Running ModusToolbox™ from the Command Line

Version 2.0

Introduction

The ModusToolbox Command Line Interface (CLI) is a GNU make-based build system that is used to perform application builds, as well as provide the logic required to launch tools and run utilities. It consists of a light and accessible set of make files deployed as part of every application. This structure allows each application to own the build process, and it allows environment-specific or application-specific changes to be made with relative ease. The system runs on any environment that has the make and git utilities.

The ModusToolbox Integrated Development Environment (IDE) uses this build system. Hence, switching from CLI to IDE and back is fully supported. Program/Debug and running tools can be done in either flow. In both cases, the build system relies on the presence of ModusToolbox tools that are brought into the environment by the ModusToolbox installer.

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

The majority of the make files are deployed as git repos, in the same way that libraries are deployed in the ModusToolbox software. The library that contains the recipe make files is referred to as the "base library." This is the minimum required library to enable an application build. Together, these make files form the build system.

This document covers the CLI usage of the build system. It is structured as follows:

Getting Started

Before initiating an application build, ensure that the environment is set up correctly.

1. Download and install the ModusToolbox software from the installer.
2. Ensure that the make and git programs are available in the system PATH.
   - Windows: Navigate to the modus-shell directory (Default: *<user_home>/ModusToolbox/tools_<version>/modus-shell*) and run *Cygwin.bat*. Alternatively, use your own Cygwin shell distribution. (**Note** Other shells are not supported.)
   - Linux/macOS: Open a terminal/shell.
3. Type `which make`. For most environments, it should return */usr/bin/make*.
4. Type `which git`. For most environments, it should return */usr/bin/git*.

If steps 3 and 4 do not return any paths (that is, in a customized environment), acquire the GNU make and git packages and include them in the system path. If all is successful, retrieve an application to start development using the chosen shell.

git clone

Applications are available on GitHub, and you may download or clone them onto your machine. If cloning, open the shell and type in the following command (replace the <URL> with the copied URL of the repo from GitHib):

```
git clone <URL>
```

The clone operation should return successfully. If it does not, check your internet connection/settings. Navigate to the directory that was cloned and find the application make file named "Makefile" or "makefile". This is the top-level application make file that determines the application build flow. In many cases, this will be the only file that needs to be modified. To skip ahead and see the available make targets and variables, refer to Available Make Targets and Available Make Variables.

You can alternately use the ModusToolbox project-creator utility (located in *<install_dir>/tools_x.y/project-creator*). This tool allows you to select a BSP and starter application. It then performs the `git clone` operation and the `make getlibs` operation. Refer to the *Project Creator Guide* for more details.

make getlibs

Change directory to where the application Makefile exists (`cd <DIR>`), and then type the following in the shell:

```
make getlibs
```

This instructs the build system to search for all *.lib* files in the application directory. These are special files that contain the libraries' git URL on which the application depends. These files are parsed, and the libraries are cloned into the application (by default in a directory named "libs").

Note Any .lib file that is not a text file (for instance Windows *.lib* archive files) are ignored for this process.

Note The `make getlibs` operation may take a long time to execute as it depends on your internet speed and the size of the libraries that it is cloning. To improve subsequent library cloning operations, a cache directory named ".modustoolbox" exists in the $HOME (Linux, macOS) and $USERPROFILE (Windows) directories.

You can alternately use the ModusToolbox library-manager utility (located in *<install_dir>/tools_x.y/library-manager*). This tool allows you to select BSPs and libraries. It then performs the `make getlibs` operation. Refer to the *Library Manager User Guide* for more details.

make build

When all the libraries are available, the application is ready to build. From the shell, type in the following:

```
make build
```

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

```
make build -j16
```

make program

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

```
make program
```

This performs an application build and then programs the application artifact (usually *.elf* or *.hex* file) to the board using the recipe-specific programming routine (usually OpenOCD). You may also skip the build step by using `qprogram` instead of `program`.

make help

The help documentation is readily available as a dedicated make target. From the shell, type in the following:

```
make help
```

This prints the available make targets and variables to the console. In order to access the verbose documentation of any of these targets or variables, specify it using the `CY_HELP` variable. For example:

```
make help CY_HELP=TOOLCHAIN
```

Note This help documentation is not available until after performing the `make getlibs` operation, as it is part of the base library. This target may also contain additional information specific to a BSP or a recipe.

The make variables and targets that the application should care about are documented in this help. Use the verbose documentation to get extended information about that topic.

Application Types

The build system supports the following application types:

- Normal app – The application consists of one application Makefile. The build process creates one artifact. All prebuilt libraries are brought in at link time. A normal application is constructed by defining the APPNAME variable in the application Makefile.
- Library app – The application consists of one application Makefile. The sources are built into a library. These libraries may be linked in as part of a Normal app build. A library application is constructed by defining the LIBNAME variable in the application Makefile.

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

```
SEARCH_LIBS_AND_INCLUDES=../bspLib
shared_libs:
    make -C ../bspLib build -j
```

Boards

An application must specify a target board through the `TARGET` variable. Cypress provides reference BSP libraries for its development kits. Use these as a reference to construct your own board files. Each BSP contains the following:

- *<BSP_NAME>.mk* – The board make file defines the board-specific information such as the device ID, compiler and linker flags, pre-builds/post-builds, and components used with this board implementation.
- *COMPONENT_BSP_DESIGN_MODUS/design.modus* – This is a configuration file (other types may also exist in a BSP) used to define the board peripherals and system settings using a graphical configuration tool (**Note** The “COMPONENT_BSP_DESIGN_MODUS” directory may not exist on all BSPs).
- (Optional) Linker file – Defines the memory layout of the application for the chosen device.
- (Optional) Startup file – Defines the startup sequence for the application.

To construct your own board, use one of the following options:

- Use an existing *TARGET* board, but update the *design.modus* configuration. This can be updated directly, or another file can be used instead of the default. For the latter case, set the make variable `DISABLE_COMPONENTS=BSP_DESIGN_MODUS` in the application Makefile. This will disable the inclusion of the default *design.modus* and its generated sources into the build (**Note** This mechanism is not applicable for BSPs that do not have the “COMPONENT_BSP_DESIGN_MODUS” directory).

- Create a new target by copying the *TARGET_<NAME>* board and renaming it. (**Note** As best practice, this should be placed in the same directory as other TARGETs). Then rename the <BSP_NAME>*.mk* file to match the new target name. For example, *TARGET_CustomBSP1/CustomBSP1.mk*.

To bring in existing boards into an application, use the library-manager utility or manually add a .lib that contains the URL and a version tag of interest in the application.

Library Files

The main mechanism for bringing source files into the application is by using *.lib* files. These are files that contain a git URL and a specific tag to a library repo. When the `getlibs` target is run, the build system finds these files in the application directory and performs git clone operations on them.

libs

The *.lib* files provided by Cypress always point to a Cypress owned repo or a repo that is sanctioned by Cypress. These files can be modified to point to specific tags or rerouted to internal repos. Cypress uses tags to denote versions when publishing content. To lock down to a certain version of a library, ensure that the tag in the .lib file points to a specific version. You may use the library manager to achieve this.

All libraries are cloned by default into a *libs* directory in the application root. This location can be modified by specifying the `CY_GETLIBS_PATH` variable. Duplicate libraries are checked to see if they point to the same commit and if so, only one copy is kept in the *libs* directory.

getlibs

The `getlibs` target finds and processes all *.lib* files and uses the `git` command to clone or pull the code as appropriate. Then, it checks out the specific tag listed in the *.lib* file. The ModusToolbox new project flow and the Library Manager invoke this process automatically.

- The `getlibs` target must be invoked separately from any other make target (for example, the command `make getlibs build` is not allowed and the make files will generate an error).
- The git clone operation uses the name “cypress” to track the upstream repository.
- The `getlibs` target performs a `git fetch` on existing libraries but will always checkout the tag pointed to by the overseeing *.lib* file.
- The `getlibs` target detects if users have modified the Cypress code and will not clobber their work. This allows you to perform some action (commit code, revert changes, as appropriate) instead of overwriting the changes.

The build system also has a `printlibs` target that can be used to print the status of the cloned libraries.

repos

The cloned libraries are situated in their individual git repos in the directory pointed to by the `CY_GETLIBS_PATH` variable (e.g. /libs). These all point to the “cypress” remote origin. This can be changed to point to your repo by editing the .git/config file or by running the git remote command.

If the repos are modified, add the changes to your source control (git branch is recommended). When getlibs is run (to either add new libraries or update libraries), it requires the repos to be clean. You may also use the .gitignore file for adding untracked files when running getlibs.

Adding source files

Source and header files placed in the application directory are automatically picked up by the auto-discovery mechanism and built. Any object file not referenced by the application are discarded by the linker. Similarly, library archives and object files are automatically picked up and linked.

To control which files are included/excluded, the build system implements a filtering mechanism based on directory names and *.cyignore* files. Refer to the Auto-Discovery section for more details.

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

Auto-Discovery

The build system implements auto-discovery of Cypress library files, source files, header files, object files, and pre-built libraries. If these files follow the specified rules, they are guaranteed to be brought into the application build automatically. This auto-discovery mechanism works on the application directory (and shared libraries) only. If files external to the application directory need to be added, they can be specified using the `SOURCES`, `INCLUDES`, and `LIBS` make variables.

Auto-discovery searches for all supported file types in the application directory and performs filtering based on a directory naming convention and specified directories, as well as files to ignore.

cyignore

Prior to applying auto-discovery and filtering, the build system will first search for *.cyignore* files and construct a set of directories and files to exclude. The *.cyignore* file contains a set of directories and files to exclude, relative to the location of the file. The `CY_IGNORE` variable can also be used to define directories and files to exclude.

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

TOOLCHAIN_<NAME>

Any directory that has the prefix “TOOLCHAIN_” is interpreted as a directory that is toolchain specific. The “NAME” corresponds to the value stored in the `TOOLCHAIN` make variable. For example, an IAR-specific set of files is located under a directory named *TOOLCHAIN_IAR*. Auto-discovery filters out all other *TOOLCHAIN_<NAME>* directories such as *TOOLCHAIN_GCC_ARM* and *TOOLCHAIN_ARM*.

TARGET_<NAME>

Any directory that has the prefix “TARGET_” is interpreted as a directory that is target specific. The “NAME” corresponds to the value stored in the `TARGET` make variable. For example, a build with `TARGET=CY8CPROTO-062-4343W` ignores all *TARGET_* directories except *TARGET_CY8CPROTO-062-4343W*.

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

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

CONFIG_<NAME>

Any directory that has the prefix “CONFIG_” is interpreted as a directory that is configuration (Debug/Release) specific. The “NAME” corresponds to the value stored in the `CONFIG` make variable. For example, a build with `CONFIG=CustomBuild` ignores all *CONFIG_* directories, except *CONFIG_CustomBuild*.

Note The output directory structure includes the *CONFIG* name in the path, so you can build for config A and B and both artifact files will exist on disk.

COMPONENT_<NAME>

Any directory that has the prefix "COMPONENT_" is interpreted as a directory that is component specific. The "NAME" corresponds to the value stored in the `COMPONENT` make variable. For example, consider an application that sets `COMPONENTS+=comp1`. Also assume that there are two directories containing component-specific sources:

COMPONENT_comp1/src.c

COMPONENT_comp2/src.c

Auto-discovery will only include *COMPONENT_comp1/src.c* and ignore *COMPONENT_comp2/src.c*. If a specific component needs to be removed, either delete it from the `COMPONENTS` variable or add it to the `DISABLE_COMPONENTS` variable.

BSP Make File

Auto-discovery will also search for a *<TARGET>.mk* file (aka BSP make file). If no matching *TARGET* make file is found, it will fail to build. This make file can also be manually specified by setting it in the `CY_EXTRA_INCLUDES` variable.

Pre-builds and Post-builds

Pre-builds and post-builds are possible at several stages in the build process. They can be specified at the application, BSP, and recipe levels. The sequence of execution in a build is as follows:

1. BSP pre-build – Defined using `CY_BSP_PREBUILD` variable.
2. Application pre-build – Defined using `PREBUILD` variable.
3. Source generation – Defined using `CY_RECIPE_GENSRC` variable.
4. Recipe pre-build – Defined using `CY_RECIPE_PREBUILD` variable.
5. Source Compilation and linking
6. Recipe post-build – Defined using `CY_RECIPE_POSTBUILD` variable.
7. BSP post-build – Defined using `CY_BSP_POSTBUILD` variable.
8. Application post-build – Defined using `POSTBUILD` variable.

Note Pre-builds happen after the auto-discovery process. Therefore, if the pre-build steps generate any source files to be included in a build, they will not be automatically included until the subsequent build. In this scenario, this step should use the `$(shell)` function directly in the application Makefile instead of using the provided pre-build make variables. For example:

```
$(shell bash ./custom_gen.sh ARG1 ARG2)
```

Program and debug

The programming step can be done through the CLI by using the following make targets:

- `program` – Build and program the board.
- `qprogram` – Skip the build step and program the board.
- `debug` – Build and program the board. Then launch the GDB server.
- `qdebug` – Skip the build step and program the board. Then launch the GDB server.

- `attach` – Starts a GDB client and attaches the debugger to the running target.

For CLI debugging, the attach target must be run on a separate shell instance. Use the GDB commands to debug the application.

Available Make Targets

A make target specifies the type of function or activity that the make invocation executes. Although multiple targets can be specified in a single invocation, the build system does not support such flows. Therefore, the targets should be called in separate make invocations. The following tables list and describe the available make targets for all recipes.

General Make Targets

Target	Description
`all`	Same as build. That it, builds the application. This target is equivalent to the `build` target.
`getlibs`	Clones the repositories and checks out the identified commit. The repos are cloned to the *libs* directory. By default, this directory is created in the application directory. It may be directed to other locations using the `CY_GETLIBS_PATH` variable.
`build`	Builds the application. The build process involves source auto-discovery, code-generation, pre-builds, and post-builds. For faster incremental builds, use the `qbuild` target to skip the auto-discovery step.
`qbuild`	Quick builds the application using the previous build's source list. When no other sources need to be auto-discovered, this target can be used to skip the auto-discovery step for a faster incremental build.
`program`	Builds the artifact and programs it to the target device. The build process performs the same operations as the `build` target. Upon successful completion, the artifact is programmed to the board.
`qprogram`	Quick programs a built application to the target device without rebuilding. This target allows programming an existing artifact to the board without a build step.
`debug`	Builds and programs. Then launches a GDB server. Once the GDB server is launched, another shell should be opened to launch a GDB client.
`qdebug`	Skips the build and program step and does Quick Debug; that is, it launches a GDB server. Once the GDB server is launched, another shell should be opened to launch a GDB client.
`clean`	Cleans the */build/<TARGET>* directory. The directory and all its contents are deleted from disk.
`help`	Prints the help documentation. Use the `CY_HELP=<NAME of target or variable>` to see the verbose documentation for a given target or a variable.

Tools Make Targets

Target	Description
`open`	Opens/launches a specified tool. This target accepts two variables: `CY_OPEN_TYPE` and `CY_OPEN_FILE`. At least one of these must be provided. The tool can be specified by setting the `CY_OPEN_TYPE` variable. A specific file can also be passed using the `CY_OPEN_FILE` variable. If only `CY_OPEN_FILE` is given, the build system will launch the default tool associated with the file's extension.

Target	Description
`config`	Runs the Device Configurator on the target **.modus* file. If no existing device-configuration files are found, the configurator is launched to create one.
`config_bt`	Runs the Bluetooth Configurator on the target **.cybt* file. If no existing bt-configuration files are found, the configurator is launched to create one.
`config_usbdev`	Runs the USB Configurator on the target **.cyusbdev* file. If no existing usbdev-configuration files are found, the configurator is launched to create one.

Utility Make Targets

Target	Description
`eclipse`	Generates Eclipse IDE launch configs. This target expects the CY_IDE_PRJNAME variable to be set to the name of the project as defined in the eclipse IDE. For example, `make eclipse CY_IDE_PRJNAME=AppV1`. If this variable is not defined, it will use the `APPNAME` variable for the launch configs.
`check`	Checks for the necessary tools. Not all tools are necessary for every build recipe. This target allows you to get an idea of which tools are missing if a build fails in an unexpected way.
`get_app_info`	Prints the app info for the eclipse IDE. As with the `get_cfg_file` target, the file types can be specified by setting the `CY_CONFIG_FILE_EXT` variable. For example, `make get_app_info CY_CONFIG_FILE_EXT="modus cybt cyusbdev"`
`get_env_info`	Prints the make, git, and, app repo info. This allows a quick printout of the current app repo and the make and git tool locations and versions.
`printlibs`	Prints the status of the library repos. This target parses through the library repos and prints the SHA1 commit. It also shows whether the repo is clean (no changes) or dirty (modified or new files).

Available Make Variables

The following make variables provide access to most of the available features to customize your build. They can either be defined in the application Makefile or be passed through the make invocation. For example:

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

Basic Configuration Make Variables

Variable	Description
`TARGET`	Specifies the target board/kit (that is, BSP). For example, CY8CPROTO-062-4343W.
`APPNAME`	Specifies the name of the application. For example, AppV1. This variable signifies that the application builds an artifact intended for a target board. For applications that need to build into an archive (library), use the `LIBNAME` variable.
`LIBNAME`	Specifies the name of the library application. For example, LibV1. This variable signifies that the application builds an archive (library). These library applications can be added as dependencies to an artifact producing application using the `SEARCH_LIBS_AND_INCLUDES` variable.

Variable	Description
`TOOLCHAIN`	Specifies the toolchain used to build the application. For example, GCC_ARM.
`CONFIG`	Specifies the configuration option for the build [Debug Release]. The `CONFIG` variable is not limited to Debug/Release. It can be other values. However in those instances, the build system will not configure the optimization flags. Debug=lowest optimization, Release=highest optimization. The optimization flags are toolchain specific. If you go with your custom config then you can manually set the optimization flag in the CFLAGS.
`VERBOSE`	Specifies whether the build is silent or verbose [true false]. Setting `VERBOSE` to true may help in debugging build errors/warnings.

Advanced Configuration Make Variables

Variable	Description
`SOURCES`	Specifies C/C++ and assembly files not under the working directory. This can be used to include files external to the application directory.
`INCLUDES`	Specifies include paths not under the working directory. **Note** These MUST NOT have `-I` prepended.
`DEFINES`	Specifies additional defines passed to the compiler. **Note** These MUST NOT have `-D` prepended.
`VFP_SELECT`	Selects hard/soft ABI for floating-point operations [softfp hardfp]. If not defined, this value defaults to softfp.
`CFLAGS`	Prepends additional C compiler flags. **Note** If the entire C compiler flags list needs to be replaced, define the `CY_RECIPE_CFLAGS` make variable with the desired C flags.
`CXXFLAGS`	Prepends additional C++ compiler flags. **Note** If the entire C++ compiler flags list needs to be replaced, define the `CY_RECIPE_CXXFLAGS` make variable with the desired C++ flags.
`ASFLAGS`	Prepends additional assembler flags. **Note** If the entire assembler flags list needs to be replaced, define the `CY_RECIPE_ASFLAGS` make variable with the desired assembly flags.
`LDFLAGS`	Prepends additional linker flags. **Note** If the entire linker flags list needs to be replaced, define the `CY_RECIPE_LDFLAGS` make variable with the desired linker flags.
`LDLIBS`	Includes application-specific prebuilt libraries. **Note** If additional libraries need to be added using `-l` or `-L`, add to the `CY_RECIPE_EXTRA_LIBS` make variable.
`LINKER_SCRIPT`	Specifies a custom linker script location. This linker script overrides the default. **Note** Additional linker scripts can be added for GCC via the `LDFLAGS` variable as a `-L` option.
`PREBUILD`	Specifies the location of a custom pre-build step and its arguments. This operation runs before the build recipe's pre-build step. **Note** BSPs can also define a pre-build step. This runs before the application pre-build step. If the default pre-build step needs to be replaced, define the `CY_RECIPE_PREBUILD` make variable with the desired pre-build step.

Variable	Description
`POSTBUILD`	Specifies the location of a custom post-build step and its arguments. This operation runs after the build recipe's post-build step. **Note** BSPs can also define a post-build step. This runs before the application post-build step. **Note** If the default post-build step needs to be replaced, define the `CY_RECIPE_POSTBUILD` make variable with the desired post-build step.
`COMPONENTS`	Adds component-specific files to the build. Create a directory named *COMPONENT_<VALUE>* and place your files. Then provide `<VALUE>` to this make variable to have that feature library be included in the build. For example, create a directory named *COMPONENT_ACCELEROMETER*. Then include it in the make variable: `COMPONENT=ACCELEROMETER`. If the make variable does not include the `<VALUE>`, then that library will not be included in the build. **Note** If the default `COMPONENT` list must be overridden, define the `CY_COMPONENT_LIST` make variable with the list of component values.
`DISABLE_COMPONENTS`	Removes component-specific files from the build. Include a `<VALUE>` to this make variable to have that feature library be excluded in the build. For example, to exclude the contents of the *COMPONENT_BSP_DESIGN_MODUS* directory, set `DISABLE_COMPONENTS=BSP_DESIGN_MODUS`.

`SEARCH_LIBS_AND_INCLUDES`	List of dependent library application paths. For example, *../bspLib*.
	An artifact-producing application (defined by setting `APPNAME`) can have a dependency on library applications (defined by setting `LIBNAME`). This variable defines those dependencies for the artifact-producing application. The actual build invocation of those libraries is handled at the application level by defining the `shared_libs` target. For example:
	`shared_libs:`
	`    make -C ../bspLib build -j`

Path Make Variables

Variable	Description
`CY_APP_PATH`	Relative path to the top-level of application. For example, *./* Settings this path to other than *./* allows the auto-discovery mechanism to search from a root directory location that is higher than the app directory. For example, `CY_APP_PATH=../../` allows auto-discovery of files from a location that is two directories above the location of *./Makefile*.
`CY_BASELIB_PATH`	Relative path to the base library. For example, *./libs/psoc6make* This directory must be relative to `CY_APP_PATH`. It defines the location of the library containing the recipe make files, where the expected directory structure is *<CY_BASELIB_PATH>/make*. All applications must set the location of the base library.
`CY_EXTAPP_PATH`	Relative path to an external app directory. For example, *../external* This directory must be relative to `CY_APP_PATH`. Setting this path allows incorporating files external to `CY_APP_PATH`. For example, `CY_EXTAPP_PATH=../external` lets auto-discovery pull in the contents of *../external* directory into the build. **Note** This variable is only supported in CLI. Use the `shared_libs` mechanism and `CY_HELP_SEARCH_LIBS_AND_INCLUDES` for tools and IDE support.

Variable	Description
`CY_GETLIBS_PATH`	Absolute path to the intended location of libs directory. The library repos are cloned into a directory named, *libs* (default: `<CY_APP_PATH>/libs`). Setting this variable allows specifying the location of the *libs* directory to be elsewhere on disk.
`CY_GETLIBS_SEARCH_PATH`	Relative path to the top directory for `getlibs` operation. The `getlibs` operation by default executes at the location of the `CY_APP_PATH`. This can be overridden by specifying this variable to point to a specific location.
`CY_DEVICESUPPORT_PATH`	Relative path to the *devicesupport.xml* file. This path specifies the location of the *devicesupport.xml* file for the Device Configurator. It is used when the configurator needs to be run in a multi-app scenario.
`CY_SHARED_PATH`	Relative path to the location of shared *.lib* files. This variable is used in shared library applications to point to the location of external *.libs* files.
`CY_COMPILER_PATH`	Absolute path to the compiler (default: `GCC_ARM` in `CY_TOOLS_DIR`). Setting this path allows custom toolchains to be used instead of the defaults. This should be the location of the */bin* directory containing the compiler, assembler, and linker. For example: `CY_COMPILER_PATH="C:/Program Files (x86)/IAR Systems/Embedded Workbench 8.2/arm/bin"`
`CY_TOOLS_DIR`	Absolute path to the tools root directory. Applications must specify the *tools_<version>* directory location, which contains the root make file and the necessary tools and scripts to build an application. Application make files are configured to automatically search in the standard locations for various platforms. If the tools are not located in the standard location, you may explicitly set this.
`CY_BUILD_LOCATION`	Absolute path to the build output directory (default: `pwd/build`). The build output directory is structured as */TARGET/CONFIG/*. Setting this variable allows the build artifacts to be located in the directory pointed to by this variable.

Miscellaneous Make Variables

Variable	Description
`CY_IGNORE`	Adds to the directory and file ignore list. E.g. ./file1.c ./inc1 Directories and files listed in this variable are ignored in auto-discovery. This mechanism works in combination with any existing *.cyignore* files in the application.
`CY_IDE_PRJNAME`	Name of the Eclipse IDE project. This variable can be used to define the file and target project name when generating Eclipse launch configurations in the `eclipse` target.
`CY_CONFIG_FILE_EXT`	Specifies the configurator file extension. E.g. modus This variable accepts a space-separated list of configurator file extensions to search when running the `get_cfg_file` and `get_app_info` targets.
`CY_SKIP_RECIPE`	Skip including the recipe make files. This allows the application to not include any recipe make files and only include the *start.mk* file from the tools install.
`CY_SUPPORTED_TOOL_TYPES`	Defines the supported tools for a BSP. BSPs can define the supported tools that can be launched using the `open` target.

Variable	Description
`CY_LIBS_SEARCH_DEPTH`	Directory search depth for *.lib* files (default: 5). This variable controls how deep the search mechanism in `getlibs` looks for *.lib* files. **Note** Deeper searches take longer to process.
`CY_UTILS_SEARCH_DEPTH`	Directory search depth for *.cyignore* and *TARGET.mk* files (default: 5). This variable controls how deep the search mechanism looks for *.cyignore* and *TARGET.mk* files. Min=1, Max=9. **Note** Deeper searches take longer to process.
`CY_EXTRA_INCLUDES`	Specifies additional make files to add to the build. The application make file cannot add additional make files directly. Instead, use this variable to include these in the build. For example: `CY_EXTRA_INCLUDES=./custom1.mk ./custom2.mk`
`TOOLCHAIN_MK_PATH`	Specifies the location of a custom *TOOLCHAIN.mk* file. Defining this path allows the build system to use a custom *TOOLCHAIN.mk* file pointed to by this variable. **Note** The make variables in this file should match the variables used in existing *TOOLCHAIN.mk* files.

Troubleshooting on Windows

There are possible issues you might see on Windows with regards to the ModusToolbox modus-shell and a separate installation of Cygwin on your machine.

HOME Directory

The modus-shell environment uses your Windows profile directory as its HOME. This is controlled by */etc/nsswitch.conf* ("db_home" is set to "windows"). For more information see:

https://cygwin.com/cygwin-ug-net/ntsec.html#ntsec-mapping-nsswitch-home

Permissions

The modus-shell environment is configured to let Windows manage permissions. This ensures Windows applications (like ModusToolbox IDE) don't have issues with files created by modus-shell.

This can cause issues if you have another Cygwin environment pre-installed, and if you switch between them. For example, `git status` will report that file permissions have changed because the two environments handle permissions differently.

Cypress recommends that you don't mix *NIX environments on Windows (that is, do not use more than one Cygwin, or Cygwin and GitBash/msys/mingw).

Permission behavior is controlled by */etc/fstab* (specifically, the “noacl” option). For more information see:

https://cygwin.com/cygwin-ug-net/ntsec.html#ntsec-files

If you must use more than one Cygwin environment, make sure both Cygwin instances use consistent "noacl" settings in their */etc/fstab* files.