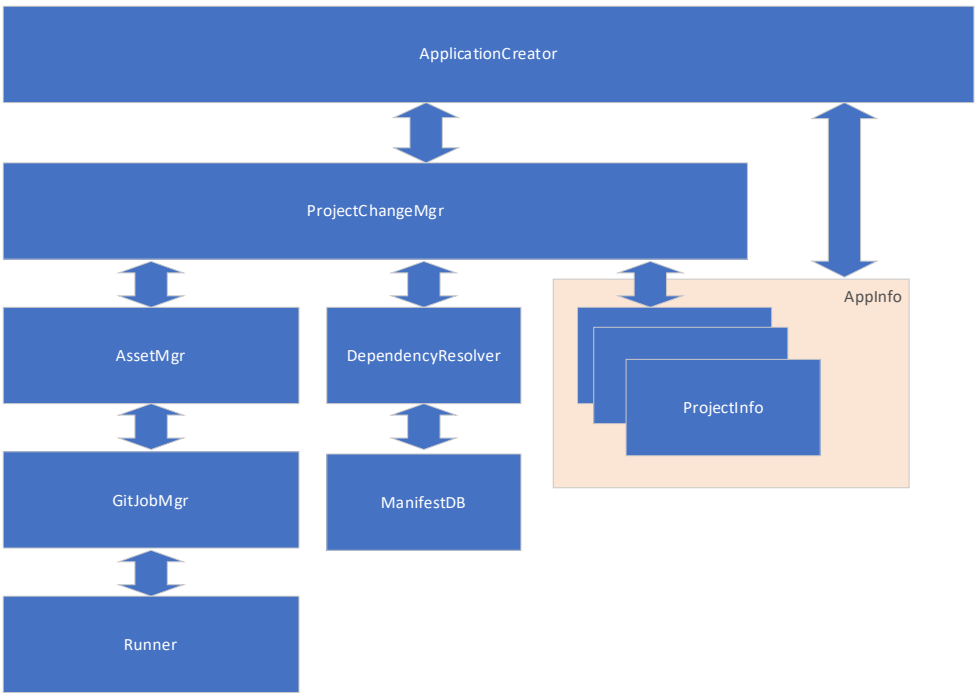
For assets delivered via a ModusToolbox pack and referenced via an mtb protocol, the URL in the manifest file uses the techpack protocol. The URL is of the form "ID,relative path". For instance, "techpack:92d937ef-0d3d-

4839-a90b-b41c7d07bd6c,content/ml". The MTBQueryAPI looks up the technology pack based on its UUID and finds the location of the asset based on the relative path that follows the comma.

8.1.20.3 Writing MTB Files

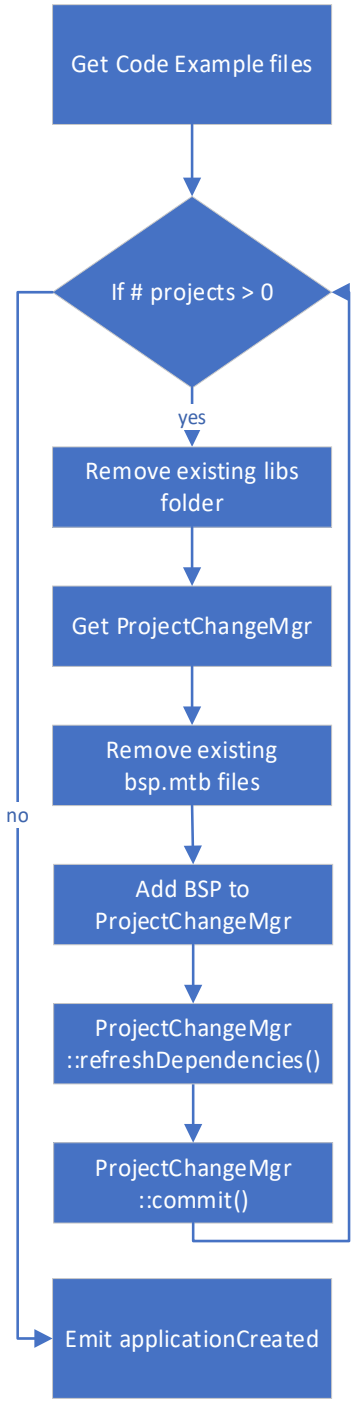
An ***AssetRequest*** knows how to write itself out to a file. This includes using the appropriate sentinel based on the location type for the asset. It will also convert itself to use the mtb protocol if the asset is referencing content in a ModusToolbox pack.

8.1.21 Application Creation



The MTBQueryAPIs allow clients to create new ModusToolbox applications using the ***createApplication*** API. Creating a new application requires a BSP and a Code Example. Either of these items can be specified as coming from a git repo or as coming from a local folder (aka, imported). There are four ***createApplication*** methods to cover each of the four combinations of BSP and Code Example sources. The create methods also require the target location and name for the application.

At a high level, the creation process is the same no matter what the sources as shown in this flowchart.



- Step 1: Acquire files from the code example source (git repo or local folder). These files are copied or cloned to a folder with the specified application name in the specified target location. If the code example comes from a git repository, the latest version of the

repository is used. For git-based code examples, the .git folder is removed after the example has been cloned and checked out.

- Step 2 (for each project): Remove the libs folder if it exists.
- Step 3 (for each project): Obtain a ***ProjectChangeMgr*** object.
- Step 4 (for each project): Remove all mtb files associated with BSPs from the deps folder.
- Step 5 (for each project): Set the new BSP. If the BSP is from the manifest DB (i.e. a git repo) then an associated ***AssetRequest*** is added to the project using the ***ProjectChangeMgr***. If the BSP is from a local folder, that folder is staged in the ***ProjectChangeMgr*** as a custom BSP folder. The actual copying of BSP files into the project happens when the ***ProjectChangeMgr::commit*** method is called later in the process.
- Step 6 (for each project): Apply the changes. The ***refreshDependencies*** and ***commit*** methods are called for each project. These calls ensure that all the required indirect dependencies are added to the project, the TARGET variable is correctly set, if there is an in-app BSP then it is copied to the application, and exported content is handled, and so on. See the ***ProjectChangeMgr*** for details (8.1.22).

Creating the application uses the ***AssetMgr*** and the ***ProjectChangeMgr*** classes. In each of these cases the streaming stdout and stderr are redirected to the ***ModusToolboxEnv*** stdout and stderr streaming signals.

8.1.21.1 Detailed Design

8.1.21.1.1 Classes

The primary classes are:

- ***IApplicationCreator*** – the public interface for application creation. This is implemented by the ***ApplicationCreator*** class.
- ***ApplicationCreator*** – this class provides four methods to create an application (one for each of the four combinations of sources for Applications and BSPs). These four methods create an appropriate ***AppSource*** and ***BspSource*** object then call a private worker method to do the creation.

- ***AppSource*** – this is an abstract base class that provides the virtual method ***getAppFromSrc***. This is a background operation and it emits the ***appAcquired*** signal when done.

- ***AppSourceFolder*** – this derives from ***AppSource***. Its implementation of ***getAppFromSrc*** copies the Application files from a local folder into the Application directory. Copying files is a foreground operation and the ***appAcquired*** signal is emitted as soon as it finishes.

- ***AppSourceGit*** – this derives from ***AppSource***. Its implementation of ***getAppFromSrc*** uses git to acquire the Application files. This uses the ***AssetMgr***. The git operations are in the background so this class waits until the git operation is done (via the ***AssetMgr***) before emitting the ***appAcquired*** signal.

- ***BspSource*** – this is an abstract base class that provides the virtual method ***addBspToProject***.

- ***BspSourceFolder*** – this derives from ***BspSource***. Its implementation of ***addBspToProject*** stages the BSP source folder with the ***ProjectChangeMgr*** as a new custom BSP and then also sets the new active target. It calls ***refreshDependencies*** with the latest-locking-directs flag set to true.

- ***BspSourceGit*** – this derives from ***BspSource***. Its implementation of ***addBspToProject*** creates an ***AssetRequest*** corresponding to the request BSP and adds that to the ***ProjectChangeMgr***. It also sets the new active target. It calls ***refreshDependencies*** with the latest-locking-directs flag set to true.

After the Application files are obtained and the BSP added to the project, the ***ProjectChangeMgr::commit*** method is called.

8.1.21.1.2 Control Flow

Internally, the application creation process performs several background operations that have to happen in sequence. These operations and signals are not visible outside ***ApplicationCreator*** but it is helpful to describe the control flow.

1. **ApplicationCreator::createApplication** calls ***AppSource::getAppFromSrc***. When the ***appAcquired*** signal is emitted ***ApplicationCreator::appAcquired*** is run.

2. ***ApplicationCreator::appAcquired*** now needs to load the new Application into the environment. When this background operation is done ***ApplicationCreator::appLoaded*** is run.

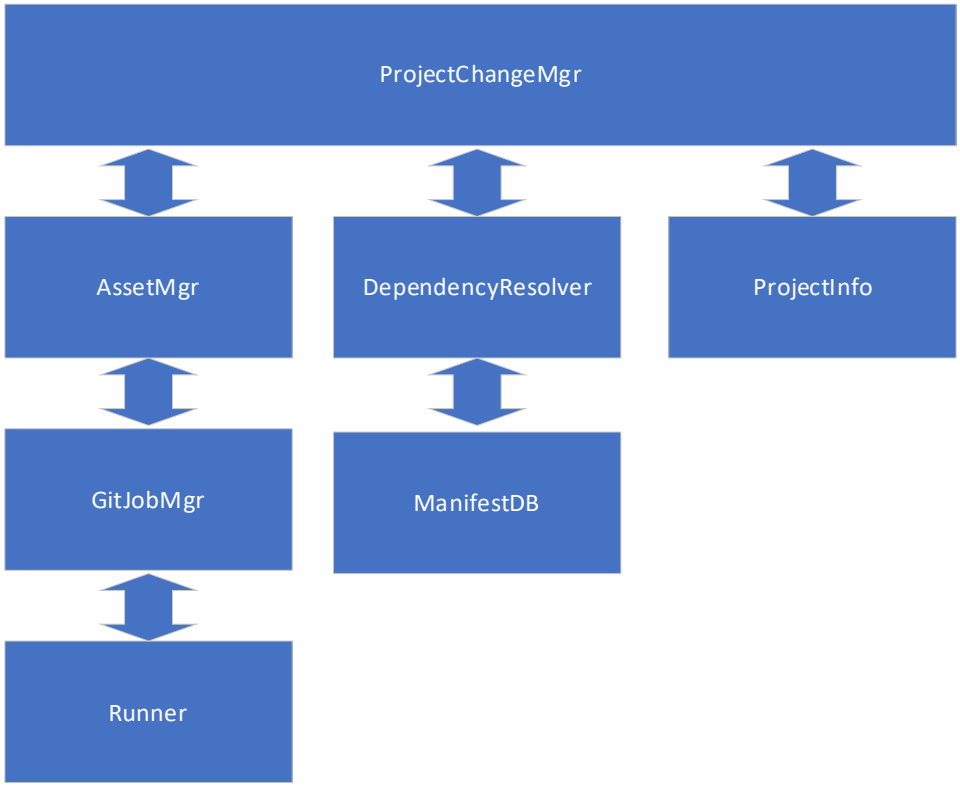
3. ***ApplicationCreator::appLoaded*** creates a list of projects (from the loaded Application) and calls ***ApplicationCreator::processNextProject***.

4. ***ApplicationCreator::processNextProject*** gets the next project from the list (from step 3), creates a ***ProjectChangeMgr*** for that project, adds the BSP to that ***ProjectChangeMgr***, then calls ***ProjectChangeMgr::commit***. When ***commit*** is done then ***ApplicationCreator::commitDone*** is called.

5. ***ApplicationCreator::commitDone*** checks to see if there are more projects to process. If so, then it calls ***ApplicationCreator::processNextProject***. Otherwise it emits the ***applicationCreated*** signal.

8.1.22 Project Change Manager (***ProjectChangeMgr***)

8.1.22.1 Overview



The project change manager provides an interface to change a project. Changes include the following:

- Adding a new middleware asset (direct dependency)
- Removing an existing middleware asset (direct dependency)
- Changing the version of a middleware asset (direct dependency)
- Changing the type of an asset (from indirect to direct, for example)
- Changing the location of an asset (from shared to local, for example)
- Changing the target BSP
- Staging and removing custom BSPs
- Viewing and removing BSP Instances in a project

Any number of these types of changes can be requested of the ***ProjectChangeMgr*** and the ***ProjectChangeMgr*** maintains a list of the minimal number actions required to execute the requested changes. No changes are made to the project until ***commit*** is called by the client software.

For instance, if a request is made to add a new middleware asset and then a request is later made to remove the middleware asset, then the net effect will be as if the middleware asset was never added.

The ***ProjectChangeMgr*** provides a method called ***getChangeLog*** that returns a list of human readable strings that describe the net changes that would happen given the current state.

Since the changes that can be made involve middleware assets, either directly or indirectly, the manifest data must be loaded in order to retrieve a ***ProjectChangeMgr*** from a project.

8.1.22.2 Adding, Updating, and Removing Middleware Assets

The ***ProjectChangeMgr*** maintains a set of “delta lists”. One each for adds, updates, and removals. It also contains an “initial list” which is the set of assets in the project before any updates are performed. The delta lists are kept in sync with each add, update, or remove request and then used to drive actions in the ***commit*** operation. For example, if there is a request to add “LibX-v1.1” and “LibX-v1.0” is in the initial asset list (from

the project), then the add request is changed to an update request and this new asset is added to the update delta list.

8.1.22.3 Changing the Target BSP

A call to change the active BSP simply sets internal state about what the new BSP will be. No actions are taken until the ***commit*** operation is called.

When the target is changed, this may trigger the ***ProjectChangeMgr*** to refresh the indirect dependencies. The indirect dependencies are determined relative to the new BSP. If the new BSP is a manifest-based BSP then the dependencies are collected from the ***ManifestDb***. If the new BSP is a custom BSP then the dependencies are collected from .mtbx files in the custom BSP. Typically, a custom BSP hasn't been copied into the project yet but it is possible that the custom BSP is already in the application (due to having been used earlier). The algorithm checks for this case to ensure that the correct .mtbx files are used.

8.1.22.4 Refreshing Dependencies

8.1.22.4.1 Overview

After changes have been made to the list of assets or the active target in a project, the set of indirect dependencies may be out of date. The dependency resolution operation is used to create a new set of indirect dependencies. The general algorithm is:

1. Get a list of all direct dependencies, limiting BSP dependencies to only those BSP assets that match the current target BSP.

2. Get extra dependencies provides by an external BSP (these are .mtbx files) if the external BSP is the current target BSP.

3. Call the ***DependencyResolver*** with the list of direct dependencies and set of extra dependencies. This returns a new set of indirect dependencies.

4. Remove all the indirect dependencies from the old set (using the Remove Asset functionality).

5. Perform latest-locking on dependencies in the new indirect set that the resolver returned. Normally latest-locking only occurs on indirect dependencies but during application creation, it is required that direct dependencies are latest-locked also.

6. Add all new dependencies using the Add Asset functionality.

The key classes are mentioned in more detail in the next subsections.

8.1.22.4.2 Dependency Resolver

8.1.22.4.2.1 Assets and versions

Assets in the manifests are defined as either BSPs, Middleware, or Apps. Elements can declare dependencies on other assets and these dependencies are version specific. It doesn't matter if the asset is a BSP, Middleware library, or App project, the dependency rules are the same.

8.1.22.4.2.2 Version naming conventions

MTB naming conventions for versions allow for the following:

latest-v*m*.X – This is interpreted by MTB tools as the "latest version for major version m". The "v" and X" are a part of the name. This is a floating tag and is always updated to coincide with the latest release for a major version. It is only possible to have one "latest-vm.X" for a given major version. Example: latest-v1.X.

release-v*x.y.z* – This is a specified major.minor.patch version of an element. This is a fixed tag in git and once it has been assigned it doesn't move. The patch level is optional. Example: release-v1.1.0.

custom_v_v – This is any custom string for a version as specified by a customer. It could be "v1.1" or "version 1" or "alpha" or whatever. MTB tools display these versions in the GUIs so that users can select them but the tools do not do any other processing on them.

Dependencies must always be specified using the release-v*x.y.z* notation. This simplifies the dependency resolution processes (outlined below).

While the QueryAPI accepts any version string, it is expected that all assets created by Infineon will use the latest or release version syntax. No assets released by Infineon (in the public manifest or in a pack) should use the custom version syntax. For example, “release_v1.1.0-beta” uses custom syntax and should not be used by Infineon. Custom versions do not participate in latest-locking and, unless the ONLY version of an asset is a custom version, they are not automatically matched as the “best version” for processes such as project creation.

8.1.22.4.2.3 Version ordering

Version ordering is important in presenting elements in GUIs and also in algorithmically picking the best match of an element when multiple versions can satisfy a constraint. Given a list of version strings for an element, the versions are ordered as follows:

1) All "latest-v*n*.X" come first, in descending order of n.

2) All "release-v*a.b.c*" in descending numerical order.

3) After sorting by 1 and 2, the "custom_v_v" version identifiers are ordered in descending lexicographical order.

For example, given the following version identifiers for an element:

"latest-v1.X", "release-v1.0.0", "release-v1.1.0", "latest-v2.X", "release-v2.0.0", "v0.9", "v1.0", "v1.1", "v2.0"

The ordered list would be:

"latest-v2.X", "latest-v1.X", "release-v2.0.0", "release-v1.1.0", "release-v1.0.0", "v2.0", "v1.1", "v1.0", "v0.9""

8.1.22.4.2.4 Resolving dependencies

Assets specify dependencies on specific versions of other assets in the dependency manifest. Dependencies for MTB provided manifest items must be specified using the "release-v*x.y.z*" form of version elements.

Dependency resolution revolves around a method called "resolve". The "resolve" method takes the current list of direct dependencies and a list of extra dependencies (from .mtbx files in custom BSPs) as input then returns a list of indirect libraries that need to be added to the design to satisfy all the dependencies.

The method works by doing a breadth-first traversal over the graph of dependencies, starting with the direct set nodes, and adding each dependency as it is encountered. This is complicated by the fact that indirect dependencies can be promoted if and only if there are multiple requests for different versions of the same item. In this case, the dependency that is added to the indirect list must be the "largest" version using the version ordering as described in a previous section. Direct dependencies cannot not be promoted. When there is a discrepancy between what is requested and what is in the direct or indirect set, the algorithm issues a warning to the user.

When there is an unresolvable conflict for required versions, that dependency becomes “dead”. After all of the dependencies have been resolved in the breadth first traversal we have a “toadd" set. But it might be the case that some items were added to the “toadd" set due to dependencies that later became “dead”. A final pass over the “toadd" set is needed to make sure that each one is still reachable from something in the direct set without using any items from the dead list.

Note: capabilities are not involved in dependency resolution.

Here is pseudo-code for resolving the dependencies:

```
List<DesignElement> resolve(const List<Asset> directs, List<Asset> extraDependencies) {

  Queue pending;
  IndirectSet toAdd;
  List deadlist;  // Assets that are dead due to version conflict
  List seenlist;  // Assets that we've already seen

  for each Asset in the direct list {
    add to pending queue
    add to seenlist
  }

  while (!pending.empty()) {
    Asset elem = pending.dequeue();
    if (!deadList.contains(elem)) {
      foreach (dependency from elem) {  // either from the asset itself or the extraDependencies
        if (!seenlist.contains(dep)) {
          pending.enqueue(dep);
          seenlist.add(dep);
        }
          // Now check for version conflicts
        if (directs.containdsId(dep.ID) with a mismatched version) {
            issue_warning("Multiple versions of %1 requested. Keeping version
%2.",existing.ID,existing.version);
        } else {
          if (toAdd.containsId(dep.ID)) {
            if (existing.version < dep.version) {
              deadlist.add(existing);
              toAdd.replace(existing,dep); // existing = old, dep = new
              issue_warning("Multiple versions of %1 requested. Keeping version %2.",existing.ID,dep.version);
            }
            if (existing.version > dep.version) {
              deadlist.add(dep);
              issue_warning("Multiple versions of %1 requested. Keeping version
%2.",existing.ID,existing.version);
            }
          } else {
            toAdd.add(dep);
          }
        }
      }
    }
  }

  // Can we reach everything in the toAdd, starting with direct set, without using anything in the deadlist?
  foreach (Asset elem in toAdd) {
    bool foundPath = false;
    foreach (DesignElement root in direct) {
      foundPath |=  bfs(root,elem,deadlist);
    }
    if (!foundPath) {
      toAdd.remove(elem);
    }
  }

  return toAdd;
}
```

When an asset requires a dependency, it can be the case that that dependency already exists in the direct set. In this case, the direct dependency is kept and nothing new is added to the indirect set. However, if the requested dependency and direct-set dependency do not have the same major version, a warning message is emitted to inform the user that there may be a problem.

8.1.22.4.3 Latest Locking

"Latest-locking" describes the situation where a latest_v*n*.X tag in a .mtb file is modified so that the latest_v*n*.X is replaced with the best matching release_v*n.x.y* tag. Generally speaking, the best matching release tag is the tag with the same major version number and the largest minor and patch version number. In the manifest file, a release_v*n.x.y* version element can declare itself to be a non-candidate for latest-locking. This supports the situation where a particular release_v*n.x.y* is not meant for general purpose use but is still in the manifest for use in special cases. This is done using the not-for-locking="true" attribute in the <version> XML element associated with the non-candidate version. If there are no available release_v*n.x.y* versions to lock the latest_v*n*.X tag, then the latest_v*n*.X tag is kept.

Latest-locking is performed by the ***LatestLockMgr*** class.

For an example, consider the following versions defined in a manifest XML file:

```
<versions>
      <version default_location_per_version="shared">
        <num>Latest 3.x</num>
        <commit>latest_v3.x</commit>
      </version>
      <version>
        <num>Latest 1.x</num>
        <commit>latest_v1.x</commit>
      </version>
      <version not-for-locking="true">
        <num>Release 1.1</num>
        <commit>release_v1.1</commit>
      </version>
      <version>
        <num>Release 1.0</num>
        <commit>release_v1.0</commit>
      </version>
</versions>
```

If the tag to be locked is "latest_v1.x" then the resulting tag, after locking, would be "release_v1.0". The "release_v1.1" tag is skipped because it is marked as not-for-locking="true".

If the tag to be locked is "latest_v3.x" then the resulting tag, after locking, would be "latest_v3.x" because there are no corresponding release tags.

The primary reason to do latest-locking is to keep library items from changing versions without the user knowing that the item is changing version. That is, "latest_v*n*.X" tags, by definition, float to the latest released version. This is a change in github, not in the .mtb file. Latest-locking replaces the latest-style tag with the best release-style tag that is available at the time that the latest-locking is performed. Because indirect libraries can be removed at any time and then repopulated by the "make getlibs" call, it is possible that when repopulating the indirect libraries, they would get locked to a newer version than they did the first time that they were created. This completely undoes the purpose of latest-locking. To avoid this problem, it is necessary to know which release-style version a particular item is locked to. This is tracked in a new JSON file in the "deps" directory called "assetlocks.json". It is a simple file that correlates a library item and the release-style version that was used to lock that item.
The format of this file is:

```
[
    {
        "asset-name": "an asset name",
        "locked-commit": "a commit string"
    },
    ...
]
```

The sample of the locking file content:

```
[
    {
        "asset-name": "cat1cm0p",
        "locked-commit": "release-v1.4.0"
    },
    {
        "asset-name": "cmsis",
        "locked-commit": "release-v5.8.0"
    },
    {
        "asset-name": "core-lib",
        "locked-commit": "release-v1.4.1"
    },
    {
        "asset-name": "core-make",
        "locked-commit": "release-v3.2.2"
    },
    {
        "asset-name": "mtb-hal-cat1",
        "locked-commit": "release-v2.5.4"
    },
    {
        "asset-name": "mtb-pdl-cat1",
        "locked-commit": "release-v3.9.0"
    },
    {
        "asset-name": "recipe-make-cat1a",
        "locked-commit": "release-v2.1.2"
    }
]
```

When the ***LatestLockMgr*** does latest-locking, it first consults the data in this history file. If this particular indirect item has been latest-locked before

(in this project) then there will be an entry in the history file. This value is used. If the item has never been latest-locked before then the best release tag is found and used. The history file is updated to reflect that this item has now been latest-locked at least once. If the indirect library is converted to a direct then the corresponding entry in the history file is removed. There is no other way for an item to be removed from the latest-locking history file.
Note: as of MTBQueryAPI ver. 1.2.0 if ***LatestLockMgr*** finds the old version of the locking file (locking_commit.log) in the “deps” folder it substitutes it with the new "assetlocks.json" file.

8.1.22.5 Viewing the Change Log

The ***getChangeLog*** method provides a user-facing list of strings that describes all of the changes that are currently queued up in the ***ProjectChangeMgr***. This includes all additions, updates, removals, and changes to the active BSP.

8.1.22.6 Commit

With calls to ***add***, ***remove***, ***update***, ***changeActiveTarget***, etc., the ***ProjectChangeMgr*** registers the requested changes and provides the activity list required to execute the changes but does not make the changes until the ***commit*** method is called.

The commit operation can be executed as a blocking or non-blocking call based on an argument to the **commit** method. If the operation is blocking the ***commit*** method will not return until the commit is complete. If the operation is non-blocking, the ***commit*** method will return immediately but also triggers several background operations. The caller must wait for the ***commitDone*** signal to know when the commit is complete.

In both the foreground and background case, the ***commitDone***, ***standardOutReady***, and ***standardErrorReady*** signals are used to communicate to the client the status of the commit operation. The commit operation uses the standard messaging signals as well (see 8.1.7).

8.1.22.6.1 The ***commit***() Steps

The entire commit process requires several steps which are divided into three phases: the actions to take before cloning items, the actions to take to update the project on disk, the actions to take after the project info has been updated. The first phase involves removing old resources and downloading new ones. The second phase involves updating internal data structures, updating the TARGET variable, doing the in-app BSP

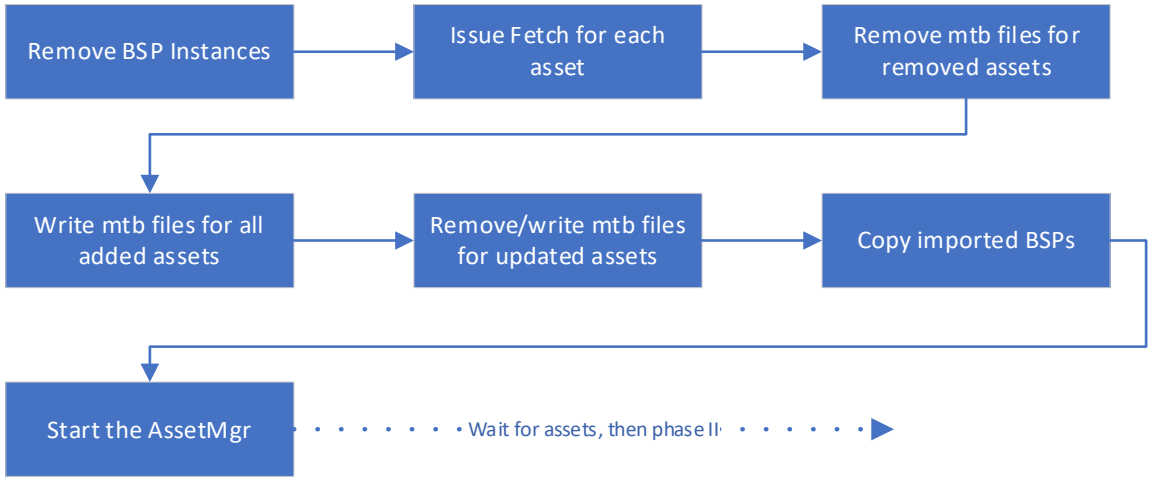
processing, exporting content, writing the latest locking file, and refreshing the in-memory model. The third phase generates the data derived from the project information (e.g., the mtb.mk file).

In the first phase, git commands are issued using the ***AssetMgr***. The ***AssetMgr*** requests are handled as background operations. When all submitted requests have been processed by the ***AssetMgr***, the second phase tasks are executed. Most of the tasks in the second phase are done as foreground operations but the final step is to reload the project information which requires a call to “make get_app_info”. Once the project has been updated, the third phase is called. All the tasks in the third phase are in the foreground.

8.1.22.6.1.1 Phase I

The primary purpose of the first phase is to remove unused assets and bring new or updated assets into the project. Assets can be manifest-based or, in the case of BSPs, can be folders (i.e., “imported BSPs”). There can be any number of assets to be added or removed from the project. Since bringing in manifest-based assets requires running git, this phase is a “background” operation.

Here is a flow chart showing the steps in Phase I.



- Remove any BSP instances that are on the ***mRemovedInstances*** list. Items are added to this list using the ***removeBspInstance*** method. Each removed BSP directory is archived as a zip file.

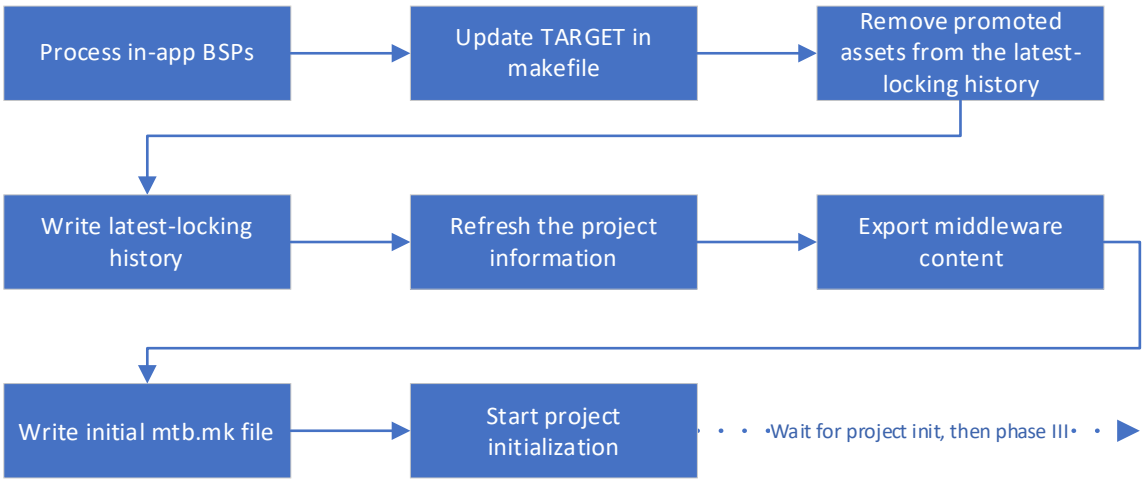
- Submit requests to the ***AssetMgr***. We keep track of BSP assets that were fetched.

- Remove mtb files for all assets in the removed list.
- Write mtbfiles for all assets being added. The direct dependencies are written into the project's "deps" folder and the indirect dependencies are written into the project's "libs" folder.
- Write mtbfiles for all assets being updated. If the asset is being promoted from an indirect to direct, then remove the asset as an indirect.
- If the project is set to use a custom BSP, copy the files from the custom BSP into the Application. Template processing is done on each copied folder (see section 8.1.22.6.3).
- Start the ***AssetMgr***.

8.1.22.6.1.2 Phase II

The primary purpose of the second phase is to adjust the content in the project. For example, moving BSPs that should be "in-app" to the correct location, updating the TARGET in the makefile, handing middleware exported content, and so on. The final step in this phase is to reinitialize the ProjectInfo object. This may require a call to "make get_app_info" so this phase is run as a "background" operation.

Here is a flow chart showing the steps in Phase II.



- For a 3.x project, all local BSPs are moved from the folder they were downloaded into ("libs") to the project's BSP folder ("bsps").

This is also known as “in-app BSP” processing and is described below (8.1.22.6.2).

- Changing the target BSP requires changing the TARGET field in the makefile. For single core applications, the TARGET field is located in the file named Makefile in the common project and application directory. For multi-core applications, the TARGET field is located in the file named common.mk in the application directory.

- Assets that were promoted from indirect to direct are removed from the latest-locking history.

- Write the latest locking history file.

- Refresh the project information by reading the mtbfiles, asset instances, and BSP instances from disk for the project.

- Copy any export files from middleware directories into the project import directory. See section 8.1.22.7 for more information.

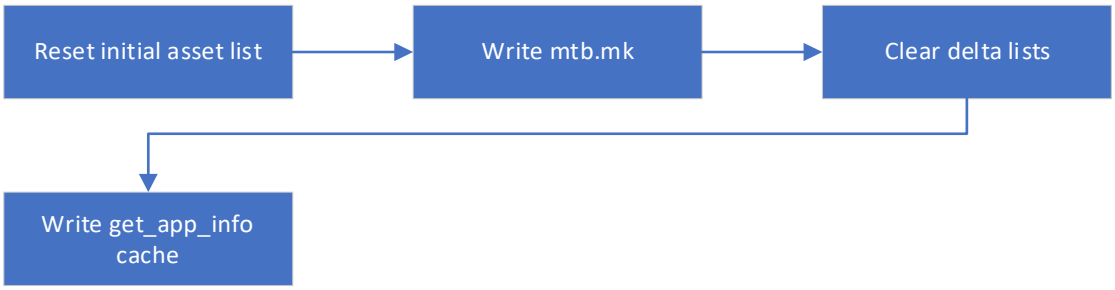
- Write a preliminary version of the mtb.mk file. This is sensitive to the ***AppStructure***. If the ***AppStructure*** is MTB2.X then the mtb.mk file is generated just like in MTB 2.x. If the ***AppStructure*** is MTB3.x then the mtb.mk file is generated with SEARCH_... variables (like MTB2.x) and COMPONENT_MW_... variables as described in the SAS. This is a preliminary version since it does not have a correct list of make targets. Generating the make targets requires an accurate PROGRAM_IDS value and this is obtained from “make get_app_info”. However, we can’t run “make get_app_info” until the initial mtb.mk exists so that there is a pointer to mtb_shared.

- Trigger full project initialization. This will run “make get_app_info”.

8.1.22.6.1.3 Phase III

The primary purpose of the third phase is to update the project with generated information. In particular, the mtb.mk file needs to be updated with the correct list of make targets for launching configurators and tools. All of the operations in this phase are executed in the “foreground”. When phase III is complete the ***commitDone*** signal is emitted.

Here is a flow chart showing the steps in Phase III.



- Reset the initial asset list.
- Write the mtb.mk file. Unlike the first time that this file was written (in phase II), we now have the correct list of PROGRAM_IDS so the make targets for launching configurators is correct.
- Clear the delta lists. This essentially sets the ProjectChangeMgr back to a “blank slate” state and it is ready for the client to start a new session of changes.
- The values from the latest run of “make get_app_info” are in memory in the ***ProjectInfo*** object. Now that all the other processing is done, this data can be cached. Note that if the QueryAPI is ignoring the cache, this step is skipped.

8.1.22.6.2 The “in-app BSP” Steps

When working with an MTB 3.x application structure, a BSP that has a “local” location in the .mtb file is converted into an “in-app” BSP. Before in-app BSP processing is started, the requested BSP has already been downloaded into the project’s “lib” folder (see 8.1.22.6.1.1) but no other processing has been done.

The following actions are taken:

Determine the new folder name for the BSP. This is derived from the BSP’s original name and the user-given name for the new BSP. If the user didn’t provide a name then the default name of “APP_<original name>” is used.

- Construct the new folder name for the BSP. This is derived from the BSP’s original name and the user-given name for the new BSP. If the user didn’t provide a name then the default name of “APP_<original name>” is used.
- Remove the .git folder

- Rename the BSP.mk file if it is using an old-style name. The old-style name is TARGET_<bspname>.mk. Newer BSPs use “bsp.mk”.

- Create .mtbx files. In-app BSPs declare their indirect dependencies via .mtbx files in the BSP folder. As a part of in-app BSP processing, the dependencies of the source BSP are extracted from the manifest database. Each of these dependencies is written as a .mtbx file in the BSP folder (in the BSP_DIR). Latest-locking is performed on these dependencies before they are written.

- Move the cloned folder from the application’s LIBS folder to the BSP_DIR with the new name.

- Copy template data from the project. See 8.1.22.6.3.8.1.21.6.3.

- Update the TARGET in the makefile to use the new BSP name. This only happens if the *original* name of the BSP being processed matches the current value of the TARGET variable in the makefile. The original name is used so that code examples can ship with a valid value in the makefile.

- Remove the associated mtbfile in the “deps” folder.

8.1.22.6.3 BSP Template Processing

A code example can have a “templates” directory that contains custom files (e.g., a custom design.modus) for one or more specific BSPs. For multi-core applications, the “templates” folder is at the application level. There is a folder for each customized BSP underneath the “templates” directory. This directory name is of the form TARGET_bspname. The “templates” directory is searched for any folder matching the active BSP name. If that folder is found, then that folder is recursively copied into the corresponding BSP folder in the project’s BSP directory. All files are overwritten. There are no backups made of any files and there are no warnings generated by this process when a file is overwritten. Existing files that aren’t overwritten remain in the BSP folder.

Note that the folder structure in a template is preserved during the copy. This means that the template data needs to match the expected directory structure in the BSP exactly. In particular, since the design.modus file is expected to be in a “config” subdirectory in BSPs, it should be in a “config” subdirectory in the template folder.

This process happens for local BSPs in 3.x projects -- both manifest-based BSPs and imported BSPs.

The templates folder should have a .cyignore entry to ensure that source code discovery doesn't find these files.

When a code example author includes an updated configuration file for one configurator (e.g., Device Configurator), care should be taken to make sure that all other configuration files that could be impacted also need to be updated. For example, a custom design.modus that removes pins may have an impact on the capsense or seglcd configuration. The QueryAPI will not warn about the potential for configuration files to get out of sync. There is no way for the QueryAPI to know if the specific configuration in one configuration file will impact another configurator.

If a "templates" folder exists but there is no corresponding BSP in the BSPs folder, a warning is generated. This represents a case where the user needs to be informed that the code example author intended the BSP to be updated.

8.1.22.6.4 Exporting Middleware Content

Middleware items can specify content that should be exported to the user's application when the middleware item is included in application. The exportable content is located in a folder named "export" in the middleware item and is meant to be copied into a folder named "import/<middleware_name>" in the user's project. The exported content consists of files that are meant to be edited by users.

The first time that a middleware item is included in the project, the exported content is copied into the import directory. Since this is the first time, the import directory will not exist. This happens as a part of the ***commit*** method in the ***ProjectChangeMgr***.

On subsequent calls to ***ProjectChangeMgr::commit*** we may be trying to export content from middleware to an import directory that already exists. When this happens there are three situations to handle:

- The user may have edited the content
- The middleware being exported might be an updated version of the middleware that was previously copied
- There is no change in the exported content

The action that needs to be taken varies based on the circumstance. The following table summarizes this.

Condition	Action
Neither imported nor exported content have changed since the last export action.	Do nothing.
The imported content has changed but the exported content has not changed since the last export action.	Do nothing. This represents a case where the user had edited the content but the version of the middleware has not changed.
The imported content has not changed but the exported content has changed since the last export action.	Overwrite the existing import directory with the updated middleware content. This represents the case where the middleware has been updated by Infineon.
The imported content and the exported content have changed since the last export action.	Archive the existing import directory then overwrite with the new content. Also, provide a message to the user to indicate what has happened. This represents the case where the user has edited the content that was originally copied but Infineon has also updated the middleware content.

Each time the exported content is copied into the import directory, a checksum file is created in the import directory. It is called "<middleware_name>.cksum". It is a checksum of the contents of the exported content. The value of this checksum is used both to determine if the imported content has changed or if the middleware exported content (that is about to be copied in) has changed.

8.1.22.7 Copying a BSP Folder to the Project

When client code needs to add a "custom" BSP to an application (either via the project-creator or the library-manager tools), there is no associated

git resource so it is not possible to create an mtbfile to point to the BSP in a git repo. Instead of the files being cloned (from the non-existent repo), they are copied from the source to the appropriate target folder during the commit operation.

To copy the files into the project, the target directory must be determined based on the ***AppStructure***. For 2.x projects, the target folder is the root of the project (in accordance with how 2.x tools work). For 3.x projects, the target folder is the BSP_DIR value. Once the target directory is identified, the files are recursively copied from the BSP source folder into the target BSP directory.

Since no actions happen in the project until the ***commit*** call, the call to ***addCustomBsp*** simply stores the source folder name and sets a flag to indicate that there is a folder-based BSP to be copied in. This information is used when creating the change log message and when ***commit*** is called.

8.1.23 BSP Data Model

The BSP Data Model must be able to be created in one of four ways.

- Load an existing BSP from a loaded application (from the ***AppInfo*** object) even for BSPs that are not the active BSP.
- Load an existing BSP given a directory when no application data is loaded into the ***IModusToolboxEnv***.
- Create a new BSP given the id of a BSP template found in the manifest files.
- Create a new BSP given a directory containing a template.

For each of these cases the BSP data model must be able determine dependencies for the BSP and in the case of writing a new BSP, write these dependencies to .mtbx files.

For new BSPs the “requires” information must be extracted from either an existing BSP template or the manifest files and stored in the new BSP. For an existing BSP, this “requires” information must be loaded into the BSP data model.

Finally, the BSP data model must be able to determine the required libraries needed to provide configurator information (personalities and data files) and if not available locally, download the required assets.

The BSP data model must have methods to change the target device. Target device changes are limited to those that use the same device support library. The information that governs this will be stored in the UDD database.

The BSP data model must have methods to change the connectivity device. Connectivity device changes are limited to those that have hardware (e.g. SPI, SDIO, UART, etc.) compatible with the target device. This compatibility information will be stored in the UDD database.

The BSP data model must have methods to manage the “requires” data for the BSP. This means adding and removing this data and writing the new data to the props.json file stored in the BSP directory.

The BSP data model must have the ability to manage target device and connectivity device specific options. The information about what options are available are stored in UDD. The format for these options (on/off choices, one of a set of choices, etc.) will be defined as part of the implementation. The results of these choices must be stored as either COMPONENTs or DEFINEs in the BSP make file.

The specific UDD views required for Target BSPs, Target Devices, and Connectivity Devices will be defined by PR2.

8.1.24 Retrieving Content

In ModusToolbox there are many places where external content is needed. This is the power of ModusToolbox in that only a minimal and necessary system is installed on the local machine and the system can be constantly extended by releasing more content that is retrieved via the internet. To retrieve this content, there are three “manager” objects that manage retrieving this content. The ***DownloadManager*** is responsible for downloading content using the http/https protocol. The ***AssetMgr*** is responsible for downloading git-based assets. The ***AssetMgr*** uses the ***GitJobMgr*** for performing git operations on a git repository. Client code can use the ***GitJobMgr*** directly but it is recommended to use the ***AssetMgr*** when possible since this seamlessly uses the correct content based on the OperatingMode.

For each of these “mangers” there is a single instance of these objects stored in the top level ***ModusToolboxEnv*** object that is passed to any client need these services. This set of “single point” managers to retrieve content is a critical part of the ModusToolbox design.

8.1.24.1 The ***DownloadManager***

The download manager takes download requests via the ***download*** method. A URL is provided as well as a ***QVariant*** data element to be returned to the client when the download request is complete. When the download is complete, the ***downloadComplete*** signal will be emitted. If the download fails, the ***downloadError*** signal is emitted. Only one of these two signals will be emitted.

The download manager automatically handles redirect requests returned for the request. If a redirect response is returned, the download manager requests the resource via the new URL provided via the redirect response. This is completely transparent to the client with the exception of the fact that a redirect signal is emitted with the original URL, the current URL, and the new URL for the request. Since multiple redirects can happen the current URL is the latest URL we have attempted that resulted in the redirect request, and the new URL is the new URL to use.

An important feature of the download manager object is that any number of downloads can be requested and each of these requests are queued. The ***downloadComplete*** (or ***downloadError***) signal will always be called in the order in which the ***download*** method is called to request downloads. So, while the requests will be launched in parallel, when the results of the download request are returned, the signals are only emitted in the order the resources were requested.

Only a specific number of requests are actually launched in parallel and this is managed by the ***QNetworkAccessManager*** class from Qt.

8.1.24.2 The *AssetMgr*

The asset manager knows about the various operating modes supported for the MTBQueryAPI. When git hosted assets are required, the ***AssetMgr*** manages the asset request and launches a series of git operations via the ***GitJobMgr*** to acquire the asset. The ***ProjectChangeMgr*** and the ***ApplicationCreator*** both interact with the ***AssetMgr*** to obtain assets rather than using the ***GitJobMgr*** directly.

The ***AssetMgr*** design is such that a series of assets are requested via the ***fetchAsset*** method. When all desired assets are requested to the ***AssetMgr***, the ***start*** method is called and the assets are processed as a batch. The ***allAssetsComplete*** signal will be emitted when all of the assets are available.

The ***AssetMgr*** emits the standard set of message signals (see 8.1.7).

8.1.24.2.1 Destination

A cloned asset may be stored in one of five locations. These locations are:

- locally – this is generally “libs” directory within the project but should be found by calling ***getLibsDir*** on the ***ProjectInfo*** object.
- shared – this is generally “../mtb_shared” for single core projects or “../../mtb_shared” for multi-core projects but should be found by calling ***getSharedDir*** on the ***ProjectInfo*** object for each project.
- global – this is generally “~/.modustoolbox/global” but should be found by calling ***getGlobalDir*** on the ***ProjectInfo*** object.
- Project-relative – this is when the location field in an mtb file contains no sentinel. In this case the destination is the project root followed by the location string.
- Absolute – this is when the location field contains the ABSOLUTE sentinel. In this case the remaining portion of the location field is considered an absolute path to a location.

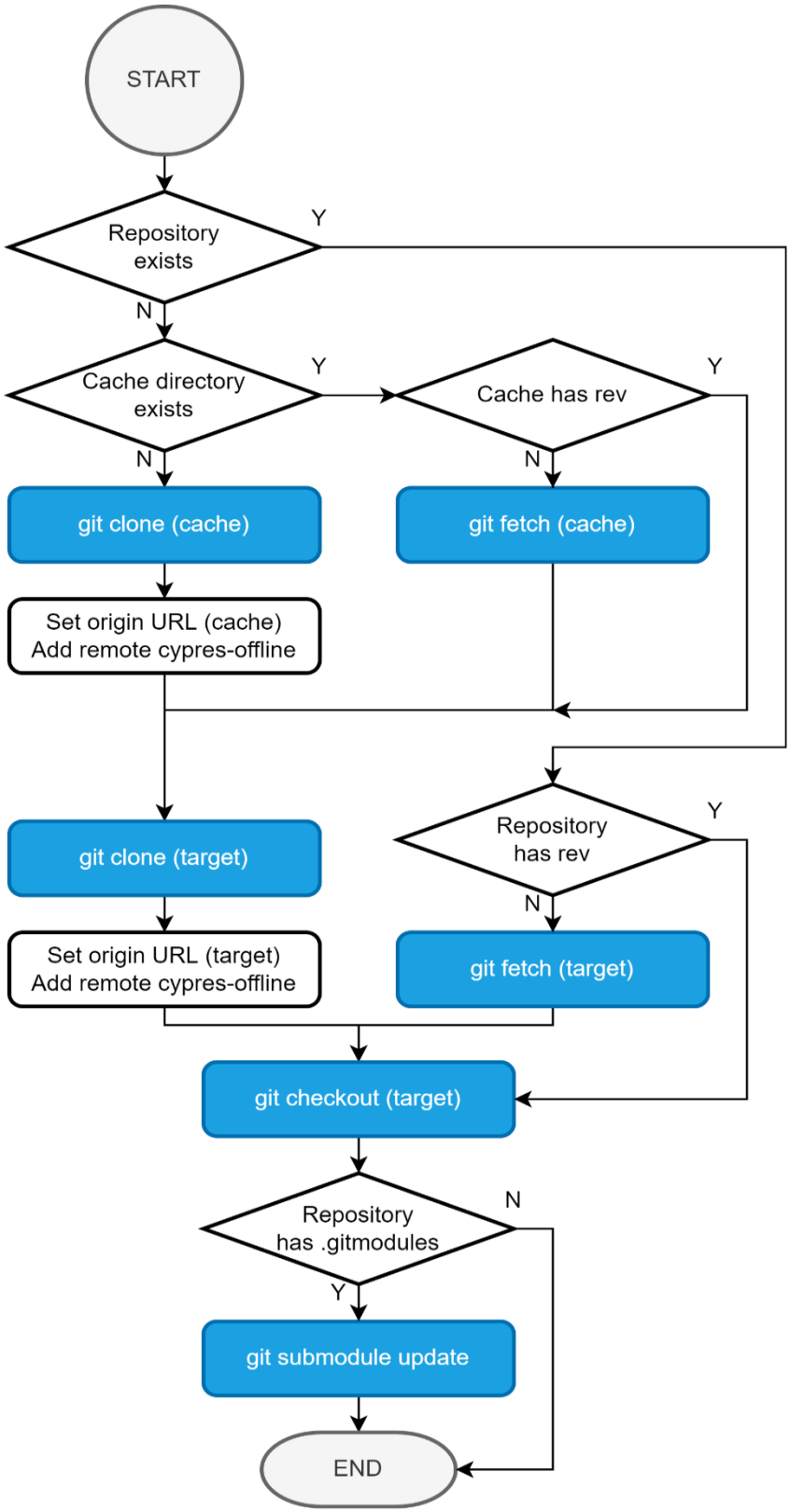
The target location is taken from the ***AssetRequest*** being cloned.

8.1.24.2.2 Modes

The ***AssetMgr*** supports all the ***OperatingMode*** enumerated type modes. The ***AssetMgr*** inherits its mode from the ***IModusToolboxEnv*** environment.

8.1.24.2.2.1 DirectMode

In ***DirectMode***, assets are cloned directly into the current project from the asset repo’s locations (URLs) described in the manifest files. The following flow chart documents the flow for an asset request while in the direct mode. Steps shown in blue are performed by an external ‘git’ subprocess; the others are performed using libgit2.



8.1.24.2.2.2 ***OfflineMode*** (legacy offline mode)

In ***OfflineMode***, asset URLs are transformed into file paths in the offline content and then cloned from there.

This mode uses the same sequence of operations as the ***DirectMode*** flow.

This mode was deprecated with MTB 3.1.

8.1.24.2.2.3 ***CachedMode***

CachedMode was available in ModusToolbox 3.0 and earlier. In ModusToolbox 3.1, ***CachedMode*** is an alias for ***DirectMode***.

8.1.24.2.2.4 ***LocalContent*** mode

Added in ModusToolbox 3.1.

For asset revisions that are available in the local content directory, ***LocalContent*** mode will use the local content directory. Otherwise it performs the same operations as ***DirectMode***.

Local content repositories are stored in a directory derived from the original URL:
<LOCAL_CONTENT_DIR>/git/<URL_HASH>/<REPO_NAME>.

LOCAL_CONTENT_DIR is ~/.modustoolbox/lcs by default.

URL_HASH is computed from the prefix of the original URL. The prefix includes the URL up to but not including the repository name The trailing slash, query, and fragment should be excluded. IDNs should be converted to punycode. Reserved characters (such as spaces) and non-ASCII characters should be %-encoded with UTF-8 encoding. The URL prefix should be converted to lower-case. The URL_HASH is the SHA1 hash of the transformed URL prefix expressed as 40 lowercase hexadecimal digits.

For example, if the original URL is https://github.com/Infineon/mtb-example-ota-mqtt.git, the URL prefix is https://github.com/infineon so the URL hash is eab62c3498129169ee1e1667803248325442a315. The repo name is mtb-example-ota-mqtt.git (“.git” suffix should be added). The local repository directory would be ~/.modustoolbox/lcs/git/eab62c3498129169ee1e1667803248325442a315/mtb-example-ota-mqtt.git.

The name mapping is intentionally different from the legacy offline directory's name mapping described inprevious versions of this IROS document. The hash has a fixed length so that URLs with long paths will not exceed the Windows path length limit.

8.1.24.2.2.5 ***LocalContentManager*** mode

Added in ModusToolbox 3.2.

This mode is introduced for creating a local copy of content. It is designed for the utility lcs-manager-cli, and can also be used by other programs to create copies of content. Data loading in this mode is performed in the same way as in the ***DirectMode***, but the processing of loaded data follows the next logic: when merging identical assets downloaded from remote manifests and packages (PACKs and/or EAP), priority is given to assets downloaded from remote manifests; in other cases, the logic of data merging is the same as in ***DirectMode***. Also, all assets downloaded from PACK's local files are deleted (as they are not needed to form local content).

Such implementation allows for creating a local copy of content that functions together with PACKs/EAP (including preserving elements from packages downloaded from remote sources) and supports working with placeholder entries.

The current implementation leads to one limitation - when working with manifests downloaded from local files, the priority rule does not apply, which means that such manifests cannot contain placeholder entries. If necessary, this feature can be added in the future.

8.1.24.2.3 Cloning from ModusToolbox Pack Repositories

ModusToolbox packs contain manifest files that can add additional content to ModusToolbox. It is expected that most of the time, these manifest files refer to content that is stored directly in the ModusToolbox Pack. The content is referenced using the "mtb" protocol in the content URL (see 8.1.20.2.1) and the repo is cloned into the project using a Direct Clone.

8.1.24.2.4 Retrieving Existing Content

In many cases the content that ModusToolbox is retrieving already exists on disk. In these cases, the default behavior is for ModusToolbox to skip the fetch command if the asset if its revision ID is already in the local repository (in other words, if library is already up to date). The exception to the above is that if it is a "latest-vX" tag the environment variable

MTB_GETLIBS_FORCE_UPDATE is set to “true” or “1”. In this case the fetch is always performed.

Also, by default, the git checkout command is run without the “--force" flag. This means that if the destination directory has been modified, the checkout command may fail. If the MTB_GETLIBS_FORCE_UPDATE variable is set to “1” or “true”, then the “--force" flag is added to the git checkout command.

8.1.24.3 GitJobMgr

The ***GitJobMgr*** is a system that allows clients to queue up git jobs. The base class for git jobs is the ***GitJob*** class and there are derived classes for specific git operations. These are ***GitCloneJob***, ***GitConfigJob***, ***GitFetchJob***, ***GitSetRemoteJob***, ***GitSubmoduleUpdateJob***, ***GitCheckoutJob***, and ***GitCheckCommitJob***, which implement git clone, git config, git fetch, git set remote, git submodule update, and git revparse --verify respectively.

When possible, git jobs are launched concurrently up the maximum number of concurrent jobs allowed. This can be set by calling the method ***setMaxConcurrentCount*** method. This maximum concurrent count installed by default is ***QThread::idealThreadCount() – 1***.

Since git jobs are queued and are launched based when resources are available, there are two signals that are emitted to track the life cycle of a git job. The ***jobStarted*** signal is emitted when the git command is actually run. The ***jobComplete*** signal is emitted when the job is complete. Note, that if there are conditions that exist that mean the git job cannot be started, only the ***jobComplete*** signal is emitted.

The ***GitJobMgr*** emits the ***startingGitCommand*** signal as each git job is started. This ***GitJobMgr*** emits the ***finishedGitCommand*** signal as each git job is completed. These two signals are intended for providing debug information about what is run and when it finished. It is not meant for code to use to track the state of a git job. The ***GitJob*** signals should be used for this purpose.

The ***GitJobMgr*** emits the ***streamStdOut*** and ***streamStdErr*** signals to relay output from the git commands to client applications. Git native behavior prints many non-error messages through the stderr channel. If these messages are streamed through the QueryAPI on the stderr channel then clients will not know which messages are error text and which are non-error text. Because of this, all stderr streamed from git is

redirected to the streamStdOut QueryAPI signal instead of the streamStderr signal.

Many of the git operations support a –quiet flag. This flag is not exposed directly to clients of the ***GitJobMgr*** but instead a ***GitQuietPolicy*** enumeration is provided. This has three values:

- SILENT – all git commands are run with --quiet
- USER – clone and checkout are run without --quiet, everything else uses --quiet
- LOUD – no git commands are run with --quiet

8.1.24.3.1 Using INSTEADOF

All git operations that access the git server can use the GIT_INSTEADOF central setting. This setting consists of an old server string and a new server string. If these values are non-empty strings, then the “-c url.<newserver>.insteadof=<oldserver>” command line argument is added to each command.

8.1.24.4 GlobalContentMgr

The ***GlobalContentMgr*** is a class that simplifies loading and accessing content in the global space. The ***GlobalContentMgr*** uses the ***AssetMgr*** to acquire assets.

Clients gain a ***GlobalContentMgr*** object by calling ***getGlobalContentMgr*** on the MTB environment object. This method uses a singleton pattern. It created a ***GlobalContentMgr*** object on the first call. Subsequent calls receive the same object. Each time that ***getGlobalContentMgr*** is called, the objects ***OperatingMode*** is updated to be the same as the current ***OperatingMode*** of the environment.

8.1.24.5 Runner

The ***Runner*** class is a class that executes external programs. This should not be confused with the ***ProgramRunner*** class associated with Tools in the ***ToolsDb***. The actual execution is performed using the Qt ***QProcess*** class. The ***Runner*** class has the options of combining standard output and standard error to a single stream (standard output). This class can also stream standard output and standard error via a set of signals as output is produced, or this class can store the output until the process is complete. When the process is complete, the exit code is available. In addition to standard output and standard error, there is a signal that is

emitted when the process is finished. A single argument boolean to this finished signal indicates success or failure. Note, that the signal ***finish*** is emitted in all cases, even if something is invalid and the requested command cannot be started.

8.1.25 Centralized Settings

The objectives of this feature are:

- Improve ease of use for our users by allowing them to control some aspects of the MTB environment without setting environment variables
- Provide a centralized API for QueryAPI clients to access the MTB environment related settings.

The global settings will be stored in ~/.modustoolbox/settings.json.

Some settings may be configured in multiple locations. Unless otherwise specified, the following precedence order (highest to lowest) should be used.

1. Command line argument (if applicable; only available for select settings such as MTB_TOOLS_DIR)
2. get_app_info (if applicable)
3. Environment variable (if applicable)
4. Global settings file (~/.modustoolbox/settings.json)

The QueryAPI enforces this priority order so clients do not need to be aware of it.

8.1.25.1 Settings Usage Model

The use model for settings is that clients request a Settings object from the MTB environment and use that object to get information about what settings are provided by MTB, information about particular settings, and set new values on settings.

Most clients will only use the Settings object to read values. There is a special client called the Settings Tool (see below) that also makes use of the ability to set values through the Settings object. While all clients *can* use the write APIs, we expect that only the Settings Tool does this.

8.1.25.1.1 Reading Settings

Clients use the Settings API to access all information about a setting. In most cases clients are only interested in the value of a setting and they call the getValue() method on the Settings API. In some cases, clients need more detailed information about a setting. In that case they can use the getSetting() API to get a Setting object. The Setting object has the following properties:

- name – a user-facing name for the setting
- id – a unique name for the setting
- setting type – the type (TEXT, BOOLEAN, CHOICE, COMPOSITE)
- JSON name – the JSON property name used to serialize this setting to the settings file.
- default value – the default value for this setting
- presets – a list of preset values (required for CHOICE types, optional for others)
- description – a user-facing description of this setting
- environment variables – a list of environment variables that affect this setting value
- environment variable lock description – a user-facing text description that describes why a variable is locked due to an environment variable
- readonly – a boolean property that indicates if the setting is readonly or not
- hidden – a boolean property that indicates if the setting should be shown to the user or not (in a UI)
- value – the current value of the setting
- setting source – the source of the current value of the setting (DEFAULT, FILE, ENV_VAR)

There are some convenience methods on the Settings API that allow getting the setting source or the setting presets. For all other properties,

the client needs to get the Setting object itself and use that object's getter methods.

The Setting object returned by the Settings::getSetting() API should be considered a copy of the actual setting in memory. Clients should not attempt to modify it (via the setValue() API) nor should they hold onto a reference to this object in expectation that it will be updated when the setting value changes.

8.1.25.1.2 Writing Settings

In cases where a client needs to update a setting value (usually the Settings Tool), the Settings::setValue() API should be used. At this point in time there is a Setting::setValue() API (note Setting vs Settings) but that should not be used since it doesn't correctly synchronize the setting with the settings file. In future versions of the QueryAPI the Setting::setValue() method will be made private.

When the setValue() API is called, the Settings object attempts to change the current value of the the identified setting. This will fail if the setting is a readonly setting or if the new value fails validation.

The validity of a value depends on the setting type. For a BOOLEAN setting, the value must be a variation of true/false or 1/0. For a CHOICE setting, the value must match one of the values in the presets list.

If a setting value is successfully changed, the in-memory model is updated and the new value is immediately written out to the settings file.

8.1.25.2 Settings Storage

The settings are stored in a JSON (ECMA-404/STD-90) file. JSON files are always UTF-8 without BOM (as required by the JSON specification).

The JSON file should contain an object. Each of the object's keys represents a setting group. Each group may contain values and group objects. Values may have string, number, or boolean types.

```
{
  "group1": {
    "name1": "This is a string value",
    "name2": true,
    "nestedGroup": {
      "nestedValue": 12.345
    }
  }
}
```

There may be multiple tools that can write to the settings file. The settings file will need locking/merging to avoid corruption or lost changes.

8.1.25.2.1 Reading the settings file

As long as all writers use atomic updates, reading the file should always be safe.

The query API implementation may cache the settings in memory. When a client requests the value of a setting, if that setting was obtained from the settings file (as opposed to an environment variable or other source), then the QueryAPI will check the timestamp of the settings file on disk to see if it needs to refresh the in-memory value before returning it to the user.

8.1.25.2.2 Writing the settings file

1. Lock the settings file by creating a "<filename>.lock" file in the same directory (e.g. settings.json.lock). If the file already exists, wait and retry. If the timeout is exceeded, return an error.

2. If the settings file modification time has changed since the last time it was read, re-read the file and update all settings values that were obtained from the settings file previously.

3. Write the new contents to a temporary file.

4. Atomically move the temporary file to replace the settings file (may require retry on Windows).

5. Remove the lock file.

8.1.25.3 Synchronization Signal

It is possible (even likely) that while a client is running, another client will do something that alters the settings file on disk. When this happens, it is necessary that the running client is made aware so that it doesn’t rely on out of date setting values.

The Settings object has a file watcher set up on the settings JSON file. Whenever that file changes, the settings file is reread, the in-memory value for all settings is updated and the Settings object emits a signal that indicates that the values have changed. Clients can connect to that signal so that they can respond appropriately when the settings have changed.

For example, project-creator and library-manager should reload the ManifestDb because the user may have switched from online to local mode.

Currently, the signal simply states that some value changed. It doesn't contain any information about what setting changed or what the previous value was. This means that when clients respond to the change signal, they should assume the worst case. If the need arises, then in future versions of the QueryAPI we can update this signal to provide details about what changed and how.

8.1.25.4 Settings Initialization

As a part of initializing the ModusToolbox Environment, the QueryAPI also creates and initializes the Settings object in the environment. Initialization of the Settings object involves adding the list of known settings to the internal database. These settings are:

- operating_mode – online or local
- proxy_mode – direct or manual
- proxy_server – the proxy server to use
- lcs_path – a custom path to the LCS directory
- information_level – the "loudness" level for git commands
- manifestdb_system_url – the optional location for the remote manifestdb super manifest
- git_insteadof – values to use for the git "insteadof" setting (in support of the China server)
- enabled_eap – the id for the enabled EAP (if any). This is a CHOICE setting and is initialized with the list of installed (but not necessarily enabled) EAPs on the user's system.
- global_path – a custom path to the global directory
- manifest_loc_path – a custom path for the manifest.loc file
- tools_path – a readonly value that shows the default tools path

As these settings are added to the settings database, the values are read from the settings file.

8.1.25.5 Settings Tool

A tool, mtb-settings, is provided that allows users to easily view and modify any of the settings in the Settings object. This tool has both a GUI and a CLI interface.

Previous versions of MTB did not have a separate settings tool and instead had custom code in each client that provided a user interface for viewing and modifying the settings. This had two major problems. First of all, it was awkward and unintutive for users to make a settings change. For example, if the user wanted to change from online to local mode, they had to launch project-creator or library-manager just to access the setting, make the change, then close the tool. In otherwords, they had to launch the tool solely for the purpose of changing the setting. Secondly, if the user had two clients open, changing settings in one client did not notify the other tool that setting values had changed and therefore the two clients could be out of sync.

Providing an independent Settings Tool handles both of these problems. Now, all clients have a menu item to launch the Settings Tool. Also, the Settings Tool can be launched from the ModusToolbox Dashboard and it also has its own Start Menu entry on Windows. This creates a more intuitive access point for the ModusToolbox Environment settings than previously.

Another benefit of providing the tool is that we now provide a way to view and set settings from the command line. This makes internal development and CI easier. It also helps our customers who also have automated development environments.

8.1.25.5.1 GUI Interface

The Settings Tool GUI provides a list of all the settings. For each setting it shows the current value and user-facing documentation for that setting. If the setting is locked due to an environment variable, that is indicated. If the setting is not locked, then the user can update the value. The GUI uses appropriate GUI elements based on the type of the setting.

8.1.25.5.2 CLI Interface

The Setting Tool CLI provides command line options to view and set settings.

The options are:

- -h, --help – Show help information

- -v, --version – show the version number.
- --verbose – provide detailed output (most useful with --list-json)
- --list-json – list all settings in JSON format.
- --list-make – list all settings as name=value in make syntax.
- --conditional – create the make listing with conditional assignments.
- --value <name> – print the value of a setting.
- --set <name=value> – set a setting using name=value syntax as long as the value is valid.
- --set-force <name=value> – set a setting using name=value syntax. Force the setting even if it is not valid. This will still fail if the value is currently locked due to being set by an environment variable.
- --reset <name> – reset a setting to the default value.
- --reset-all – reset all settings to the default values.

If the command encounters an error when executing then it returns a non-zero exit code and error messages are printed to stderr. The following are errors:

- Attempt to set a boolean setting to a non-boolean convertible value.
- Attempt to set a dropdown setting to a value that doesn't exist in the choices.
- Attempt to set or reset a setting where the associated environment variable is set.
- Attempt to set an unknown setting.
- Pass bad command line arguments.

All the commands that require a setting name use the JSON property name as found in the settings.json file.

8.1.25.6 Settings API

IModusToolboxEnv has a function to obtain the settings manager from the environment. The settings manager is owned by the environment and has the same lifetime as the environment.

A client uses the following static method to get a pointer to the settings manager.

std::shared_ptr<ISettings> &IModusToolboxEnv::getSettings();

The ISettings object provides the following methods to access the settings:

```
class ISettings
{
  public:
    /**
     * @brief Gets a setting value using the SettingNames enumeration
     * @param name setting to lookup
     * @return setting value, or an invalid QVariant if the setting
doesn’t exist.
     */
    virtual QVariant getValue(SettingNames name) = 0;

    /**
     * @brief Gets a setting using the SettingNames enumeration
     * @param name setting to lookup
     * @return setting object or an invalid Setting if the setting
doesn't exist.
     */
    virtual Setting getSetting(SettingNames name) = 0;

    /**
     * @brief Returns true if the setting is configured.
     * @param name setting enum
     * @return true if this is a known setting
     */
    virtual bool contains(SettingNames name) = 0;

    /**
     * @brief Sets a setting value.
     * @param name setting enumeration
     * @param value setting value
     * @return error
     */
    virtual bool setValue(SettingNames name, const QVariant &value) = 0;

    /**
     * @brief Gets a source of setting value
     * @param name setting to lookup
      * @return source of setting value.
    */
    virtual settingSource getSource(SettingNames name) override;

    /**
     * @brief Get a list of preset values for the given setting.
     * @param name The setting to return preset values for.
```

```
     * @returns a list of preset values.
     */
    virtual const QStringList getPresets(SettingNames name) const = 0;

    /**
     * @brief Get all the existing settings.
     * @return a list of all the settings.
     */
    QVector<Setting> getAllSettings() const override;
};

class Setting
{
public:
    QString getName();
    QVariant getValue();
    bool isReadonly();
}
```

Currently, the Settings API only supports the known settings as described above. For future versions of this API, we will support client-provided settings. When these are supported then the clients will be able to refer to settings using a string. The names will be qualified by the group name, e.g. "group1.nestedGroup.nestedValue". This is referred to as ‘dotted notation’.

8.1.26 Local content storage

The objective of this feature is to solve the problem of allowing users with limited and restricted internet access to more fully use MTB. This feature is not trying to improve performance for either manifest loading, project-creation, "getlibs", or any other aspect of MTB.

8.1.26.1 Overview and Requirements

In MTB 3.0 (and earlier) users could use offline content rather than the online data. This required users to download and install offline content bundles that were created by Infineon from time to time. The process of creating offline content bundles turned out to be labor intensive and therefore, to date, only a few offline content bundles were created.

MTB 3.1 supports a different method for users to work “offline”. In a nutshell, MTB 3.1 includes a tool to allow users to create their own local content store of assets they are interested in and then direct the ModusToolbox tools to use that local content. Users will be able to decide if they want the entire database (approximately 5GB of data) or a subset of data related to a set of BSPs that they define. After acquiring local content, users will explicitly set whether they are using the local content or online content. The rest of the MTB system will use what the user selected.

Here is a summary of the requirements for Local Content Storage:

- Users create a local content store with a command line tool (lcs-manager-cli)
- Users explicitly tell MTB if they want to use local content or online content in one of the GUIs (project-creator, library-manager, device-configurator, etc.)
- From a client point of view, everything works just as it did in MTB 3.0.
- The QueryAPI automatically loads the correct database (based on user settings)
- The QueryAPI provides a notification method so that clients can know if there is newer content available
- The QueryAPI uses a simplified git flow (clients won't see this but it may end up in a slight performance improvement)

8.1.26.2 Local Content vs Offline

As mentioned above, earlier versions of ModusToolbox supported the concept of “offline content” and an associated “offline mode”. The idea of “offline” in these earlier versions was that users were physically disconnected from the network (or at least from github.com). In some cases, the ModusToolbox tools would try to detect this disconnected state and move users to “offline mode” automatically.

The Local Content Storage (LCS) feature in ModusToolbox 3.1 should not be viewed as an improved or renamed “offline mode”. Instead, it is a different way of solving the same problem that “offline mode” was trying to solve. It is incorrect and confusing to refer to LCS as “offline mode” because there are some significant differences between these.

The primary difference is that users create and own their own local content storage. That is, they explicitly run a tool (lcs-manager-cli) to obtain local content and update it when they want to. This local content is defined as a local copy of remote resources. Once it is created, it is its own database that is distinct from the “normal” online manifest database. In the obsolete “offline” model, there was still just one database, we just supplied an alternate location for resources.

Another difference is that being in “local content” mode does not imply that the user is physically disconnected. There is no relationship between “local content” and internet connectivity. Being in “local content” mode simply means that the user has requested to use content from their local storage. The reason that they are doing this is irrelevant. To be consistent with this new way of thinking, none of the ModusToolbox tools will attempt to detect a user’s actual connection status and then make a corresponding change in their mode.

For MTB 3.1, all references to “offline mode” have been removed from user-facing documentation and none of the ModusToolbox tools use the OFFLINE_MODE OperatingMode as defined in the QueryAPI.

8.1.26.3 User Settings

The following settings are related to the LCS.

mode: The "mode" is set to either “direct” or “local” (OperatingMode::DIRECT and OperatingMode::LOCAL in the QueryAPI). This setting controls which database is loaded and used by the QueryAPI.

In general, the DIRECT mode includes all data from the remote manifest, Tech Packs, enabled Early Access Packs, and manifest.loc while the LOCAL mode includes data from the LCS, local data in Tech Packs, local data in enabled Early Access Packs, and local data in manifest.loc. See 8.1.26.4 for more details about loading the data in different modes.

watchlist: The "watchlist" is a list of reponames that the QueryAPI "watches". That is, when looking for updated content, every asset on the watchlist is considered. Each of these assets has a set of related assets that is created by looking at the dependencies of the asset (and all the dependencies of those dependencies, etc.) and all the other items that are compatible with that asset. For example, for a BSP on the watchlist, we look for all code examples and middleware that has matching capabilities. The set also includes all dependencies of the compatible items.

This information is used by the LCS in two ways. First, when downloading (or updating) the LCS content, each asset from the watchlist, and all related assets, are added to the LCS. Second, when the QueryAPI is checking for new versions of items, each asset on the watchlist is checked. This check is for new versions of existing items as well as new items (e.g., new code examples) that are part of the asset's related asset set. For ModusToolbox 3.1 the watchlist is not implemented using the UserSettings feature. It is implemented as a custom file in the LocalContentStorage class.

lcs_location:

This setting allows the user to change the root of the LCS data. By default it is in ~/.modustoolbox/lcs.

8.1.26.4 Loading the ManifestDb

The way that clients request for the ManifestDb to be loaded has not changed. That is, if the MANIFEST_DB load flag is specified in the call to loadEnvironment(), the QueryAPI will load the database. After the loadEnvironment() completes successfully, the client will use the existing getManifestDb() API to access the database exactly the same way that it is done now.

In the QueryAPI implementation of the loader, we will check the user setting "mode" to determine which data to load and how to generate warnings. If the "mode" is DIRECT, then everything is loaded the same way that it was for MTB 3.0. If the mode is LOCAL then instead of using the default remote manifest (or CyRemoteManifestOverride), we will find the supermanifest and manifests located in the LCS (using the "lcs_location" setting) and use those. We will still attempt to load the local data from Tech Packs, enabled Early Access Packs, and manifest.loc files.

To be more explicit, here is exactly what is loaded in each mode.

8.1.26.4.1 DIRECT mode

The system super-manifest is loaded as normal. We use the default super-manifest URL (on Infineon) or use the setting of CyRemoteManifestOverride. If there are any errors accessing the super-manifest file or any of the manifest files referenced by that super-manifest, we report the access errors.

The Tech Packs are loaded as normal. We look for a super-manifest file in each Tech Pack. If it exists, we read it and then process each manifest file that it refers to. All access errors are reported. It is valid for a Tech Pack to not have a super-manifest file. That is not reported as an error.

If an enabled Early Access Pack exists, process it the same way as we process Tech Packs.

The manifest.loc is processed as normal. The manifest.loc contains a list of super-manifest files. For each super-manifest file, we attempt to read it and then process each manifest file that it refers to. All access errors are reported.

8.1.26.4.2 LOCAL mode

Instead of using the normal system super-manifest, we use the super-manifest file located in the LCS. If there is no super-manifest file in the LCS then an informational message is generated that states that the user has selected LOCAL mode but there is no LCS data. This message recommends to the user that they use lcs-manager-cli to update the LCS data. If there is a super-manifest file in the LCS, then this file is processed as normal for a super-manifest file. The manifest files referred to in this super-manifest file will be local URLs (i.e., file://) that refer to the generated manifest files in the LCS. There should not be any errors accessing these files. If there are errors accessing these files then that indicates a problem with the LCS data itself and an appropriate error message is generated.

For each Tech Pack, we look for a super-manifest in the root. If there is no super-manifest file then the Tech Pack is skipped. If the file exists then we process it. The super-manifest file contains links to manifest files. If it's a local manifest file then process it. For each asset from the manifest file we check to see if that asset is already in the LCS, any version. If not, then we add it to the manifest db. If there is at least some version of the asset in the LCS then we check for the particular version we want to add. If it is not there, we skip it. If the version we are processing is already in the manifest db (due to being in the LCS) then we use the priority to decide if we override it or skip it.

If it's a remote manifest file, print an INFO message that says we're skipping it. Tell the user that if they want to include that data, they need to update their LCS with lcs-manager-cli.

If an enabled Early Access Pack exists, process it the same way that we process Tech Packs.

The manifest.loc file is processed as normal. This file contains a list of super-manifest files. For each super-manifest file, we attempt to read it and then process each manifest file that it refers to. This is done for both local and remote URLS. All access errors are reported. When processing assets from these manifest files, if any version of the asset already exists in the manifest database, an appropriate informational message is printed and the asset is skipped. If no versions of the asset already exist in the manifest database, it is added.

8.1.26.5 Notifications of New Content

In ModusToolbox 3.1 there is no automatic detection of new content for the LCS. Users must manually run the lcs-manager-cli command with the correct flags to check for updates.

Since many users will be in LCS mode due to limited or non-existent internet connection, running the notification check automatically will frequently fail. Presumably, users with limited or non-existent connectivity know that they are in this state and know how to make temporary adjustments to their connectivity before running the notification check.

For future versions of ModusToolbox we are considering a way for clients such as library-manager to use the QueryAPI to check for updates so that they can notify users when their LCS can be updated. We have not adequately defined the requirements or the use cases for this feature yet. Also, there are complications about how to make a meaningful check for updates when the user has limited connectivity.

8.1.26.6 Acquiring Assets

The AssetMgr is the part of the QueryAPI that is responsible for acquiring assets for an application. The AssetMgr is described in 8.1.24.2.

The only impact that the OperatingMode has on this is where the "clone" or "pull" is coming from. When the environment is in LOCAL mode then the URL is converted to a file path located in the LCS directory. See 8.1.24.2.2.4 for details about URL conversion.

If the request reponame/version is not in the database, then the AssetMgr should attempt to download the resource with the given URL. This should be done without any URL conversion regardless of the mode.

8.1.26.7 URLs in Manifest Files

In MTB 3.0 (and earlier) manifest files contain URLs to the remote assets. This will still be the case in MTB 3.1. This includes manifest files in the LCS (see LCS Data Model). Essentially, the remote URL is considered the canonical URL. It is only during the fetch operation that the URL is converted to a file path in the LCS (see Acquiring Assets).

8.1.26.8 Updating the Manifest Database

The lcs-manager-cli (see 8.2.5) needs to be able to update the manifest database. It also needs to be able to write out manifest files that represent the database. In order to do this the ManifestDb class will be modified to include a writeManifestFiles() method. This method takes the root directory as an argument. The write method will create a

supermanifest.xml file and then manifest files for Code Examples, BSPs, and Middleware, a dependency manifest file, and a capabilities description file. The output of writing the database will be a single xml file for each of the super-manifest file, app, bsp, and middleware manifests, a single xml file for the dependencies, and a single json file for the capabilities description data. In the current manifest XML files there are situations where global data differs between different files that contain different versions of the same item. In order to preserve this information, new item-level attributes or elements will be included in the XML. This essentially makes the global data for a manifest group redundant in the LCS manifest files. The manifest parser will be updated to handle these new XML attributes and elements but these will be undocumented attributes and elements. They are only intended for use by the LCS. We will need to develop a conflict strategy for the unexpected case where global data is provided and it conflicts with the item data.

8.1.26.9 Constructing the LCS

Data is added to the LCS using the lcs-manager-cli. The user specifies if they want all the remote content copied into the LCS or just a subset as described by using assets (usually BSPs) as "trailheads". Either way results in a list of assets that must be added to the LCS. Assets are added to the LCS by doing a git clone from the remote server into the LCS location. By default, this is a "bare" clone. However, for git repos that contain submodules, it is necessary to do a regular (i.e., non-bare) clone of the repo in order to download the submodules.

If an asset being acquired comes from a Tech Pack or manifest.loc, then we check to see if the asset is from a local URL or from a remote URL. Assets from local URLs are not included in the LCS data but assets from remote URLs are.

Assets from Early Access Packs are never included in the LCS data.

8.1.26.10 Locking the LCS

When doing either the update operation or the add operation, the QueryAPI must lock the LCS. If there are any other clients attempting to use the LCS at that time, those operations should fail or block. This is especially important for a long-running process like updating all data.

The locking mechanism is implemented in the LocalContentStorage class by providing a “lockLcs()” and “unlockLcs()” API. The implementation for these APIs uses QLockFile. In most cases QLockFile can handle stale

locks but in the case where it can’t, clients should present a message to the user indicating that the user should manually delete the lock file.

8.1.26.11 Canceling and resuming LCS operations

Canceling a 'git clone' operation should result in no changes to LCS. To achieve this, lcs-manager-cli should create new clones in a temporary directory and then move them into the final location once the clone is complete.

Canceling a 'git fetch' operation should maintain repository consistency. The lcs-manager-cli should send SIGINT to the 'git fetch' subprocess trees and wait for the subprocesses to exit.

Git clone/fetch operations cannot be effectively resumed because the Git protocol transfers objects in monolithic packs so interruption results in all of the data being discarded. However, lcs-manager-cli may be able to implement coarse-grained resumption by skipping repositories that are already up to date. For asset repositories, this means that lcs-manager-cli must have a way to determine whether a repository is up to date using only the local repository and the manifest data. lcs-manager-cli should update the index file(s) after fetching each repository. For submodule repositories, lcs-manager-cli should only re-fetch the repository if at least one submodule revision requested by a super-repository does not exist in the local submodule repository.

8.1.26.12 LCS Data Model and API

This is an initial draft of the LocalContentStorage class. This is presented here only for review and initial discussion. The actual source code in the QueryAPI code base is the definitive definition.

```
/**
 * @brief The LocalContentStorage class is used to manage the LCS
data. This is not necessary for most clients
 * since they will simply use the standard ManifestDb that is
loaded using the existing loadEnvironment()
 * method. Behind the sceenes, the ModusToolboxEnv will honor the
user's selected mode (direct/local) and will
 * load the correct set of data. This is transparent to the
client and does not use the LocalContentStorage class.
 *
 * The LocalContentStorage class is used when the LCS data needs
to be updated or when it needs to be compared
 * to another database. That is, when lcs-manager-cli is adding
(or updating) new content in the LCS or when the
 * ModusToolboxEnv is checking for new content to give clients
notifications.
```

```
 */

class LocalContentStorage
{
  public:
  LocalContentStorage(QueryAPIErrorList& errorList);

  // @brief Load the manifest db from the LCS data. After the
load is called
  // the "loadComplete" signal is emitted. This method looks at
the user settings to
  // find the LCS root.
  void loadLcsData();

  // @brief Get the manifest db for the LCS data.
  // @return the wrapped manifest db. This can be nullptr. This
will happen if the load method
  // hasn't been called yet or there were fatal errors while
trying to load the LCS data.
  std::shared_ptr<const ManifestDb> getManifestDb();

  // Write the current manifest db (in memory) out to the super-
manifest and manifest
  // files for the LCS
  bool writeManifest() const;

  // Add or update the given group in the LCS data.
  bool addGroup(ManfiestGroup group);

  signal:

  // @brief This signal is emitted with the LCS load is complete.
Any errors encountered during
  // the load are added to the mErrorList.
  // @param success true if loading was successful, false
otherwise.
  loadComplete(bool success);

  private:
  bool mLoading; // This is used as a "lock" to make sure that we
don't try operations while loading
  std::shared_ptr<ManifestDb> mManifestDb;
  QueryAPIErrorList& mErrorList;
}
```

8.1.27 Device Support Information

Many tools in ModusToolbox require information about the DeviceDb and information about assets that are used to support device configuration. Determining the correct assets and versions for the DeviceDb and Device

Support Libraries depends on several factors: an override flag, the presence of an Early Access Pack, the presence of explicit dependencies in a project, and so on. The QueryAPI contains the logic to determine the correct device support information assets and exposes this logic through two APIs. The first is on ***AppInfo*** and it meant to be used by clients that are running in the context of a MTB application. The second is on ***IModusToolboxEnv*** and is meant to be used by clients who are not in an application. Both APIs return a ***DeviceFilesPaths*** that holds the information about the DeviceDb and DSLs.

8.1.27.1 Search algorithm

The algorithm for finding the correct files can be broken into two main scenarios. Either there is a known design file or not. This file doesn't have to exist. For example, most of the time when starting the device-configurator, a design.modus file is specified. When starting the library-manager, there is never a design file.

If there is a design file, there are different rules for if the design file is in an application, in a 3.x BSP structure, 2.x BSP structure, or completely independent (aka as "in Timbuktu").

A second dimension is whether or not the tool that is running is in the context of an MTB application or not.

Given these aspects, these are the following situations and the rules for discovering the correct DeviceDb and DSLs. Most of these situations rely on a common algorithm to find a DeviceDb or DSL based on MTB/MTBX files and another common algorithm to identify the default DeviceDb. Both of these algorithms are described after this list of situations.

Situation1: A design file is provided, the executing tool is using a project, the design file is in the project folder (i.e., not shared)

- In this case there are MTB files
- Use the MTB files to find the DeviceDb
- If MTB files don't identify a DeviceDb, fall back to the default DeviceDb
- Use the MTB files to find the DSL(s)

Situation 2: A design file is provided, the executing tool is not using a project, the design file is in a 3.x BSP structure

- In this case there are MTBX files
- Use the MTBX files to find the DSL(s)
- Use dependencies from the DSL(s) to find the DeviceDb
- If the DSL dependencies don't identify the DeviceDb, fall back to the default DeviceDb

Situation 3: A design file is provided, the executing tool is not using a project, the design file is in a 2.x BSP structure

- In this case there are no MTB/MTBX files
- Find the BSP name from the TARGET_ directory name
- Look up the BSP name in the ManifestDb to find the BSP Item
- Use the BSP Item dependencies to find the DSL (should be just one)
- If no DSL dependencies are found, check to see if the MTB_DEVICESUPPORT_SEARCH_PATH make variable is set. If so, use that to look for folders with a devicesupport.xml file.
- The DeviceDb is also the DSL

Situation 4: A design file is provided, the executing tool is not using a project, the design file is anywhere (the "Timbuktu" case)

- In this case there are no MTB/MTBX files
- No DSL can be discovered
- Use the default DeviceDb

Situation 5: No design file is provided, the executing tool is using a project

- In this case there are MTB files
- Use the MTB files to find the DSL(s)
- Use the MTB files to find the DeviceDb
- If no DeviceDb is found yet, use dependencies from DSLs to find a DeviceDb.

- If no DeviceDb is found yet, use the default DeviceDb

Situation 6: No design file is provided, the executing tool is not using a project

- In this case there are no MTB/MTBX files
- No DSL can be discovered
- Use the default DeviceDb

In all these situations, if the rules do not find a DeviceDb or DSL then those values are left empty in the returned object.

8.1.27.1.1 Using MTB/MTBX files

Several of the situations require getting the DeviceDb or DSL from an MTB or MTBX file. There are two options for doing this. First, if a ManifestDb is already present in the ModusToolboxEnv then we will use that to find the “device_data” or “device_support” type for the item referred to by the MTB/MTBX file. If the ManifestDb is not present and the item referred to by the MTB/MTBX file is present on disk, then we will look at the props.json file in that asset and look for the “device_db” or “configurator_support” property in that file. If there is no ManifestDb and the asset is not already on disk then we cannot check to see if a given MTB/MTBX file is a DeviceDb or DSL.

8.1.27.1.2 The default DeviceDb

The “default” DeviceDb is defined as the highest release version available in the ManifestDb. If any versions of the DeviceDb come from an enabled Early Access Pack then the highest release version from the Early Access Pack is used. If there are no enabled Early Access Packs or no DeviceDb assets listed in that pack, then the highest release version from the public manifests is used.

8.1.27.2 Override

The search algorithm (described above) can be overridden by clients. If the “—library” command line argument is provided or the MTB_DEVICE_LIBRARY_PATHS environment variable is present, then the value provided by that flag or variable is used. That value is a list of one or more files, separated by commas. If the file is a props.json, it is checked for the appropriate property to identify it as a DeviceDb or DSL. If it is a devicesupport.xml file, we assume it is a DSL. Any directories that are not one of those are ignored.

The discoveries from searching these files are used to populate the DeviceFilesPaths object.

If an override is present, then none of the other search algorithms are used.

If both the flag and the environment variable are present, the flag is used.

If an override value is used, then the environment variable is written with that value (in process) so that all downstream processes will inherit the value.

8.1.28 Declaring documentation

Each of the following can be a potential location for documentation:

- Project scope: Assets such as BSP, middleware
- Environment scope: Tools, Packs

These items may or may not have a props.json file. If one exists, then we look for the “opt.documentation” property. This contains the following data:

```
{
  "opt": {
    "documentation": [
      {
        "filters": {
           "components": []
         }
        "type": "text/html",
        "title": "PDL API reference",
        "location":
"docs/pdl_api_reference_manual/index.html",
        "path": [ "Reference Guides", "PSoC" ]
      }
    ]
  }
}
```

In the case of assets (not tools or packs) that do not have a props.json file, the entire asset is searched for “index.html” or “index.htm” files. This is called a “legacy search”.

The files listed in the props.json and the files found by the legacy search are added to the documentation set.

Legacy projects (MTB 2.x) were allowed to put an “index.html” or “index.htm” file in the $project/libs folder hierarchy. Because of this, the $project/libs folder is also added to the search path. No other locations in the project are searched.

Field	**Description**
opt.documentation[n].type	MIME type of the document.
opt.documentation[n].title	Title of the document.
opt.documentation[n].location	Document file location relative to asset directory.
opt.documentation[n].path[]	Document category name, with one array element for each level of hierarchy starting from the top. In the example above, the top-level category would be “Reference Guides” and the sub-category would be “PSoC”.
opt.documentation[n].filters{}	OPTIONAL: Optional list of filter types that apply to this document.
opt.documentation[n].filters .components[]	OPTIONAL: Components list, with one array element for each valid component. The filter passes if any listed component is in the project.

The set of found documents is exposed through the QueryAPI ProjectInfo, IToolDb, and IModusToolboxPackDb classes.

Tools may specify their documentation in opt.documentation in their props.json. At the moment only lcs-manager-cli does so. There is an API in the QueryAPI, named Utils::getHelpFileByToolName, that can be used to parse the tool’s props.json and return the requested program’s documentation path. The requested program name may differ from the tool name this program belongs to. For example, capsense-configurator

and capsense-tuner both belong to the same capsense-configurator tool. Therefore for capsense-configurator tool it would be necessary to specify two opt.documentation blocks – one for each programs this tool comprises of. The value returned by the function is a value of the opt.documentation.location field. This value is an absolute path to the corresponding User Guide, e.g. lcs-manager.cli.pdf, capsense-configurator.pdf, capsense-tuner.pdf, etc. The function requires the opt.document.type to be of type “help” and the document name in opt.documentation.location to match the input argument for the program name.

8.1.29 Alternative Server support

Some users, notably those in China, are not able to use the existing MTB system as is due to firewall issues that prevent access to the public github server. For users who are unable to access the standard github location, Infineon has a mirror of the public github data that is more accessible. Using the alternative server requires indicating a new URL for downloading the ManifestDb data and a new setting for git for processing git operations.

These two values are managed through two settings from the ModusToolbox Settings. The user sets these settings using the standard settings dialog GUI that is present for other settings (e.g., the operating mode or proxy settings).

8.1.29.1 The system manifest URL

The MANIFESTDB_SYSTEM_URL indicates to the ModusToolbox environment where the super-manifest is located. Priori to having these settings in the ModusToolbox Settings, the super-manifest URL was hard-coded to a specify value and an environment variable, CyRemoteManifestOverride, could be set by the user to override the location.

8.1.29.2 The git “insteadof” setting

The GIT_INSTEADOF setting is used to set the “url.<newserver>.insteadof=<oldserver>” git property. See the git documentation for full documentation on how this setting is used by git. In summary, any instance of the <oldserver> substring in a URL is substituted with <newstring>.

The value of this setting is a tuple of a manifest url, the old server substring, and the new server substitution string. The URL in this tuple is

used to associate the old/new server strings with a particular MANIFESTDB_SYSTEM_URL setting. All these values are encoded in a single string and delimited by a “|” symbol.

There is no environment variable to override this setting. The default value for this setting is an empty string.

8.1.29.3 Preset and Custom Values

Both of these settings have a list of presets. The first is the previous hard-coded value for the URL and empty strings for the new and old server. The second is the hard-coded URL for the already existing alternate server and the new/old server string values that Infineon has been using manually up till now.

Default system URL:

https://modustoolbox.infineon.com/manifests/mtb-super-manifest/v2.X/mtb-super-manifest-fv2.xml

Default old|new git server:

<empty>|*<empty>*

Alternate system URL:

https://mtbgit.infineon.cn/raw/Infineon/mtb-super-manifest/main/mtb-super-manifest-fv2-mirror.xml

Alternate old|new git server:

https://github.com|*https://mtbgit.infineon.cn*

In addition to the preset values, users are free to specify custom values for either of these settings. The settings GUI gives the user the opportuntity to select on of the presets or enter values directly.

There is no validation done on the data that the user enters. This is considered an advanced usage and users are responsible for making sure the values are correct.

8.2 Executables

8.2.1 mtbsearch

8.2.1.1 Purpose

The mtbsearch tool is used to find files for compilation and to construct an include path. It does this by searching all relevant folders in a project.

8.2.1.2 Arguments:

-o filename	specify an output file name for the generated output. If no -o argument is provided, the output is sent to standard out.
--output filename	same as the -o argument
-a	generate absolute paths to the files and include directories
--absolute	same as the -a option
--verbose=INTEGER	set the level for output from the program. The larger the integer the more information that will be output.
--project projdir	the project directory to search for file
--generate	generate source files before searching
--create-dependencies	generate make rules for .xxx.o, .a, and .elf (MTB 3.2+)

8.2.1.3 Search algorithm

The algorithm essentially does a recursive search over relevant directories in the project. This search is altered by special folder filtering, a list of files and directories to ignore, and a list of files and directories to include (given that the parent of these items were in the ignore list). The data for the ignore list and addme list can come from different sources as described below.

8.2.1.3.1 Autodiscovery Caching

Autodiscovery is dependent on file system performance and is greatly impacted by anti-virus scanning. To maintain acceptable levels of performance we cache file system scanning results:

- Files and directories owned by the user (applications, projects, and BSPs) are **never** cached. We always scan the file system for these. This includes the make “SEARCH” variable.

- Asset search results are cached.
- If a valid autodiscovery cache exists it is used and we do not scan asset directories. The cache file is named .mtbsearch and it is read/written adjacent to the primary output .mk file, which is controlled by mtbsearch's -o argument (in practice, this is the project's build output root directory).

The cached file is determined to be valid if and only if ALL of the following are true:

- No assets have been added or removed.
- No asset versions have changed (based on selected tag).
- CONFIG, TARGET, COMPONENTS, DISABLED_COMPONENTS, TOOLCHAIN have not changed.
- User-owned (project and BSP) .cyignore files are older than the cache.
- The version of the cache is supported by mtbsearch.

If the cache is invalid, then the traditional file system scan is performed, and those results are written to the cache (replacing the old / out of date cache).

The cache file has a version field that captures the version of the file format. The only valid version as of now is 1.

Errors follow standard mtbsearch semantics – they are written to stderr.

Sample JSON

```
[
    {
        "assets": [
            {
                "asset": "core-make",
                "version": "release-v3.2.2"
            },
            {
                "asset": "mtb-pdl-cat1",
                "version": "release-v3.9.0"
            }
        ],
        "components": [
            "CAT1",
            "CAT1A",
            "SOFTFP"
        ],
        "config": "Debug",
        "created": "Wed Jan 17 11:43:25 2024",
        "cyignorelist": [
            "/path/to/project/.cyignore"
```

```
        ],
        "files": [
            "/path/to/mtb_shared/asset/version/foo.c",
            "/path/to/mtb_shared/asset/version/foo.h"
        ],
        "includes": [
            "../mtb_shared/mtb-pdl-cat1/release-v3.9.0/devices/COMPONENT_CAT1A",
            "../mtb_shared/mtb-pdl-cat1/release-v3.9.0/devices"
        ],
        "target": "APP_CY8CKIT-062-BLE",
        "toolchain": "GCC_ARM",
        "version": 1
    }
]
```

8.2.1.3.2 Special folder filtering

A project may contain folders will special names. For example, TARGET_somebsp. When the search algorithm encounters any folders with a special name, the appropriate comparisons are made to decide if the folder should be included in the search or not.

Folders in the form “COMPONENT_<component>” are only included in the search if “<component>” is defined in the COMPONENT make variable.

Folders in the form “TARGET_<bspname>” are only included in the search if the TARGET make variable value matches “<bspname>”.

Folders in the form “TOOLCHAIN_<toolchain>” are only included in the search if the TOOLCHAIN make variable matches “<toolchain>”.

Folders in the form "CONFIG_<config>" are only included in the search if CONFIG make variable value matches "<config>".However due a bug in core-make-v3.0.0, the CONFIG make variable may not be passed via get_app_info, in this case, no filter should be done.

8.2.1.3.3 “.cyignore” files

A project can contain files with the name “.cyignore” to specify directories and files that should be ignored by the search algorithm. These files can only be at the project root or the root of any asset in the project. Each line in the “.cyignore” file specifies a file or directory that should be ignored. If the line starts with “#” then it’s a comment and that line is skipped. The path on the line can contain “.” or “..” or spaces. Leading and trailing whitespace will be ignored. There is no support for wildcards in the “.cyignore” file. The entries from all these files are combined and added to the “ignore list”.

8.2.1.3.4 CY_IGNORE

The make variable CY_IGNORE can be set to a list of files or directories to ignore. These values are added to the “ignore list”.

8.2.1.3.5 props.json

Each asset in the project contains a “props.json” file. If the optional JSON property “/opt/ignore” is set then this is treated like a list of entries similar so the CY_IGNORE variable. These entries are added to the “ignore list”. If the optional “/opt/addme” property is set then these values are added to the “addme list”. In order for an item to be added to the “addme list” then some parent of the item must be in the “ignore list”. If the optional “/opt/stop-at” property is set then this value is added to the “stopat list”. The “stopat list” is used while generating the include path.

8.2.1.3.6 Comparing filenames and paths

Whenever the algorithm needs to compare file or directory names (for example, to see if something is on the ignore list), the names are converted to ***QFileInfo*** objects. The ***QFileInfo*** class comparison handles “.” and “..” resolution, whitespace in paths, and operating specific case sensitivity.

8.2.1.3.7 Directory ordering

The order of the entries on the include path is important. The algorithm ensures that directories in the include path are as follows:

- Project root
- Directories listed in MTB_SEARCH (from the user’s makefile)
- Local assets
- Shared assets

8.2.1.3.8 Recursive search

After collecting the starting directories, the ignore list, the addme list, and the stopat list, the algorithm does a DFS starting with the starting directories. When the DFS encounters a file that is on the ignore list, that file is omitted from the output. When the DFS encounters a directory that is on the ignore list, that directory is omitted and the DFS does not descend into that directory. All files in the addme list are added to the output. Each directory in the addme list is added to the output and is considered a new root for the DFS.

While processing a directory, if that directory contains any *.h, *.hpp, or *.hxx files then that directory is considered for adding to the include path.

When a directory is to be added to the include path, that directory's full path is compared with items in the stopat list. If there is a string in the stopat list that is a prefix of the directory's path, then the stopat value is added to the include path rather than the full directory path.

8.2.1.3.9 Output

The output file is a makefile excerpt that assigns the following variables:

CY_SEARCH_ALL_FILES contains a list of all of the files discovered during the search process.

CY_SEARCH_ALL_INCLUDES contains a list of all of the directories that contained at least one .h or .hpp file.

8.2.1.3.9.1 Dependency rules

The dependency rules will only be generated when the --create-dependencies command line argument is present. Supported in MTB 3.3+.

For each source file (.c, .cc, .cpp, .cxx, .s, .S), mtbsearch will generate an empty rule:

```
$(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/shared/asset1/src/a1x.c.o: \
    $(SEARCH_asset1)/src/a1x.c
$(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/shared/asset1/src/a1y.c.o: \
    $(SEARCH_asset1)/src/a1y.c
$(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/shared/asset2/src/a2.S.o:  \
    $(SEARCH_asset2)/src/a2.S
$(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/project/appmain.c.o: appmain.c
```

Within the `$(MTB_TOOLS__OUTPUT_BASE_DIR)/$(MTB_CONFIG)` directory, object files produced from asset source files will be placed in the `shared` or `local` subdirectory for assets in mtb_shared and the libs directory respectively. Object files produced from project source files will be placed in the `project` subdirectory. Object files produced from source files in other directories (added via `MTB_SEARCH` variable) will be stored in the `external` subdirectory with a hierarchical path based on the file's location. The path for external files will be modified to replace special names and characters.

- Replace `.` with _ and `..` with __
- On Windows, also replace colons (`C:` becomes `C_`)

If the --archive-assets option is enabled, mtbsearch will generate an empty rule with the list of the asset's objects:

```
$(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/asset1.a:                    \
    $(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/shared/asset1/src/a1x.c.o \
    $(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/shared/asset1/src/a1y.c.o
$(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/asset2.a:                    \
    $(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/shared/asset2/src/a2.S.o
```

mtbsearch will generate a list of the project's link dependencies. When using --archive-assets the archive paths will be added to MTB_SEARCH_ALL_OBJECTS. Otherwise, the asset object files will be added instead.

```
MTB_SEARCH_ALL_OBJECTS =                                  \
    $(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/asset1.a          \
    $(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/asset2.a          \
    $(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/project/appmain.c.o
```

mtbsearch will generate lists of source files and the corresponding object files (for compile_commands.json):

```
MTB_SEARCH_SOURCES =           \
    $(SEARCH_asset1)/src/a1x.c \
    $(SEARCH_asset1)/src/a1y.c \
    $(SEARCH_asset2)/src/a2.S  \
    appmain.c

MTB_SEARCH_OBJECTS =                                                \
    $(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/shared/asset1/src/a1x.c.o \
    $(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/shared/asset1/src/a1y.c.o \
    $(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/shared/asset2/src/a2.S.o  \
    $(MTB_TOOLS__OUTPUT_BASE_DIR)/Debug/project/appmain.c.o
```

8.2.2 mtbgetlibs

8.2.2.1 Purpose

The console application mtbgetlibs is used to implement the "make getlibs" operation. This operation examines a given ModusToolbox project for dependent middleware assets that are required. These middleware assets are defined by "mtb" files that are found in the project. The "mtb" files are generally stored in the "deps" directory but can be anywhere in the project. Once the dependent middleware assets are identified, the dependency manifests are checked to see if the dependent assets have any dependencies. These dependencies are known as indirect dependencies. The mtbgetlibs executable ensures that all direct and indirect dependencies of the project are downloaded and available.

The mtbgetlibs performs these functions while honoring the desired destination of the various assets.

The ***ProjectChangeMgr*** in the MTBQueryAPIs implements all of these functions. The mtbgetlibs executable is just a wrapper for using the ***ProjectChangeMgr*** to resolve all dependencies and then commit the results.

8.2.2.2 Arguments:

--project dirname — specifies the directory name for a directory that contains a ModusToolbox project (and not application).

--app dirname — specifies the directory name for a directory that contains a ModusToolbox application

--verbose=INTEGER — set the level for output from the program. The larger the integer the more information that will be output.

--print-git — print information about each GIT operation as it is started and completed.

--offline — use the offline data to complete this request

--direct — use the direct mode to complete this request

8.2.3 mtbquery

8.2.3.1 Purpose

The mtbquery executable provides information about the ModusToolbox environment, including a specific application or project if desired. The mtbquery tool has multiple “modes” of operation. Each of these is described below and only one of these modes can be active per invocation of the mtbquery executable.

8.2.3.2 Arguments:

Note, only one of --toolopen, --toolinfo, and –envinfo can be invoked for a given execute of mtbquery. Multiple of these will be an error.

--appinfo — generate data at the application level. This is a reduced set.

--toolopen	this argument specifies that the information needed by the “make open” target should be output as a set of name/value pairs. There should be no project or application associated with this option and to provide it will be an error.
--toolinfo toolid	this argument provides information about a specific tool.
--tool	specifies the tool name
--normalize	this argument invokes the environment information output for mtbquery. This output takes a set of project data input or application data input and normalizes the values to match the SAS. The normalized values are then output as a set of NAME=VALUE pairs.
--envinfo	indicate that the data should be normalized to MTB 3.0 values.
--eval	??
--json	specifying this flag cases the output for the --toolinfo, --envinfo, and the --all modes to be output in JSON. This option is not supported for all output types.
--printlibs <project>	prints the git commit info about the assets in the project and device-db.
--project dirname	this operation provides the directory name requires for the various modes listed above.
--output filename	the name of the file for the output
--alltools	this argument causes mtbquery to process the opt.tools section of the props.json file for each valid tool and output the relevant information.
--verbose=INTEGER	set the level for output from the program. The larger the integer the more information that will be output.

8.2.3.3 Details

8.2.3.4 –toolinfo toolid

This --toolinfo option will out the following if the --json command line argument is provided.

```
{
  “id” : “UUID”,
  “name” : “TOOLNAME”,
  “path” : “AbsolutionPathToTool”
}
```

This --toolinfo option will out the following if the --json command line argument is NOT provided.

```
ID=”TOOLID”
NAME=”TOOLNAME”
PATH=”AbsolutePathToTool”
```

8.2.3.5 –envinfo

This option is intended to normalize an application environment to the ModusToolbox 3.0 environment as it is defined in the SAS.

This option will just take the values used to describe a project or application environment and echo them back to the output as a set of NAME/VALUE pairs. The only exception is that an additional value of MTB_QUERY=1 will be output if the MTB_QUERY value is not defined in the input. Finally, there is a set name/value pairs on the input that will be translated to ModusToolbox 3.0 values on the output. These are defined in the table below.

Input	**Output**
TARGET_DEVICE	MTB_DEVICE
SEARCH	MTB_SEARCH
TOOLCHAIN	MTB_TOOLCHAIN
TARGET	MTB_TARGET
CONFIG	MTB_CONFIG (This entry should be required, but due to a bug in core-make-v3.0.0. MtbQueryAPI need to handle case where this is not set)
COMPONENTS	MTB_COMPONENTS
DISABLE_COMPONENTS	MTB_DISABLED_COMPONENTS
ADDITIONAL_DEVICES	MTB_ADDITIONAL_DEVICES
CY_GETLIBS_PATH	MTB_LIBS
CY_GETLIBS_DEPS_PATH	MTB_DEPS
CY_GETLIBS_SHARED_NAME	MTB_WKS_SHARED_NAME
CY_GETLIBS_SHARED_PATH	MTB_WKS_SHARED_DIR
FLOW_VERSION	MTB_FLOW_VERSION
CY_TOOLS_PATH	MTB_TOOLS_DIR
SUPPORTED_TOOL_TYPES	MTB_BSP_TOOLS_TYPES

Input	Output
APP_NAME	MTB_APP_NAME
CY_IGNORE	MTB_IGNORE

In addition, there are a set of names that will receive default values if they are not present in the input.

Name	Default Value
MTB_IGNORE_EARLY_ACCESS_PACKS	FALSE
MTB_TYPE	LEGACY
MTB_TOOLS_MAKE	FALSE
MTB_FLOW_VERSION	1
MTB_MW_TOOLS_TYPES	“”

8.2.3.6 --toolopen

This mode of operation is enabled via the –toolopen DIR command line flag where DIR is the directory path to a ***project*** within a ModusToolbox ***application***. This mode prints a set of information to standard out that is intended to be consumed by make.

The options –absolute and – relative controls the type of path that is output. If neither are specified, then the App Structure will determine the type of paths. MTB2x applications will use relative paths and MTB3X applications will use absolute paths. If the paths are relative, tools from ModusToolbox packs will be omitted.

Legal “open” values (CY_OPEN_ prefix) come from the “opt.programs[].compat.open” section of tools props.json files. They generate make “open” variables that are compatible with the ModusToolbox 2.X make open support. For ModusToolbox 2.4, these variables are “hard coded” in the file tools_2.4/make/tools.mk. In ModusToolbox 3.X, these variables are generated by this mtbquery executable.

Programs are described to make via the opt.programs[].make-vars” section of the props.json file. They generate make variables prefixed with CY_TOOL (required for core-make 2.x support). For ModusToolbox 2.4, these variables are “hard coded”. In ModusToolbox 3.x, these variables are generated by this mtbquery executable.

Each entry in this section is prepended by the text “CY_TOOL_” and the value is processed for any pseudo variables. These pseudo variables are described below.

The MTBQueryAPIs processes tools found across all legal locations and for a given tool, only picks up the tool with the latest version. The CY_TOOL_ output will describe this tool and therefore this mechanism implements the patch mechanism currently used in ModusToolbox 2.x.

8.2.3.6.1 Pseudo Variables

The following tables list all the pseudo-variables, explains what the variable is for, and lists the arguments (along with explanations for the arguments).

Variable: ADDITIONAL_DEVICES
Purpose: Returns any additional devices that are advertised as part of the BSP. This generally includes connectivity devices.
Arguments: The argument specifies the delimiter that will be used to separate multiple items if there are multiple items in the resulting list. The special argument value “%space” indicates that a space character should be used as the delimiter.

Variable: APP_DIR, APPDIR
Purpose: Returns the absolute path to the application directory.
Arguments: none

Variable: APPINFO
Purpose: Write the current get_app_info data to a file and return that filename.
Arguments: none

Variable: BSP_DIR, BSPDIR
Purpose:

The absolute path to the BSP directory for the project.
Arguments: none

Variable: CFGARGS

Purpose:
 	
This variable is used to generate a string of command line arguments that could be passed to a configurator. The arguments to the configurator will indicate the design file and the config file that that configurator needs.
 	
The algorithm for generating the arguments is as follows:
 	
`file = search(regexp,scope);`
`if (file) {`
`  path = file.path()`
`} else {`
`  Path = project.path()`
`}`
 	
`designFile = path + designFile`
 	
`if (designFile.exists()) {`
`  output = design_flag + design_file + config_flag + config_file`
`} else {`
`  Output = config_flag + config_file`
`}`
 	
In other words, we search for a file matching the given regular expression (arg1). The list of directories to search is provided by the scope argument (arg6). If we find that file then we capture the path to that file. If we don't find the file then we capture the project path. Then, using the provided design file name (arg3), we construct the design file name. If that file exists, then we generate the output string using the provided design flag (arg2), design filename (arg3), config flag (arg4), and config filename (arg5). If the design file doesn't exist then we generate the output string using the provided config flag (arg4) and the config filename (arg5).

Arguments: 1. Regular expression for matching a filename to look for. This is used only to find a path. 2. The name of the design file (e.g. "design.modus") 3. The name of the config file

4. The flag used to specify the design file on the command line
5. The flag used to specify the config file on the command line
6. The scope of directories to search in
    a. all" – search the entire search path
    b. "bsp" – search only the BSP directory
    c. "project" – search only the project directory, skipping the BSP directory if it is located in the project
    d. "bsp_project" – search the project directory and the BSP directory

Variable: DEVICE
Purpose: Returns the device for the project. This is the same as the DEVICE variable from the BSP makefile. Note that this assumes that all projects in an application use the same BSP.
Arguments: none

Variable: DEVICE_SUPPORT_LIB
Purpose: The result is the information required to launch the device configurator. If this project is supported by an MTB2X version of the device support library, the result is the full path to the devicesupport.xml file. If this project is supported by an MTB3X version of the device support library, the results is two paths separated by a comma. The first path, is the full path to the props.json file in the device support library. The second path is the full path to the props.json file in the device database library.
Arguments: none

Variable: DEVICE_SUPPORT_LIB_OPT
Purpose: This has the same functionality as DEVICE_SUPPORT_LIB. The results from DEVICE_SUPPORT_LIB are first calculated. If those results are non-empty, then they are prefixed with "--library". If the results of DEVICE_SUPPORT_LIB are an empty

string, then the result from DEVICE_SUPPORT_LIB_OPT will also be an empty string (that is, we don't put the --library prefix on an empty string).
Arguments: none

Variable: FINDDIR
Purpose: Perform a recursive search for the directory that contains the file that matches the regular expression given. The starting point of the search is specified by the second argument to the variable. The third argument controls whether the resulting path is absolute or relative.
Arguments: 1. A regular expression that represents the file to look for 2. Indicate which directories to search. a. "all" – search the entire search path b. "bsp" – search only the BSP directory c. "project" – search only the project directory, skipping the BSP directory if it is located in the project d. "bsp_project" – search the project directory and the BSP directory 3. Specify the format for the returned path. The options are: a. "relative" – return the path as relative to the project b. "full" or "absolute" – return the path as an absolute path 4. Indicate how to respond if 0 entries are found. The options are: a. "project" – indicate success and return the path to the project directory b. "error" – indicate an error and return an empty string c. "ignore" – ignore the "ignore condition" and return an empty string

Variable: FINDFILE
Purpose:

Perform a recursive search and search for a single file that matches the regular expression given. The starting point of the search is specified by the second argument to the variable. The third argument controls whether the resulting path is absolute or relative. This is typically used to find files like “design.modus” when invoking configurators.

If multiple files are matched, it is an error. If no files match, this returns an empty string.

Arguments:

1. A regular expression that represents the file to look for
2. Indicate which directories to search.
    a. “all” – search the entire search path
    b. “bsp” – search only the BSP directory
    c. “project” – search only the project directory, skipping the BSP directory if it is located in the project
    d. “bsp_project” – search the project directory and the BSP directory
3. Specify the format for the returned path. The options are:
    a. “relative” – return the path as relative to the project
    b. “full” or “absolute” – return the path as an absolute path

Variable: MPN_LIST_OPT

Purpose:

If the project has a non-empty value for the MPN_LIST property, output “—check-mpn-list $value” otherwise output an empty string.

Arguments: none

Variable: PROJECT_DIR, PROJECTDIR

Purpose:

Returns the absolute path to the project directory.

Arguments: none

Variable: TOOL_DIR, TOOLDIR
Purpose: Return the path to the tool being processed. The argument controls if the returned path is relative or absolute. If the path is relative then it is relative to the base tools directory.
Arguments: 1. Specify the format for the returned path. The options are: a. “relative” – return the path to the tool relative to the base tools directory b. “absolute” – return the path as an absolute path c. “mtb3x_mtb2x” – if the project is a MTB2X project then treat this as relative, otherwise treat it as absolute.

Variable: TOOLPATH
Purpose: This variable is used to find the path to a tool executable. This variable is evaluated in the context of a tool. It uses that tool’s path and looks for the given file at that path. If the given file doesn’t exist there then it also tries that file with “.exe” appended. If the file is found then the path to that file is returned. The second argument specifies what style of path to return. Note that if the returned path is relative then that is relative to the *base* tools directory.
Arguments: 2. The name of a tool executable file (e.g., “library-manager-cli”). 3. Specify the format for the returned path. The options are: a. “relative” – return the path to the tool relative to the base tools directory b. “absolute” – return the path as an absolute path c. “mtb3x_mtb2x” – if the project is a MTB2X project then treat this as relative, otherwise treat it as absolute.

8.2.3.7 --alltools

This option is a subsystem of the --toolopen option. When mtbquery is called with the --alltools, the same flow is followed as when the --toolopen option is provided but only the opt.tool section of the props.json file is processed. With the --alltools option, an application is not loaded into the

MTBQueryAPI, so any of the Pseudo variables that are referenced that require application information will result in an error.

The intent of this option is just to output a set of make variables that describe what tools are available and where they are located on current machine.

8.2.4 mtblaunch

8.2.4.1 Purpose

The mtblaunch application has two purposes: output a JSON object that describes that Programs (as formal objects that are found in Tools) that are valid in the given application, launch a program.

8.2.4.2 Arguments

Argument	Purpose
--quick	This option indicates that the Program quick launch information is desired.
--app dir	This option is required and gives the application directory that contains the application to load.
--verbose	This option is optional and if specified causes mtblaunch to output helpful debug information as comments in the output JSON file.
--docs	Generate information about docs from the projs.json files.
--short-name name	The short-name ID for a tool to be launched.
--project dir	The directory to launch “short-name” on.

8.2.4.3 Generating documentation links

If the “--docs" argument is supplied, the mtblaunch will use the QueryAPI APIs (see ProjectInfo, IToolDb, and IModusToolboxPackDb classes) to collect the found documents and then generate a list of document links that an IDE can display for users to have easy access to important documents.

8.2.4.4 Output Format

The output are 2 JSON objects with a single top field named “configs” and "projects".

The value of the "configs" field is an array of JSON objects that describe each Program that can be launched.

Field	Purpose
base-dir	The base directory for the Tool that contains this Program.
tool-id	The ID for the tool that contains this Program.
display-name	The display name for this Program.
icon	The icon to display for this Program.
id	The ID for the Program.
short-name	The short name for the Program (e.g. device-configurator, capsense-configurator, etc.)
version	The version number of the Program.
type	Type of the Program. The valid values are “none”, “bsp”, “project”, “all”.
scope	This indicates the scope of the launch for this Program. The valid values are “global”, “bsp”, or “project”.
cmdline	This field gives the command line that should be executed to launch this Program in the context given.
operation	This field indicates the type of operation being performed with this launch. Valid values are “run”, “open”, and “new”.

file	If the operation is open, this field gives the name of the file that will be opened. This field is only present if the operation is “open”.
project	If the scope of the entry is “project”, this field gives the specific project this entry is targeting.

In general, all of this is intended to provide a rich set of information for the caller to display these Program launch options to the user in a meaningful way. Since display is left to the calling program, generally an IDE, there is no attempt here to provide hints as to how this should be displayed.

To invoke any of these programs, the “cmdline” field should be invoked exactly as is.

The value of the "projects" is array that describes information that caller may want to be displayed. Currently this field only contains BSP information for each project

Field	Purpose
name	The base directory of the project
bsp	The project's selected BSP

8.2.5 lcs-manager-cli

The command line tool (tentative called "lcs-manager-cli") allows users to see a list of the content that is stored locally, remove local content, and add local content.

lcs-manager-cli --list

List all the content in the LCS. Each asset/version combination is on a line and it prints the reponame and version.

lcs-manager-cli --add-all

Add all the content from remote into the LCS. This updates all existing items and adds new items.

lcs-manager-cli --clear-all

Remove all content from the LCS.

lcs-manager-cli --list-bsps

Show the list of BSPs on the watchlist

lcs-manager-cli --add-bsp <bsp>

Download the BSP and all related assets. This includes all dependencies, all middleware that is compatible with the BSP, all code examples that are compatible with the BSP, and all middleware referenced by those code examples. This also adds the specified BSP to the watchlist.

lcs-manager-cli --clear-bsp <bsp>

Remove the content related to the BSP. Note that if content is related to another BSP then it needs to remain. This BSP is removed from the watchlist.

lcs-manager-cli --check-for-updates

Check for new versions of anything that already exists in the LCSand for new assets. Generate a report for all things that can be updated and all assets that can be added to LCS.

lcs-manager-cli --update [<reponame>]

Update the data in the LCS for the specified reponame, or all items related to the specified bsp.

lcs-manager-cli --update-existing

Release each asset in the LCS with the newer one in the source manifest. This means that new versions of existing assets will be added to the LCS but also, if old assets were modified in the public database, those modifications will be acquired as well.

If the user has setup a watchlist, then all new assets that are compatible with items on the watchlist will be added to the LCS. Note that if there is not a watchlist, assets that exist in the public manifest db but do not already exist in the LCS will not be downloaded.

8.2.6 mtbninja

The mbtnijna executable is a tool that generates a NINJA build file (as opposed to the existing build system that generates the data for a

makefile). Ninja build files are processed faster so using mtbninja improves the overall build performance significantly. Mtbninja is called from the make build system. It takes information provided by make in the form of a set of dotfiles and the MTBQueryAPI to generate an NINJA build file. The following dotfiles are used:

- .arflags - flags to use when calling ar for creating libraries from object files (per asset)
- .asflags - flags to use when building assembly files
- .cflags - flags to use when building "C" files
- .cppflags - flags to use when building "CPP" files
- .defines - the defines to use when building C and CPP files.
- .includes - includes to add to the generated include list (INCLUDES list in makefile)
- .ldflags - flags to use when linking
- .ldlibs - list of libraries to add to the link phase
- .sources - any additional sources to build (SOURCES list in makefile)

These are the arguments to mtbninja:

- --force-unique - forces all generated object files paths to be unique, may result in long path names, off by default.
- --src-paths-relative - forces all source files in build lines to be relative paths, off by default
- --src-obj-names - forces output object files names to include source extension (x.c → x.c.o), off by default
- --disable-windows-path-checks - disables windows path length checks and colliding object file paths, on by default
- --verbose=# the level of debug logging displayed
- --archive-assets - builds by creating a library archive (.a file) per assets and links against these .a files when linking, off by default

- --generate - run code generation as part of the build process
- --project PATH - the directory containing the project of interest
- --ninja FILENAME - the name of the ninja file to write
- --aspath FILENAME - name of the assembler executable
- --ccpath FILENAME - name of the CC executable
- --cpppath FILENAME - name of the CPP executable
- --ldpath FILENAME - name of the linker executable
- --objcopypath FILENAME - name of the objcopy executable
- --arpath FILENAME - name of the ar executable
- --gccpath PATH - the path to a GCC compile, shortcut for --aspath, --ccpath, --cpppath, --ldpath, --objcopypath, --arpath
- --clangpath PATH - the path to a CLANG compiler, shortcut for --aspath, --ccpath, --cpppath, --ldpath, --objcopypath, --arpath
- --elffile FILENAME - name of the output elf file (if not in the .ldflags file)
- --ldscript FILENAME - name of the linker script (if not in the .ldflags file)
- --mapfile FILENAME - name of the mapfile (if not in the .ldflags file)
- --defines FILENAME - name of the file that contains the list of defines
- --ldlibs FILENAME - name of the file that contains extra libraries to add to linking
- --includes FILESNAME - name of the file that contains includes to add to the include list
- --sources FILENAME - name of the file to add additonal sources to the compile process
- --asflags FILENAME - name of the file that contains assembler flags

- --cflags FILENAME - name of the file that contains c compiler flags
- --cppflags FILENAME - name of the file that contains cpp compiler flags
- --ldflags FILENAME - name of the file that contains linker flags
- --arflags FILENAME - name of the file that contains archiver flags
- --build-dir DIR – the output directory

8.2.6.1 Adjusting Input Values

One of the inputs to mtbninja is a list of defines that need to be output in the generated ninja file. The defines come from the Makefiles and the values in the defines have been decorated with the necessary escape characters so that they can be passed through bash. This is because this is the default behavior for make when it calls gcc. This doesn't work for ninja because ninja doesn't call a shell to launch gcc but instead uses fork/exec. When the define values are processed with ninja, none of the bash escape characters are processed and this leads to incorrect values being presented to gcc.

As the values are read in by mtbninja, they are unscaped to remove the bash-style escaping. After doing this, the data is in "raw" form. Much of the data is used as arguments to other commands. When constructing commands from this data, it is necessary to apply the correct escaping. The ninja documentation indicates that on Windows, it launches each command using CreateProcess() from the Windows API with the first argument null. For all other operating systems, ninja launches each command using bash. Because of this, as each command line is generated from the raw data, it has to be escaped correctly for either CreateProcess() or bash.

Finally, the fully escaped command is eventually going to end up in a ninja file. The ninja file format has its own set of escape rules. As the escaped command data is written to the ninja file, ninja escapes are applied.

8.2.6.2 Unescaping bash

For "unescaping" the values, we do not cover all bash expansions and substitutions. We only consder the following cases:

1) bare word (no quotes wrapping the value)

Example: VAR=value

In this case we find all instances of *x* and replace them with *x*.

2) single quotes

Example: VAR='value'

In this case we strip the single quotes off the front and back of the value and leave the rest of the string as is.

3) double quotes

Example: VAR="value"

In this case we strip the double quotes off the front and back of the value. Then, for each found \, we check the subsequent character. If the subsequent character is one of $, `, \, ", or !, then the backslash is removed and the subsequent character is used. If that character is not one of the "special" characters, then the backslash is left in the string (as well as the subsequent character).

8.2.6.3 CreateProcess escaping

The escaping rules for CreateProcess are not well document. We generated the rules by reverse engineering how CommandLineToArgv in the Windows API works. The goal is to quote and escape the orig string so that CommandLineToArgv will see it as the original. For example, to preserve a " character, it has to be escaped with a \. There are arcane details about odd vs even numbers of \ characters. The actual rules are document in comments in the code.

8.2.6.4 Bash Escaping

Bash escaping follows the well document rules. Primary, all white space, and key punctuation is proceeded by a backslash (\) and then the entire string is enclosed in double quotes (“).

8.2.6.5 Ninja Escaping

Ninja escaping follows the documented rules for ninja but we only escape the subset that we will encounter. In particular, the dollar sign ($), the colon (:) and the space character ( ) are replaced with the same character but proceeded by a dollar sign ($).

8.2.7 mtbarchive

The mtbarchive is a tool that allows a user to create an archive of one or more MTB applications. The archive is a zip file that contains only the essential files for an application. The intention is that this zip file can be transferred from one user to another and the recipient can unpack the archive and import the application into their IDE of choice. It is important to note that the archive is intended to be IDE-agnostic to allow for complete flexibility between users. That is, the sender can use any IDE and the recipient can use any IDE.

In addition to the files and directories, the archive contains a readme.txt file that provides instructions to the recipient on how to import the application into an IDE, the list of applications/projects in the archive, and the list of Technology Packs that are installed and in use on the sender's machine.

The mtbarchive tool is provided as both a commandline tool (mtbarchive.exe) and a GUI tool (mtbarchive-gui.exe). The details of each of these interfaces are provided in sections below.

8.2.7.1 Basic Functionality Details

The basic functionality of the mtbarchive tool is described here. It is provided with a list of directories and the name of the target archive file. It then goes through the following steps:

1. Create a temporary folder.
2. Copy all selected applications to be archived into the temporary folder.
3. Process recursively all copied projects, performing the following steps:
    a. Check if there is an application project in the folder. If not, output an error message, delete the temporary folder, and terminate work; if there is a project in the folder, continue with further actions;
    b. If there is a ".mtbLaunchConfigs" folder, delete it;
    c. If there is a "build" folder, delete it;
    d. If there is a "libs" folder, delete it;
    e. If code generation for the project is not turned off, recursively search the project directory (and BSP directory if it is outside

the project) for all folders named "GeneratedSource" and delete those folders. Note that in the case of 3.x projects, these will be in the "bsps" folder in the project but for 2.x projects, these will either be at the root or a sub-directory of a BSP in the "libs" folder.

f. If there is a ".cproject" file, delete it;

g. If there is a ".project" file, delete it;

h. If there is a ".mtbqueryapi" file, delete it;

4. Call the QueryApi function to generate a readme.txt file in the root of the temporary directory.

5. Perform archiving of all contents of the temporary folder into the specified archive file.

6. Delete the temporary folder.

The mtbarchive tool uses the ProjectInfo API from the QueryAPI to determine all the relevent directories and if code generation is on/off for a project.

8.2.7.2 QuickPanel integration

The mtbarchive tool provides a props.json file that allows the GUI tool to be advertised via mtblaunch for tools that have a QuickPanel (e.g., Eclipse and VSCode). The GUI tool sets its type to "all" and provides a "open-file" command with arguments to specify a default output file and uses the $$APP_DIR$$ psuedo variable to determine the default application directory to archive. The CLI tool sets its type to "none" and does not provide a command.

It is expected that IDEs can use the GUI tool for simple operation but if the IDE wants to provide a richer user interface then it can provide that in its native environment then call the CLI tool behind the scenes. For example, in the future, we may update the Eclipse IDE to integrate with the existing "export" dialog which would allow archiving multiple applications at one time.

8.2.7.3 CLI tool

The commandline tool provide a command line interface for users to create archives of an application. The commandline syntax is simple. It supports the following options:

-h, --help Show help information

-v, --verbose Detailed output.

-o, --output <filename> Output archive name

-f, --force Overwrite existing output file if it already exists

The "output" option is required. It is also required that the user provide at least one application directory. A simple check for the existance of a Makefile is run on each application directory to ensure that it is (probably) an application.

After the command completes without errors, the output file contains an archive of the requested directories.

8.2.7.4 GUI tool

The GUI tool is a very simple interface that has a field that allows the user to specify the output file and a field that allows the user to specify a single application folder to archive. The GUI tool accepts commandline options to optionally specify the output file and application folder. If these are provided on the commandline then those GUI input fields are preset to those values but the user is free to change them.

When the user presses on the "create archive" button in the GUI, the archive is created the the success or failure status in reported to the user.

8.3 Unit Testing

Unit tests for the query APIs are located in the MTBQueryApiTest subdirectory of the mtb-env project. The test suite is implemented using the Google Test framework, and exercises the functionality of the query APIs on a controlled set of test projects, manifests, and libraries.

8.3.1 Mock Data

The TestData directory contains applications, BSPs, and tech packs used to test the query APIs, including both MTB2.X and 3.X applications. Some applications need to have 'make getlibs' run before the test suite is executed. The bootstrap.sh script handles this by copying those applications to the mtb2x_prepared and mtb3x_prepared subdirectories of TestData and running getlibs.

Rather than using production manifests, boards, and middleware (which are subject to change and take a relatively long time to download

repeatedly), the unit tests run on mock data located at https://gitlab.intra.infineon.com/repo/mtbide/mtb_env_testdata/unit_tests. The mock data implements the minimal subset of manifests, BSPs, and middleware required to run the query APIs without errors; actual code, documentation, and any other files that are unnecessary for the unit tests are not include. The mock data includes the following assets:

8.3.1.1 /manifests

A super manifest pointing to BSP, middleware, and code example manifests; and dependency manifests for code and middleware.

8.3.1.2 /bsps

Three mock BSPs; two supporting both generation 2 and generation 3, and one supporting generation 4.

8.3.1.3 /middleware

Mock middleware. `core-make` and `recipe-make-cat1a` are required for testing as without them the make build infrastructure will fail. `has-devicesupport` is an empty library which declares configurator support, and `has-dependencies` is an empty depends on two other empty libraries (`dependency-a` and `dependency-b`).

8.3.2 List of Tests

Test Suite	**Behavior Tested**
ApplicationCreationTest	Creating applications from code examples
AssetRequestTest	Creating asset requests from manifest items and .mtb files
BSPDataModelTest	Loading metadata from a BSP on disk
CommitVersion	Parsing and comparing version strings (such as ‘release-v1.1’ or ‘latest-v2.X’)
ConfigDirSupport	Detecting configurator support libraries
DocInfoTest	Constructing, validating, and exporting JSON containing documentation info
DownloadAssetTest	Fetching assets from remote Git repositories
DownloadManagerTest	Downloading files from URLs

EvalTest	Resolving macros in strings
getapitest	Creating a ModusToolox environment
IsPathInAppTest	Determining whether a file is located within an application directory
LoadApplicationTest	Loading metadata for MTB2.X and MTB3.X applications
LoadManifestTest	Downloading and parsing a whole manifest
LoadPacksTest	Loading metadata for tech packs
LoadToolsTest	Loading the tools database and correctly identifying tools for each supported platform
ManifestDbLocMgrTest	Determining which manifest to use
ManifestDbTest	Semantics of the manifest database, including conflict resolution and error handling
ManifestGroupTest	Semantics of app and library groups in the manifest database
ManifestItemTest	Semantics of manifest items & capabilities
ManifestParserTest	Parsing manifest database items from XML
OperatingModeTest	Locating local content
ProjectChangeMgrTest	Managing a project's direct and indirect dependencies, editing Makefiles, and importing BSPs
RunnerTest	Invoking external utilities
SuperManParserTest	Parsing super manifest XML files
VersionTest	Parsing and manipulating semantic versions

9. QUALITY REQUIREMENTS

All assets described by this document will adhere to Cypress specifications for design, development, review and test of software assets. The IROS for the individual assets should describe each of these per the IROS specification.

10. COMPLIANCE REQUIREMENTS

Selective assets described by this document have specific compliance requirements. For instance, the Cypress WIFI driver will have compliance requirements for WIFI. The same will be true of the assets that support BlueTooth and BLE. Each of these assets is responsible for describing compliance strategy, both development and test in the associated IROS.

11. RECORDS

Storage location and retention period for records is specified in procedure 00-00064 Cypress Record Retention Policy.

12. PREVENTIVE MAINTENANCE: N/A

13. POSTING SHEETS/FORMS/APPENDIX

This section will contain any appendices required to further explain any particular section of the software IROS.

Document History Page

Rev.	ECN No.	Orig. of Change	Description of Change
**	7652392	MNO	Initial creation.
*A	7794181	MNO	Updates for PR2. See change bars.
*B	7813880	MNO	Revised most of the document to account for design changes leading up to PR3 and ES100.
*C	7872029	MNO	Updates for MTB 3.1 PR1. • Information about LCS (in several different sections) • External Tool Support • Capability descriptions and manifest • Added section for GlobalConentMgr • Centralized Settings • Various minor edits for clarification
*D	7908352	MNO	Updates for MTB 3.1 ES100 • Expanded ManifestDb information • Updated and expanded pseudo variable information • marked offline mode as deprecated • updated "new offline" mode terminology to "LCS" • explanded and clarified LCS mode usage and implementation • updated external tool support design and search algorithm • described the tool loading process • updated capability matching rules, including sensitivity lists • added usage model for centralized settings • documented design changes for centralized settings • minor updates to mtbgetlibs description • documented document generation in mtblaunch • updated command line args for lcs-manager-cli • generate edits for clarity and format throughout
*E	7943654	MNO	Updates for MTB 3.2 PR1 • Minor edits for wording • Placeholder entry support

Document History Page

Rev.	ECN No.	Orig. of Change	Description of Change
			• Clarifications on MTB URL protocols • Added --create-dependencies to mtbsearch documentation • Described the output for mtbsearch • Described the dependency rules for mtbsearch • Updated “Latest Locking” section with description of the new assetlocks.json file
*F	8007551	MNO	• Minor changes for wording and clarification throughout • Updated information about Device Support Files in overview • Added section 8.1.27 Device Support Files • Updated out of date information about error handling • Updated section 8.1.9 with the new QueryAPI cache behavior • Added section 8.1.13.4 to describe tool properties (brought over from the SAS) • Updated section 8.1.15.2.2.4 to describe the device_data and device_support manifest db attributes • Added section 8.1.16.2 to fill in missing details about the AppInfo and ProjectInfo APIs • Updated information about placeholders to deal with Early Access Packs • Updated section 8.1.22 to document the new latest locking history file • Described the LocalContentManager mode • Added section 8.2.1.3.1 to explain mtb_search caching • Updated section 8.2.2.1 to explain when we skipping fetching with git
*G	8076003	MNO	• Added export-memory property to tools props.json. • Extended opt.documentation to all types of props.json files and improved description of opt.documentation property. • Updated opt.configurator_support property information.

Document History Page

Rev.	ECN No.	Orig. of Change	Description of Change
			• Reworded the section about retrieving content to clarify the use of the force variable. • Minor clarification for use of --library. • Updated dependency handling in mtbsearch. • Added mtbninja section.
*H	8095702	MNO	• Documented INSTEADOF use for git • Updated Settings API to document getPresets • Clarified MTB_DEVICE_LIBRARY_PATHS delimiter • Added Alternative Server support • Reworded and expanded the mtbninja section
*I	8117501	MNO	• Clarified how the effective capabilities set is formed • Rewrote the Centralized Settings section based on the new behavior (MTB 3.5) • Removed the incorrect mtbquery command line arguments for managing settings • Added the mtbarchive tool section