cmake documentation

==========

Command-Line Tools¶

cmake(1)

Synopsis

```
cmake [<options>] {<path-to-source> | <path-to-existing-build>}
cmake [<options>] -S <path-to-source> -B <path-to-build>
cmake [{-D <var>=<value>}...] -P <cmake-script-file>
cmake --build <dir> [<options>...] [-- <build-tool-options>...]
cmake --open <dir>
cmake -E <command> [<options>...]
cmake --find-package <options>...
```

Description

The “cmake” executable is the CMake command-line interface. It may be used to configure projects in scripts. Project configuration settings may be specified on the command line with the -D option.

CMake is a cross-platform build system generator. Projects specify their build process with platform-independent CMake listfiles included in each directory of a source tree with the name CMakeLists.txt. Users build a project by using CMake to generate a build system for a native tool on their platform.

Options

- `-S <path-to-source>`

  Path to root directory of the CMake project to build.

- `-B <path-to-build>`

  Path to directory which CMake will use as the root of build directory.If the directory doesn’t already exist CMake will make it.

- `-C <initial-cache>`

Pre-load a script to populate the cache.When cmake is first run in an empty build tree, it creates a CMakeCache.txt file and populates it with customizable settings for the project. This option may be used to specify a file from which to load cache entries before the first pass through the project's cmake listfiles. The loaded entries take priority over the project's default values. The given file should be a CMake script containing SET commands that use the CACHE option, not a cache-format file.

- `-D <var>:<type>=<value>, -D <var>=<value>`

  Create or update a cmake cache entry.When cmake is first run in an empty build tree, it creates a CMakeCache.txt file and populates it with customizable settings for the project. This option may be used to specify a setting that takes priority over the project's default value. The option may be repeated for as many cache entries as desired.If the `:<type>` portion is given it must be one of the types specified by the `set()` command documentation for its `CACHE` signature. If the `:<type>` portion is omitted the entry will be created with no type if it does not exist with a type already. If a command in the project sets the type to `PATH` or `FILEPATH` then the `<value>` will be converted to an absolute path.This option may also be given as a single argument: `-D<var>:<type>=<value>` or `-D<var>=<value>`.

- `-U <globbing_expr>`

  Remove matching entries from CMake cache.This option may be used to remove one or more variables from the CMakeCache.txt file, globbing expressions using * and ? are supported. The option may be repeated for as many cache entries as desired.Use with care, you can make your CMakeCache.txt non-working.

- `-G <generator-name>`

  Specify a build system generator.CMake may support multiple native build systems on certain platforms. A generator is responsible for generating a particular build system. Possible generator names are specified in the `cmake-generators(7)` manual.

- `-T <toolset-spec>`

  Toolset specification for the generator, if supported.Some CMake generators support a toolset specification to tell the native build system how to choose a compiler. See the `CMAKE_GENERATOR_TOOLSET` variable for details.

- `-A <platform-name>`

  Specify platform name if supported by generator.Some CMake generators support a platform name to be given to the native build system to choose a compiler or SDK. See the `CMAKE_GENERATOR_PLATFORM` variable for details.

- `-Wno-dev`

  Suppress developer warnings.Suppress warnings that are meant for the author of the CMakeLists.txt files. By default this will also turn off deprecation warnings.

- `-Wdev`

  Enable developer warnings.Enable warnings that are meant for the author of the CMakeLists.txt files. By default this will also turn on deprecation warnings.

- `-Werror=dev`

  Make developer warnings errors.Make warnings that are meant for the author of the CMakeLists.txt files errors. By default this will also turn on deprecated warnings as errors.

- `-Wno-error=dev`

Make developer warnings not errors.Make warnings that are meant for the author of the CMakeLists.txt files not errors. By default this will also turn off deprecated warnings as errors.

- `-Wdeprecated`

  Enable deprecated functionality warnings.Enable warnings for usage of deprecated functionality, that are meant for the author of the CMakeLists.txt files.

- `-Wno-deprecated`

  Suppress deprecated functionality warnings.Suppress warnings for usage of deprecated functionality, that are meant for the author of the CMakeLists.txt files.

- `-Werror=deprecated`

  Make deprecated macro and function warnings errors.Make warnings for usage of deprecated macros and functions, that are meant for the author of the CMakeLists.txt files, errors.

- `-Wno-error=deprecated`

  Make deprecated macro and function warnings not errors.Make warnings for usage of deprecated macros and functions, that are meant for the author of the CMakeLists.txt files, not errors.

- `-E <command> [<options>...]`

  See Command-Line Tool Mode.

- `-L[A][H]`

  List non-advanced cached variables.List cache variables will run CMake and list all the variables from the CMake cache that are not marked as INTERNAL or ADVANCED. This will effectively display current CMake settings, which can then be changed with -D option. Changing some of the variables may result in more variables being created. If A is specified, then it will display also advanced variables. If H is specified, it will also display help for each variable.

- `--build <dir>`

  See Build Tool Mode.

- `--open <dir>`

  Open the generated project in the associated application. This is only supported by some generators.

- `-N`

  View mode only.Only load the cache. Do not actually run configure and generate steps.

- `-P <file>`

  Process script mode.Process the given cmake file as a script written in the CMake language. No configure or generate step is performed and the cache is not modified. If variables are defined using -D, this must be done before the -P argument.

- `--find-package`

  See Find-Package Tool Mode.

- `--graphviz=[file]`

  Generate graphviz of dependencies, see `CMakeGraphVizOptions` for more.Generate a graphviz input file that will contain all the library and executable dependencies in the project. See the documentation for `CMakeGraphVizOptions` for more details.

- `--system-information [file]`

Dump information about this system.Dump a wide range of information about the current system. If run from the top of a binary tree for a CMake project it will dump additional information such as the cache, log files etc.

- `--debug-trycompile`

  Do not delete the try_compile build tree. Only useful on one try_compile at a time.Do not delete the files and directories created for try_compile calls. This is useful in debugging failed try_compiles. It may however change the results of the try-compiles as old junk from a previous try-compile may cause a different test to either pass or fail incorrectly. This option is best used for one try-compile at a time, and only when debugging.

- `--debug-output`

  Put cmake in a debug mode.Print extra information during the cmake run like stack traces with message(send_error) calls.

- `--trace`

  Put cmake in trace mode.Print a trace of all calls made and from where.

- `--trace-expand`

  Put cmake in trace mode.Like `--trace`, but with variables expanded.

- `--trace-source=<file>`

  Put cmake in trace mode, but output only lines of a specified file.Multiple options are allowed.

- `--warn-uninitialized`

  Warn about uninitialized values.Print a warning when an uninitialized variable is used.

- `--warn-unused-vars`

  Warn about unused variables.Find variables that are declared or set, but not used.

- `--no-warn-unused-cli`

  Don’t warn about command line options.Don’t find variables that are declared on the command line, but not used.

- `--check-system-vars`

  Find problems with variable usage in system files.Normally, unused and uninitialized variables are searched for only in CMAKE_SOURCE_DIR and CMAKE_BINARY_DIR. This flag tells CMake to warn about other files as well.

- `--help,-help,-usage,-h,-H,/?`

  Print usage information and exit.Usage describes the basic command line interface and its options.

- `--version,-version,/V [<f>]`

  Show program name/version banner and exit.If a file is specified, the version is written into it. The help is printed to a named ile if given.

- `--help-full [<f>]`

  Print all help manuals and exit.All manuals are printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-manual <man> [<f>]`

Print one help manual and exit.The specified manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-manual-list [<f>]`

  List help manuals available and exit.The list contains all manuals for which help may be obtained by using the `--help-manual` option followed by a manual name. The help is printed to a named ile if given.

- `--help-command <cmd> [<f>]`

  Print help for one command and exit.The `cmake-commands(7)` manual entry for `<cmd>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-command-list [<f>]`

  List commands with help available and exit.The list contains all commands for which help may be obtained by using the `--help-command` option followed by a command name. The help is printed to a named ile if given.

- `--help-commands [<f>]`

  Print cmake-commands manual and exit.The `cmake-commands(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-module <mod> [<f>]`

  Print help for one module and exit.The `cmake-modules(7)` manual entry for `<mod>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-module-list [<f>]`

  List modules with help available and exit.The list contains all modules for which help may be obtained by using the `--help-module` option followed by a module name. The help is printed to a named ile if given.

- `--help-modules [<f>]`

  Print cmake-modules manual and exit.The `cmake-modules(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-policy <cmp> [<f>]`

  Print help for one policy and exit.The `cmake-policies(7)` manual entry for `<cmp>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-policy-list [<f>]`

  List policies with help available and exit.The list contains all policies for which help may be obtained by using the `--help-policy` option followed by a policy name. The help is printed to a named ile if given.

- `--help-policies [<f>]`

  Print cmake-policies manual and exit.The `cmake-policies(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-property <prop> [<f>]`

  Print help for one property and exit.The `cmake-properties(7)` manual entries for `<prop>` are printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-property-list [<f>]`

List properties with help available and exit.The list contains all properties for which help may be obtained by using the `--help-property` option followed by a property name. The help is printed to a named ile if given.

- `--help-properties [<f>]`

  Print cmake-properties manual and exit.The `cmake-properties(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-variable <var> [<f>]`

  Print help for one variable and exit.The `cmake-variables(7)` manual entry for `<var>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-variable-list [<f>]`

  List variables with help available and exit.The list contains all variables for which help may be obtained by using the `--help-variable` option followed by a variable name. The help is printed to a named ile if given.

- `--help-variables [<f>]`

  Print cmake-variables manual and exit.The `cmake-variables(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

Build Tool Mode

CMake provides a command-line signature to build an already-generated project binary tree:

```
cmake --build <dir> [<options>...] [-- <build-tool-options>...]
```

This abstracts a native build tool's command-line interface with the following options:

- `--build <dir>`

  Project binary directory to be built. This is required and must be first.

- `-j [<jobs>], --parallel [<jobs>]`

  The maximum number of concurrent processes to use when building. If `<jobs>` is omitted the native build tool's default number is used.The `CMAKE_BUILD_PARALLEL_LEVEL` environment variable, if set, specifies a default parallel level when this option is not given.

- `--target <tgt>`

  Build `<tgt>` instead of default targets. May only be specified once.

- `--config <cfg>`

  For multi-configuration tools, choose configuration `<cfg>`.

- `--clean-first`

  Build target `clean` first, then build. (To clean only, use `--target clean`.)

- `--use-stderr`

  Ignored. Behavior is default in CMake >= 3.0.

- `--`

  Pass remaining options to the native tool.

Run `cmake --build` with no options for quick help.

Command-Line Tool Mode

CMake provides builtin command-line tools through the signature:

```
cmake -E <command> [<options>...]
```

Run `cmake -E` or `cmake -E help` for a summary of commands. Available commands are:

- `capabilities`

  Report cmake capabilities in JSON format. The output is a JSON object with the following keys: `version` A JSON object with version information. Keys are: `string` The full version string as displayed by cmake `--version`. `major` The major version number in integer form. `minor` The minor version number in integer form. `patch` The patch level in integer form. `suffix` The cmake version suffix string. `isDirty` A bool that is set if the cmake build is from a dirty tree. `generators` A list available generators. Each generator is a JSON object with the following keys: `name` A string containing the name of the generator. `toolsetSupport``true` if the generator supports toolsets and `false` otherwise. `platformSupport``true` if the generator supports platforms and `false` otherwise. `extraGenerators` A list of strings with all the extra generators compatible with the generator. `serverMode``true` if cmake supports server-mode and `false` otherwise.

- `chdir <dir> <cmd> [<arg>...]`

  Change the current working directory and run a command.

- `compare_files <file1> <file2>`

  Check if `<file1>` is same as `<file2>`. If files are the same, then returns 0, if not it returns 1.

- `copy <file>... <destination>`

  Copy files to `<destination>` (either file or directory). If multiple files are specified, the `<destination>` must be directory and it must exist. Wildcards are not supported.

- `copy_directory <dir>... <destination>`

  Copy directories to `<destination>` directory. If `<destination>` directory does not exist it will be created.

- `copy_if_different <file>... <destination>`

  Copy files to `<destination>` (either file or directory) if they have changed. If multiple files are specified, the `<destination>` must be directory and it must exist.

- `echo [<string>...]`

  Displays arguments as text.

- `echo_append [<string>...]`

  Displays arguments as text but no new line.

- `env [--unset=NAME]... [NAME=VALUE]... COMMAND [ARG]...`

Run command in a modified environment.

- `environment`

  Display the current environment variables.

- `make_directory <dir>...`

  Create `<dir>` directories. If necessary, create parent directories too. If a directory already exists it will be silently ignored.

- `md5sum <file>...`

  Create MD5 checksum of files in `md5sum` compatible format: `351abe79cd3800b38cdfb25d45015a15 file1.txt 052f86c15bbde68af55c7f7b340ab639 file2.txt`

- `sha1sum <file>...`

  Create SHA1 checksum of files in `sha1sum` compatible format: `4bb7932a29e6f73c97bb9272f2bdc393122f86e0 file1.txt 1df4c8f318665f9a5f2ed38f55adadb7ef9f559c file2.txt`

- `sha224sum <file>...`

  Create SHA224 checksum of files in `sha224sum` compatible format: `b9b9346bc8437bbda630b0b7ddfc5ea9ca157546dbbf4c613192f930 file1.txt 6dfbe55f4d2edc5fe5c9197bca51ceaaf824e48eba0cc453088aee24 file2.txt`

- `sha256sum <file>...`

  Create SHA256 checksum of files in `sha256sum` compatible format: `76713b23615d31680afeb0e9efe94d47d3d4229191198bb46d7485f9cb191acc file1.txt 15b682ead6c12dedb1baf91231e1e89cfc7974b3787c1e2e01b986bffadae0ea file2.txt`

- `sha384sum <file>...`

  Create SHA384 checksum of files in `sha384sum` compatible format: `acc049fedc091a22f5f2ce39a43b9057fd93c910e9afd76a6411a28a8f2b8a12c73d7129e292f94fc0329c309df49434 file1.txt 668ddeb108710d271ee21c0f3acbd6a7517e2b78f9181c6a2ff3b8943af92b0195dcb7cce48aa3e17893173c0a39e23d file2.txt`

- `sha512sum <file>...`

  Create SHA512 checksum of files in `sha512sum` compatible format: `2a78d7a6c5328cfb1467c63beac8ff21794213901eaadafd48e7800289afbc08e5fb3e86aa31116c945ee3d7bf2a6194489ec6101051083d1108defc8e1dba89 file1.txt 7a0b54896fe5e70cca6dd643ad6f672614b189bf26f8153061c4d219474b05dad08c4e729af9f4b009f1a1a280cb625454bf587c690f4617c27e3aebdf3b7a2d file2.txt`

- `remove [-f] <file>...`

  Remove the file(s). If any of the listed files already do not exist, the command returns a non-zero exit code, but no message is logged. The `-f` option changes the behavior to return a zero exit code (i.e. success) in such situations instead.

- `remove_directory <dir>`

  Remove a directory and its contents. If a directory does not exist it will be silently ignored.

- `rename <oldname> <newname>`

Rename a file or directory (on one volume).

- `server`

  Launch `cmake-server(7)` mode.

- `sleep <number>...`

  Sleep for given number of seconds.

- `tar [cxt][vf][zjJ] file.tar [<options>...] [--] [<file>...]`

  Create or extract a tar or zip archive. Options are: `--` Stop interpreting options and treat all remaining arguments as file names even if they start in `-`. `--files-from=<file>` Read file names from the given file, one per line. Blank lines are ignored. Lines may not start in `-` except for `--add-file=<name>` to add files whose names start in `-`. `--mtime=<date>` Specify modification time recorded in tarball entries. `--format=<format>` Specify the format of the archive to be created. Supported formats are: `7zip`, `gnutar`, `pax`, `paxr` (restricted pax, default), and `zip`.

- `time <command> [<args>...]`

  Run command and display elapsed time.

- `touch <file>`

  Touch a file.

- `touch_nocreate <file>`

  Touch a file if it exists but do not create it. If a file does not exist it will be silently ignored.

- `create_symlink <old> <new>`

  Create a symbolic link `<new>` naming `<old>`.

Note

Path to where `<new>` symbolic link will be created has to exist beforehand.

Windows-specific Command-Line Tools

The following `cmake -E` commands are available only on Windows:

- `delete_regv <key>`

  Delete Windows registry value.

- `env_vs8_wince <sdkname>`

  Displays a batch file which sets the environment for the provided Windows CE SDK installed in VS2005.

- `env_vs9_wince <sdkname>`

  Displays a batch file which sets the environment for the provided Windows CE SDK installed in VS2008.

- `write_regv <key> <value>`

  Write Windows registry value.

Find-Package Tool Mode

CMake provides a helper for Makefile-based projects with the signature:

```
cmake --find-package <options>...
```

This runs in a pkg-config like mode.

Search a package using `find_package()` and print the resulting flags to stdout. This can be used to use cmake instead of pkg-config to find installed libraries in plain Makefile-based projects or in autoconf-based projects (via `share/aclocal/cmake.m4`).

Note

This mode is not well-supported due to some technical limitations. It is kept for compatibility but should not be used in new projects.

See Also

The following resources are available to get help using CMake:

- Home Page

  https://cmake.orgThe primary starting point for learning about CMake.

- Online Documentation and Community Resources

  https://cmake.org/documentationLinks to available documentation and community resources may be found on this web page.

- Mailing List

  https://cmake.org/mailing-listsFor help and discussion about using cmake, a mailing list is provided at cmake@cmake.org. The list is member-post-only but one may sign up on the CMake web page. Please first read the full documentation at https://cmake.org before posting questions to the list.

==========

ctest(1)

Contents

Synopsis

```
ctest [<options>]
ctest <path-to-source> <path-to-build> --build-generator <generator>
      [<options>...] [-- <build-options>...] [--test-command <test>]
ctest {-D <dashboard> | -M <model> -T <action> | -S <script> | -SP <script>}
      [-- <dashboard-options>...]
```

Description

The “ctest” executable is the CMake test driver program. CMake-generated build trees created for projects that use the ENABLE_TESTING and ADD_TEST commands have testing support. This program will run the tests and report results.

Options

- `-C <cfg>, --build-config <cfg>`

  Choose configuration to test.Some CMake-generated build trees can have multiple build configurations in the same tree. This option can be used to specify which one should be tested. Example configurations are “Debug” and “Release”.

- `--progress`

  Enable short progress output from tests.When the output of `ctest` is being sent directly to a terminal, the progress through the set of tests is reported by updating the same line rather than printing start and end messages for each test on new lines. This can significantly reduce the verbosity of the test output. Test completion messages are still output on their own line for failed tests and the final test summary will also still be logged.This option can also be enabled by setting the environment variable `CTEST_PROGRESS_OUTPUT`.

- `-V,--verbose`

  Enable verbose output from tests.Test output is normally suppressed and only summary information is displayed. This option will show all test output.

- `-VV,--extra-verbose`

  Enable more verbose output from tests.Test output is normally suppressed and only summary information is displayed. This option will show even more test output.

- `--debug`

  Displaying more verbose internals of CTest.This feature will result in a large number of output that is mostly useful for debugging dashboard problems.

- `--output-on-failure`

  Output anything outputted by the test program if the test should fail. This option can also be enabled by setting the `CTEST_OUTPUT_ON_FAILURE` environment variable

- `-F`

  Enable failover.This option allows CTest to resume a test set execution that was previously interrupted. If no interruption occurred, the `-F` option will have no effect.

- `-j <jobs>, --parallel <jobs>`

  Run the tests in parallel using the given number of jobs.This option tells CTest to run the tests in parallel using given number of jobs. This option can also be set by setting the `CTEST_PARALLEL_LEVEL` environment variable.This option can be used with the `PROCESSORS` test property.See Label and Subproject Summary.

- `--test-load <level>`

  While running tests in parallel (e.g. with `-j`), try not to start tests when they may cause the CPU load to pass above a given threshold.When `ctest` is run as a Dashboard Client this sets the `TestLoad` option of the CTest Test Step.

- `-Q,--quiet`

  Make CTest quiet.This option will suppress all the output. The output log file will still be generated if the `--output-log` is specified. Options such as `--verbose`, `--extra-verbose`, and `--debug` are ignored if `--quiet` is specified.

- `-O <file>, --output-log <file>`

  Output to log file.This option tells CTest to write all its output to a log file.

- `-N,--show-only`

  Disable actual execution of tests.This option tells CTest to list the tests that would be run but not actually run them. Useful in conjunction with the `-R` and `-E` options.

- `-L <regex>, --label-regex <regex>`

  Run tests with labels matching regular expression.This option tells CTest to run only the tests whose labels match the given regular expression.

- `-R <regex>, --tests-regex <regex>`

  Run tests matching regular expression.This option tells CTest to run only the tests whose names match the given regular expression.

- `-E <regex>, --exclude-regex <regex>`

  Exclude tests matching regular expression.This option tells CTest to NOT run the tests whose names match the given regular expression.

- `-LE <regex>, --label-exclude <regex>`

  Exclude tests with labels matching regular expression.This option tells CTest to NOT run the tests whose labels match the given regular expression.

- `-FA <regex>, --fixture-exclude-any <regex>`

  Exclude fixtures matching `<regex>` from automatically adding any tests to the test set.If a test in the set of tests to be executed requires a particular fixture, that fixture's setup and cleanup tests would normally be added to the test set automatically. This option prevents adding setup or cleanup tests for fixtures matching the `<regex>`. Note that all other fixture behavior is retained, including test dependencies and skipping tests that have fixture setup tests that fail.

- `-FS <regex>, --fixture-exclude-setup <regex>`

  Same as `-FA` except only matching setup tests are excluded.

- `-FC <regex>, --fixture-exclude-cleanup <regex>`

  Same as `-FA` except only matching cleanup tests are excluded.

- `-D <dashboard>, --dashboard <dashboard>`

  Execute dashboard test.This option tells CTest to act as a CDash client and perform a dashboard test. All tests are , where Mode can be Experimental, Nightly, and Continuous, and Test can be Start, Update, Configure, Build, Test, Coverage, and Submit.See Dashboard Client.

- `-D <var>:<type>=<value>`

  Define a variable for script mode.Pass in variable values on the command line. Use in conjunction with `-S` to pass variable values to a dashboard script. Parsing `-D` arguments as variable values is only attempted if the value following `-D` does not match any of the known dashboard types.

- `-M <model>, --test-model <model>`

  Sets the model for a dashboard.This option tells CTest to act as a CDash client where the `<model>` can be `Experimental`, `Nightly`, and `Continuous`. Combining `-M` and `-T` is similar to `-D`.See Dashboard Client.

- `-T <action>, --test-action <action>`

  Sets the dashboard action to perform.This option tells CTest to act as a CDash client and perform some action such as `start`, `build`, `test` etc. SeeDashboard Client Steps for the full list of actions. Combining `-M` and `-T` is similar to `-D`.See Dashboard Client.

- `-S <script>, --script <script>`

  Execute a dashboard for a configuration.This option tells CTest to load in a configuration script which sets a number of parameters such as the binary and source directories. Then CTest will do what is required to create and run a dashboard. This option basically sets up a dashboard and then runs `ctest -D` with the appropriate options.See Dashboard Client.

- `-SP <script>, --script-new-process <script>`

  Execute a dashboard for a configuration.This option does the same operations as `-S` but it will do them in a separate process. This is primarily useful in cases where the script may modify the environment and you do not want the modified environment to impact other `-S` scripts.See Dashboard Client.

- `-I [Start,End,Stride,test#,test#|Test file], --tests-information`

  Run a specific number of tests by number.This option causes CTest to run tests starting at number Start, ending at number End, and incrementing by Stride. Any additional numbers after Stride are considered individual test numbers. Start, End,or stride can be empty. Optionally a file can be given that contains the same syntax as the command line.

- `-U, --union`

Take the Union of `-I` and `-R`.When both `-R` and `-I` are specified by default the intersection of tests are run. By specifying `-U` the union of tests is run instead.

- `--rerun-failed`

  Run only the tests that failed previously.This option tells CTest to perform only the tests that failed during its previous run. When this option is specified, CTest ignores all other options intended to modify the list of tests to run (`-L`, `-R`, `-E`, `-LE`, `-I`, etc). In the event that CTest runs and no tests fail, subsequent calls to CTest with the `--rerun-failed` option will run the set of tests that most recently failed (if any).

- `--repeat-until-fail <n>`

  Require each test to run `<n>` times without failing in order to pass.This is useful in finding sporadic failures in test cases.

- `--max-width <width>`

  Set the max width for a test name to output.Set the maximum width for each test name to show in the output. This allows the user to widen the output to avoid clipping the test name which can be very annoying.

- `--interactive-debug-mode [0|1]`

  Set the interactive mode to 0 or 1.This option causes CTest to run tests in either an interactive mode or a non-interactive mode. On Windows this means that in non-interactive mode, all system debug pop up windows are blocked. In dashboard mode (Experimental, Nightly, Continuous), the default is non-interactive. When just running tests not for a dashboard the default is to allow popups and interactive debugging.

- `--no-label-summary`

  Disable timing summary information for labels.This option tells CTest not to print summary information for each label associated with the tests run. If there are no labels on the tests, nothing extra is printed.See Label and Subproject Summary.

- `--no-subproject-summary`

  Disable timing summary information for subprojects.This option tells CTest not to print summary information for each subproject associated with the tests run. If there are no subprojects on the tests, nothing extra is printed.See Label and Subproject Summary.

`--build-and-test` See Build and Test Mode.

- `--test-output-size-passed <size>`

  Limit the output for passed tests to `<size>` bytes.

- `--test-output-size-failed <size>`

  Limit the output for failed tests to `<size>` bytes.

- `--overwrite`

  Overwrite CTest configuration option.By default CTest uses configuration options from configuration file. This option will overwrite the configuration option.

- `--force-new-ctest-process`

  Run child CTest instances as new processes.By default CTest will run child CTest instances within the same process. If this behavior is not desired, this argument will enforce new processes for child CTest processes.

- `--schedule-random`

  Use a random order for scheduling tests.This option will run the tests in a random order. It is commonly used to detect implicit dependencies in a test suite.

- `--submit-index`

  Legacy option for old Dart2 dashboard server feature. Do not use.

- `--timeout <seconds>`

  Set a global timeout on all tests.This option will set a global timeout on all tests that do not already have a timeout set on them.

- `--stop-time <time>`

  Set a time at which all tests should stop running.Set a real time of day at which all tests should timeout. Example: `7:00:00 -0400`. Any time format understood by the curl date parser is accepted. Local time is assumed if no timezone is specified.

- `--print-labels`

  Print all available test labels.This option will not run any tests, it will simply print the list of all labels associated with the test set.

- `--help,-help,-usage,-h,-H,/?`

  Print usage information and exit.Usage describes the basic command line interface and its options.

- `--version,-version,/V [<f>]`

  Show program name/version banner and exit.If a file is specified, the version is written into it. The help is printed to a named ile if given.

- `--help-full [<f>]`

  Print all help manuals and exit.All manuals are printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-manual <man> [<f>]`

  Print one help manual and exit.The specified manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-manual-list [<f>]`

  List help manuals available and exit.The list contains all manuals for which help may be obtained by using the `--help-manual` option followed by a manual name. The help is printed to a named ile if given.

- `--help-command <cmd> [<f>]`

  Print help for one command and exit.The `cmake-commands(7)` manual entry for `<cmd>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-command-list [<f>]`

  List commands with help available and exit.The list contains all commands for which help may be obtained by using the `--help-command` option followed by a command name. The help is printed to a named ile if given.

- `--help-commands [<f>]`

  Print cmake-commands manual and exit.The `cmake-commands(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-module <mod> [<f>]`

  Print help for one module and exit.The `cmake-modules(7)` manual entry for `<mod>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-module-list [<f>]`

  List modules with help available and exit.The list contains all modules for which help may be obtained by using the `--help-module` option followed by a module name. The help is printed to a named ile if given.

- `--help-modules [<f>]`

  Print cmake-modules manual and exit.The `cmake-modules(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-policy <cmp> [<f>]`

  Print help for one policy and exit.The `cmake-policies(7)` manual entry for `<cmp>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-policy-list [<f>]`

  List policies with help available and exit.The list contains all policies for which help may be obtained by using the `--help-policy` option followed by a policy name. The help is printed to a named ile if given.

- `--help-policies [<f>]`

  Print cmake-policies manual and exit.The `cmake-policies(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-property <prop> [<f>]`

  Print help for one property and exit.The `cmake-properties(7)` manual entries for `<prop>` are printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-property-list [<f>]`

  List properties with help available and exit.The list contains all properties for which help may be obtained by using the `--help-property` option followed by a property name. The help is printed to a named ile if given.

- `--help-properties [<f>]`

  Print cmake-properties manual and exit.The `cmake-properties(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-variable <var> [<f>]`

  Print help for one variable and exit.The `cmake-variables(7)` manual entry for `<var>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-variable-list [<f>]`

  List variables with help available and exit.The list contains all variables for which help may be obtained by using the `--help-variable` option followed by a variable name. The help is printed to a named ile if given.

- `--help-variables [<f>]`

  Print cmake-variables manual and exit.The `cmake-variables(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

Label and Subproject Summary

CTest prints timing summary information for each label and subproject associated with the tests run. The label time summary will not include labels that are mapped to subprojects.

When the `PROCESSORS` test property is set, CTest will display a weighted test timing result in label and subproject summaries. The time is reported with sec*proc instead of just sec.

The weighted time summary reported for each label or subproject j is computed as:

```
Weighted Time Summary for Label/Subproject j =
    sum(raw_test_time[j,i] * num_processors[j,i], i=1...num_tests[j])

for labels/subprojects j=1...total
```

where:

- raw_test_time[j,i]: Wall-clock time for the ith test for the jth label or subproject
- num_processors[j,i]: Value of the CTest PROCESSORS property for the ith test for the jth label or subproject
- num_tests[j]: Number of tests associated with the jth label or subproject
- total: Total number of labels or subprojects that have at least one test run

Therefore, the weighted time summary for each label or subproject represents the amount of time that CTest gave to run the tests for each label or subproject and gives a good representation of the total expense of the tests for each label or subproject when compared to other labels or subprojects.

For example, if "SubprojectA" showed "100 sec*proc" and "SubprojectB" showed "10 sec*proc", then CTest allocated approximately 10 times the CPU/core time to run the tests for "SubprojectA" than for "SubprojectB" (e.g. so if effort is going to be expended to reduce the cost of the test suite for the whole project, then reducing the cost of the test suite for "SubprojectA" would likely have a larger impact than effort to reduce the cost of the test suite for "SubprojectB").

Build and Test Mode

CTest provides a command-line signature to configure (i.e. run cmake on), build, and/or execute a test:

```
ctest --build-and-test <path-to-source> <path-to-build>
      --build-generator <generator>
      [<options>...]
      [--build-options <opts>...]
      [--test-command <command> [<args>...]]
```

The configure and test steps are optional. The arguments to this command line are the source and binary directories. The `--build-generator` option *must* be provided to use `--build-and-test`. If `--test-command` is specified then that will be run after the build is complete. Other options that affect this mode include:

- `--build-target`

  Specify a specific target to build.If left out the `all` target is built.

- `--build-nocmake`

  Run the build without running cmake first.Skip the cmake step.
- `--build-run-dir`

  Specify directory to run programs from.Directory where programs will be after it has been compiled.
- `--build-two-config`

  Run CMake twice.
- `--build-exe-dir`

  Specify the directory for the executable.
- `--build-generator`

  Specify the generator to use. See the `cmake-generators(7)` manual.
- `--build-generator-platform`

  Specify the generator-specific platform.
- `--build-generator-toolset`

  Specify the generator-specific toolset.
- `--build-project`

  Specify the name of the project to build.
- `--build-makeprogram`

  Override the make program chosen by CTest with a given one.
- `--build-noclean`

  Skip the make clean step.
- `--build-config-sample`

  A sample executable to use to determine the configuration that should be used. e.g. Debug/Release/etc.
- `--build-options`

  Additional options for configuring the build (i.e. for CMake, not for the build tool). Note that if this is specified, the `--build-options` keyword and its arguments must be the last option given on the command line, with the possible exception of `--test-command`.
- `--test-command`

  The command to run as the test step with the `--build-and-test` option. All arguments following this keyword will be assumed to be part of the test command line, so it must be the last option given.
- `--test-timeout`

  The time limit in seconds

Dashboard Client

CTest can operate as a client for the CDash software quality dashboard application. As a dashboard client, CTest performs a sequence of steps to configure, build, and test software, and then submits the results to a CDashserver. The command-line signature used to submit to CDash is:

```
ctest (-D <dashboard> | -M <model> -T <action> | -S <script> | -SP <script>)
      [-- <dashboard-options>...]
```

Options for Dashboard Client include:

- `--track <track>`

  Specify the track to submit dashboard toSubmit dashboard to specified track instead of default one. By default, the dashboard is submitted to Nightly, Experimental, or Continuous track, but by specifying this option, the track can be arbitrary.

- `-A <file>, --add-notes <file>`

  Add a notes file with submission.This option tells CTest to include a notes file when submitting dashboard.

- `--tomorrow-tag`

  Nightly or experimental starts with next day tag.This is useful if the build will not finish in one day.

- `--extra-submit <file>[;<file>]`

  Submit extra files to the dashboard.This option will submit extra files to the dashboard.

- `--http1.0`

  Submit using HTTP 1.0.This option will force CTest to use HTTP 1.0 to submit files to the dashboard, instead of HTTP 1.1.

- `--no-compress-output`

  Do not compress test output when submitting.This flag will turn off automatic compression of test output. Use this to maintain compatibility with an older version of CDash which doesn't support compressed test output.

Dashboard Client Steps

CTest defines an ordered list of testing steps of which some or all may be run as a dashboard client:

- `Start`

  Start a new dashboard submission to be composed of results recorded by the following steps. See the CTest Start Step section below.

- `Update`

  Update the source tree from its version control repository. Record the old and new versions and the list of updated source files. See the CTest Update Step section below.

- `Configure`

  Configure the software by running a command in the build tree. Record the configuration output log. See the CTest Configure Step section below.

- `Build`

  Build the software by running a command in the build tree. Record the build output log and detect warnings and errors. See the CTest Build Step section below.

- `Test`

Test the software by loading a `CTestTestfile.cmake` from the build tree and executing the defined tests. Record the output and result of each test. See the CTest Test Step section below.

- `Coverage`

  Compute coverage of the source code by running a coverage analysis tool and recording its output. See the CTest Coverage Step section below.

- `MemCheck`

  Run the software test suite through a memory check tool. Record the test output, results, and issues reported by the tool. See the CTest MemCheck Step section below.

- `Submit`

  Submit results recorded from other testing steps to the software quality dashboard server. See the CTest Submit Step section below.

Dashboard Client Modes

CTest defines three modes of operation as a dashboard client:

- `Nightly`

  This mode is intended to be invoked once per day, typically at night. It enables the `Start`, `Update`, `Configure`, `Build`, `Test`, `Coverage`, and `Submit` steps by default. Selected steps run even if the `Update` step reports no changes to the source tree.

- `Continuous`

  This mode is intended to be invoked repeatedly throughout the day. It enables the `Start`, `Update`, `Configure`, `Build`, `Test`, `Coverage`, and `Submit` steps by default, but exits after the `Update` step if it reports no changes to the source tree.

- `Experimental`

  This mode is intended to be invoked by a developer to test local changes. It enables the `Start`, `Configure`, `Build`, `Test`, `Coverage`, and `Submit` steps by default.

Dashboard Client via CTest Command-Line

CTest can perform testing on an already-generated build tree. Run the `ctest` command with the current working directory set to the build tree and use one of these signatures:

```
ctest -D <mode>[<step>]
ctest -M <mode> [ -T <step> ]...
```

The `<mode>` must be one of the above Dashboard Client Modes, and each `<step>` must be one of the above Dashboard Client Steps.

CTest reads the Dashboard Client Configuration settings from a file in the build tree called either `CTestConfiguration.ini` or `DartConfiguration.tcl` (the names are historical). The format of the file is:

```
# Lines starting in '#' are comments.
# Other non-blank lines are key-value pairs.
<setting>: <value>
```

where `<setting>` is the setting name and `<value>` is the setting value.

In build trees generated by CMake, this configuration file is generated by the `CTest` module if included by the project. The module uses variables to obtain a value for each setting as documented with the settings below.

Dashboard Client via CTest Script

CTest can perform testing driven by a `cmake-language(7)` script that creates and maintains the source and build tree as well as performing the testing steps. Run the `ctest` command with the current working directory set outside of any build tree and use one of these signatures:

```
ctest -S <script>
ctest -SP <script>
```

The `<script>` file must call CTest Commands commands to run testing steps explicitly as documented below. The commands obtain Dashboard Client Configuration settings from their arguments or from variables set in the script.

Dashboard Client Configuration

The Dashboard Client Steps may be configured by named settings as documented in the following sections.

CTest Start Step

Start a new dashboard submission to be composed of results recorded by the following steps.

In a CTest Script, the `ctest_start()` command runs this step. Arguments to the command may specify some of the step settings. The command first runs the command-line specified by the `CTEST_CHECKOUT_COMMAND` variable, if set, to initialize the source directory.

Configuration settings include:

- `BuildDirectory`

  The full path to the project build tree.CTest Script variable: `CTEST_BINARY_DIRECTORY` `CTest` module variable: `PROJECT_BINARY_DIR`

- `SourceDirectory`

  The full path to the project source tree.CTest Script variable: `CTEST_SOURCE_DIRECTORY` `CTest` module variable: `PROJECT_SOURCE_DIR`

CTest Update Step

In a CTest Script, the `ctest_update()` command runs this step. Arguments to the command may specify some of the step settings.

Configuration settings to specify the version control tool include:

- `BZRCommand`

  `bzr` command-line tool to use if source tree is managed by Bazaar.CTest Script variable: `CTEST_BZR_COMMAND` `CTest` module variable: none

- `BZRUpdateOptions`

  Command-line options to the `BZRCommand` when updating the source.CTest Script variable: `CTEST_BZR_UPDATE_OPTIONS` `CTest` module variable: none

- `CVSCommand`

  `cvs` command-line tool to use if source tree is managed by CVS.CTest Script variable: `CTEST_CVS_COMMAND` `CTest` module variable: `CVSCOMMAND`

- `CVSUpdateOptions`

  Command-line options to the `CVSCommand` when updating the source.CTest Script variable: `CTEST_CVS_UPDATE_OPTIONS` `CTest` module variable: `CVS_UPDATE_OPTIONS`

- `GITCommand`

  `git` command-line tool to use if source tree is managed by Git.CTest Script variable: `CTEST_GIT_COMMAND` `CTest` module variable: `GITCOMMAND` The source tree is updated by `git fetch` followed by `git reset --hard` to the `FETCH_HEAD`. The result is the same as `git pull` except that any local moficiations are overwritten. Use `GITUpdateCustom` to specify a different approach.

- `GITInitSubmodules`

  If set, CTest will update the repository's submodules before updating.CTest Script variable: `CTEST_GIT_INIT_SUBMODULES` `CTest` module variable: `CTEST_GIT_INIT_SUBMODULES`

- `GITUpdateCustom`

  Specify a custom command line (as a semicolon-separated list) to run in the source tree (Git work tree) to update it instead of running the `GITCommand`.CTest Script variable: `CTEST_GIT_UPDATE_CUSTOM` `CTest` module variable: `CTEST_GIT_UPDATE_CUSTOM`

- `GITUpdateOptions`

  Command-line options to the `GITCommand` when updating the source.CTest Script variable: `CTEST_GIT_UPDATE_OPTIONS` `CTest` module variable: `GIT_UPDATE_OPTIONS`

- `HGCommand`

  `hg` command-line tool to use if source tree is managed by Mercurial.CTest Script variable: `CTEST_HG_COMMAND` `CTest` module variable: none

- `HGUpdateOptions`

  Command-line options to the `HGCommand` when updating the source.CTest Script variable: `CTEST_HG_UPDATE_OPTIONS` `CTest` module variable: none

- `P4Client`

  Value of the `-c` option to the `P4Command`.CTest Script variable: `CTEST_P4_CLIENT` `CTest` module variable: `CTEST_P4_CLIENT`

- `P4Command`

  `p4` command-line tool to use if source tree is managed by Perforce.CTest Script variable: `CTEST_P4_COMMAND` `CTest` module variable: `P4COMMAND`

- `P4Options`

  Command-line options to the `P4Command` for all invocations.CTest Script variable: `CTEST_P4_OPTIONS` `CTest` module variable: `CTEST_P4_OPTIONS`

- `P4UpdateCustom`

  Specify a custom command line (as a semicolon-separated list) to run in the source tree (Perforce tree) to update it instead of running the `P4Command`.CTest Script variable: none `CTest` module variable: `CTEST_P4_UPDATE_CUSTOM`

- `P4UpdateOptions`

  Command-line options to the `P4Command` when updating the source.CTest Script variable: `CTEST_P4_UPDATE_OPTIONS` `CTest` module variable: `CTEST_P4_UPDATE_OPTIONS`

- `SVNCommand`

  `svn` command-line tool to use if source tree is managed by Subversion.CTest Script variable: `CTEST_SVN_COMMAND` `CTest` module variable: `SVNCOMMAND`

- `SVNOptions`

  Command-line options to the `SVNCommand` for all invocations.CTest Script variable: `CTEST_SVN_OPTIONS` `CTest` module variable: `CTEST_SVN_OPTIONS`

- `SVNUpdateOptions`

  Command-line options to the `SVNCommand` when updating the source.CTest Script variable: `CTEST_SVN_UPDATE_OPTIONS` `CTest` module variable: `SVN_UPDATE_OPTIONS`

- `UpdateCommand`

  Specify the version-control command-line tool to use without detecting the VCS that manages the source tree.CTest Script variable: `CTEST_UPDATE_COMMAND` `CTest` module variable: `<VCS>COMMAND` when `UPDATE_TYPE` is `<vcs>`, else `UPDATE_COMMAND`

- `UpdateOptions`

  Command-line options to the `UpdateCommand`.CTest Script variable: `CTEST_UPDATE_OPTIONS` `CTest` module variable: `<VCS>_UPDATE_OPTIONS` when `UPDATE_TYPE` is `<vcs>`, else `UPDATE_OPTIONS`

- `UpdateType`

  Specify the version-control system that manages the source tree if it cannot be detected automatically. The value may be `bzr`, `cvs`, `git`, `hg`, `p4`, or `svn`.CTest Script variable: none, detected from source tree `CTest` module variable: `UPDATE_TYPE` if set, else `CTEST_UPDATE_TYPE`

- `UpdateVersionOnly`

  Specify that you want the version control update command to only discover the current version that is checked out, and not to update to a different version.CTest Script variable: `CTEST_UPDATE_VERSION_ONLY`

Additional configuration settings include:

- `NightlyStartTime`

  In the `Nightly` dashboard mode, specify the “nightly start time”. With centralized version control systems (`cvs` and `svn`), the `Update` step checks out the version of the software as of this time so that multiple clients choose a common version to test. This is not well-defined in distributed version-control systems so the setting is ignored.CTest Script variable: `CTEST_NIGHTLY_START_TIME` `CTest` module

variable: `NIGHTLY_START_TIME` if set, else `CTEST_NIGHTLY_START_TIME`

CTest Configure Step

In a CTest Script, the `ctest_configure()` command runs this step. Arguments to the command may specify some of the step settings.

Configuration settings include:

- `ConfigureCommand`

  Command-line to launch the software configuration process. It will be executed in the location specified by the `BuildDirectory` setting.CTest Script variable: `CTEST_CONFIGURE_COMMAND` `CTest` module variable: `CMAKE_COMMAND` followed by `PROJECT_SOURCE_DIR`

- `LabelsForSubprojects`

  Specify a semicolon-separated list of labels that will be treated as subprojects. This mapping will be passed on to CDash when configure, test or build results are submitted.CTest Script variable: `CTEST_LABELS_FOR_SUBPROJECTS` `CTest` module variable: `CTEST_LABELS_FOR_SUBPROJECTS` See Label and Subproject Summary.

CTest Build Step

In a CTest Script, the `ctest_build()` command runs this step. Arguments to the command may specify some of the step settings.

Configuration settings include:

- `DefaultCTestConfigurationType`

  When the build system to be launched allows build-time selection of the configuration (e.g. `Debug`, `Release`), this specifies the default configuration to be built when no `-C` option is given to the `ctest` command. The value will be substituted into the value of `MakeCommand` to replace the literal string `${CTEST_CONFIGURATION_TYPE}` if it appears.CTest Script variable: `CTEST_CONFIGURATION_TYPE` `CTest` module variable: `DEFAULT_CTEST_CONFIGURATION_TYPE`, initialized by the `CMAKE_CONFIG_TYPE` environment variable

- `LabelsForSubprojects`

  Specify a semicolon-separated list of labels that will be treated as subprojects. This mapping will be passed on to CDash when configure, test or build results are submitted.CTest Script variable: `CTEST_LABELS_FOR_SUBPROJECTS` `CTest` module variable: `CTEST_LABELS_FOR_SUBPROJECTS` See Label and Subproject Summary.

- `MakeCommand`

  Command-line to launch the software build process. It will be executed in the location specified by the `BuildDirectory` setting.CTest Script variable: `CTEST_BUILD_COMMAND` `CTest` module variable: `MAKECOMMAND`, initialized by the `build_command()` command

- `UseLaunchers`

For build trees generated by CMake using one of the Makefile Generators or the `Ninja` generator, specify whether the `CTEST_USE_LAUNCHERS` feature is enabled by the `CTestUseLaunchers` module (also included by the `CTest` module). When enabled, the generated build system wraps each invocation of the compiler, linker, or custom command line with a "launcher" that communicates with CTest via environment variables and files to report granular build warning and error information. Otherwise, CTest must "scrape" the build output log for diagnostics.CTest Script variable: `CTEST_USE_LAUNCHERS` `CTest` module variable: `CTEST_USE_LAUNCHERS`

CTest Test Step

In a CTest Script, the `ctest_test()` command runs this step. Arguments to the command may specify some of the step settings.

Configuration settings include:

- `LabelsForSubprojects`

  Specify a semicolon-separated list of labels that will be treated as subprojects. This mapping will be passed on to CDash when configure, test or build results are submitted.CTest Script variable: `CTEST_LABELS_FOR_SUBPROJECTS` `CTest` module variable: `CTEST_LABELS_FOR_SUBPROJECTS` See Label and Subproject Summary.

- `TestLoad`

  While running tests in parallel (e.g. with `-j`), try not to start tests when they may cause the CPU load to pass above a given threshold.CTest Script variable: `CTEST_TEST_LOAD` `CTest` module variable: `CTEST_TEST_LOAD`

- `TimeOut`

  The default timeout for each test if not specified by the `TIMEOUT` test property.CTest Script variable: `CTEST_TEST_TIMEOUT` `CTest` module variable: `DART_TESTING_TIMEOUT`

CTest Coverage Step

In a CTest Script, the `ctest_coverage()` command runs this step. Arguments to the command may specify some of the step settings.

Configuration settings include:

- `CoverageCommand`

  Command-line tool to perform software coverage analysis. It will be executed in the location specified by the `BuildDirectory` setting.CTest Script variable: `CTEST_COVERAGE_COMMAND` `CTest` module variable: `COVERAGE_COMMAND`

- `CoverageExtraFlags`

  Specify command-line options to the `CoverageCommand` tool.CTest Script variable: `CTEST_COVERAGE_EXTRA_FLAGS` `CTest` module variable: `COVERAGE_EXTRA_FLAGS` These options are the first arguments passed to `CoverageCommand`.

CTest MemCheck Step

In a CTest Script, the `ctest_memcheck()` command runs this step. Arguments to the command may specify some of the step settings.

Configuration settings include:

- `MemoryCheckCommand`

  Command-line tool to perform dynamic analysis. Test command lines will be launched through this tool.CTest Script variable: `CTEST_MEMORYCHECK_COMMAND` `CTest` module variable: `MEMORYCHECK_COMMAND`

- `MemoryCheckCommandOptions`

  Specify command-line options to the `MemoryCheckCommand` tool. They will be placed prior to the test command line.CTest Script variable: `CTEST_MEMORYCHECK_COMMAND_OPTIONS` `CTest` module variable: `MEMORYCHECK_COMMAND_OPTIONS`

- `MemoryCheckType`

  Specify the type of memory checking to perform.CTest Script variable: `CTEST_MEMORYCHECK_TYPE` `CTest` module variable: `MEMORYCHECK_TYPE`

- `MemoryCheckSanitizerOptions`

  Specify options to sanitizers when running with a sanitize-enabled build.CTest Script variable: `CTEST_MEMORYCHECK_SANITIZER_OPTIONS` `CTest` module variable: `MEMORYCHECK_SANITIZER_OPTIONS`

- `MemoryCheckSuppressionFile`

  Specify a file containing suppression rules for the `MemoryCheckCommand` tool. It will be passed with options appropriate to the tool.CTest Script variable: `CTEST_MEMORYCHECK_SUPPRESSIONS_FILE` `CTest` module variable: `MEMORYCHECK_SUPPRESSIONS_FILE`

Additional configuration settings include:

- `BoundsCheckerCommand`

  Specify a `MemoryCheckCommand` that is known to be command-line compatible with Bounds Checker.CTest Script variable: none `CTest` module variable: none

- `PurifyCommand`

  Specify a `MemoryCheckCommand` that is known to be command-line compatible with Purify.CTest Script variable: none `CTest` module variable: `PURIFYCOMMAND`

- `ValgrindCommand`

  Specify a `MemoryCheckCommand` that is known to be command-line compatible with Valgrind.CTest Script variable: none `CTest` module variable: `VALGRIND_COMMAND`

- `ValgrindCommandOptions`

  Specify command-line options to the `ValgrindCommand` tool. They will be placed prior to the test command line.CTest Script variable: none `CTest` module variable: `VALGRIND_COMMAND_OPTIONS`

CTest Submit Step

In a CTest Script, the `ctest_submit()` command runs this step. Arguments to the command may specify some of the step settings.

Configuration settings include:

- `BuildName`

  Describe the dashboard client platform with a short string. (Operating system, compiler, etc.)CTest Script variable: `CTEST_BUILD_NAME` `CTest` module variable: `BUILDNAME`

- `CDashVersion`

  Specify the version of CDash on the server.CTest Script variable: none, detected from server `CTest` module variable: `CTEST_CDASH_VERSION`

- `CTestSubmitRetryCount`

  Specify a number of attempts to retry submission on network failure.CTest Script variable: none, use the `ctest_submit()` `RETRY_COUNT` option. `CTest` module variable: `CTEST_SUBMIT_RETRY_COUNT`

- `CTestSubmitRetryDelay`

  Specify a delay before retrying submission on network failure.CTest Script variable: none, use the `ctest_submit()` `RETRY_DELAY` option. `CTest` module variable: `CTEST_SUBMIT_RETRY_DELAY`

- `CurlOptions`

  Specify a semicolon-separated list of options to control the Curl library that CTest uses internally to connect to the server. Possible options are `CURLOPT_SSL_VERIFYPEER_OFF` and `CURLOPT_SSL_VERIFYHOST_OFF`.CTest Script variable: `CTEST_CURL_OPTIONS` `CTest` module variable: `CTEST_CURL_OPTIONS`

- `DropLocation`

  The path on the dashboard server to send the submission.CTest Script variable: `CTEST_DROP_LOCATION` `CTest` module variable: `DROP_LOCATION` if set, else `CTEST_DROP_LOCATION`

- `DropMethod`

  Specify the method by which results should be submitted to the dashboard server. The value may be `cp`, `ftp`, `http`, `https`, `scp`, or `xmlrpc` (if CMake was built with support for it).CTest Script variable: `CTEST_DROP_METHOD` `CTest` module variable: `DROP_METHOD` if set, else `CTEST_DROP_METHOD`

- `DropSite`

  The dashboard server name (for `ftp`, `http`, and `https`, `scp`, and `xmlrpc`).CTest Script variable: `CTEST_DROP_SITE` `CTest` module variable: `DROP_SITE` if set, else `CTEST_DROP_SITE`

- `DropSitePassword`

  The dashboard server login password, if any (for `ftp`, `http`, and `https`).CTest Script variable: `CTEST_DROP_SITE_PASSWORD` `CTest` module variable: `DROP_SITE_PASSWORD` if set, else `CTEST_DROP_SITE_PASWORD`

- `DropSiteUser`

  The dashboard server login user name, if any (for `ftp`, `http`, and `https`).CTest Script variable: `CTEST_DROP_SITE_USER` `CTest` module variable: `DROP_SITE_USER` if set, else `CTEST_DROP_SITE_USER`

- `IsCDash`

  Specify whether the dashboard server is CDash or an older dashboard server implementation requiring `TriggerSite`.CTest Script variable: `CTEST_DROP_SITE_CDASH` `CTest` module variable: `CTEST_DROP_SITE_CDASH`

- `ScpCommand`

  `scp` command-line tool to use when `DropMethod` is `scp`.CTest Script variable: `CTEST_SCP_COMMAND` `CTest` module variable: `SCPCOMMAND`

- `Site`

  Describe the dashboard client host site with a short string. (Hostname, domain, etc.)CTest Script variable: `CTEST_SITE` `CTest` module variable: `SITE`, initialized by the `site_name()` command

- `TriggerSite`

  Legacy option to support older dashboard server implementations. Not used when `IsCDash` is true.CTest Script variable: `CTEST_TRIGGER_SITE` `CTest` module variable: `TRIGGER_SITE` if set, else `CTEST_TRIGGER_SITE`

See Also

The following resources are available to get help using CMake:

- Home Page

  https://cmake.orgThe primary starting point for learning about CMake.

- Online Documentation and Community Resources

  https://cmake.org/documentationLinks to available documentation and community resources may be found on this web page.

- Mailing List

  https://cmake.org/mailing-listsFor help and discussion about using cmake, a mailing list is provided at cmake@cmake.org. The list is member-post-only but one may sign up on the CMake web page. Please first read the full documentation at https://cmake.org before posting questions to the list.

==========

cpack(1)

Synopsis

```
cpack [<options>]
```

Description

The `cpack` executable is the CMake packaging program. CMake projects use `install()` commands to define the contents of packages which can be generated in various formats by this tool. The `CPack` module greatly simplifies the creation of the input file used by `cpack`, allowing most aspects of the packaging configuration to be controlled directly from the CMake project's own `CMakeLists.txt` files.

Options

- `-G <generators>`

  `<generators>` is a semicolon-separated list of generator names. `cpack` will iterate through this list and produce package(s) in that generator's format according to the details provided in the `CPackConfig.cmake` configuration file. A generator is responsible for generating the required inputs for a particular package system and invoking that system's package creation tools. All supported generators are specified in the `Generators` section of the manual and the `--help` option lists the generators supported for the target platform.If this option is not given, the `CPACK_GENERATOR` variable determines the default set of generators that will be used.

- `-C <Configuration>`

  Specify the project configuration to be packaged (e.g. `Debug`, `Release`, etc.). When the CMake project uses a multi-configuration generator such as Xcode or Visual Studio, this option is needed to tell `cpack` which built executables to include in the package.

- `-D <var>=<value>`

  Set a CPack variable. This will override any value set for `<var>` in the input file read by `cpack`.

- `--config <configFile>`

  Specify the configuration file read by `cpack` to provide the packaging details. By default, `CPackConfig.cmake` in the current directory will be used.

- `--verbose,-V`

  Run `cpack` with verbose output. This can be used to show more details from the package generation tools and is suitable for project developers.

- `--debug`

  Run `cpack` with debug output. This option is intended mainly for the developers of `cpack` itself and is not normally needed by project developers.

- `--trace`

  Put the underlying cmake scripts in trace mode.

- `--trace-expand`

  Put the underlying cmake scripts in expanded trace mode.

- `-P <packageName>`

  Override/define the value of the `CPACK_PACKAGE_NAME` variable used for packaging. Any value set for this variable in the `CPackConfig.cmake` file will then be ignored.

- `-R <packageVersion>`

Override/define the value of the `CPACK_PACKAGE_VERSION` variable used for packaging. It will override a value set in the `CPackConfig.cmake` file or one automatically computed from `CPACK_PACKAGE_VERSION_MAJOR`, `CPACK_PACKAGE_VERSION_MINOR` and `CPACK_PACKAGE_VERSION_PATCH`.

- `-B <packageDirectory>`

  Override/define `CPACK_PACKAGE_DIRECTORY`, which controls the directory where CPack will perform its packaging work. The resultant package(s) will be created at this location by default and a `_CPack_Packages` subdirectory will also be created below this directory to use as a working area during package creation.

- `--vendor <vendorName>`

  Override/define `CPACK_PACKAGE_VENDOR`.

- `--help,-help,-usage,-h,-H,/?`

  Print usage information and exit.Usage describes the basic command line interface and its options.

- `--version,-version,/V [<f>]`

  Show program name/version banner and exit.If a file is specified, the version is written into it. The help is printed to a named ile if given.

- `--help-full [<f>]`

  Print all help manuals and exit.All manuals are printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-manual <man> [<f>]`

  Print one help manual and exit.The specified manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-manual-list [<f>]`

  List help manuals available and exit.The list contains all manuals for which help may be obtained by using the `--help-manual` option followed by a manual name. The help is printed to a named ile if given.

- `--help-command <cmd> [<f>]`

  Print help for one command and exit.The `cmake-commands(7)` manual entry for `<cmd>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-command-list [<f>]`

  List commands with help available and exit.The list contains all commands for which help may be obtained by using the `--help-command` option followed by a command name. The help is printed to a named ile if given.

- `--help-commands [<f>]`

  Print cmake-commands manual and exit.The `cmake-commands(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-module <mod> [<f>]`

  Print help for one module and exit.The `cmake-modules(7)` manual entry for `<mod>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-module-list [<f>]`

List modules with help available and exit.The list contains all modules for which help may be obtained by using the `--help-module` option followed by a module name. The help is printed to a named ile if given.

- `--help-modules [<f>]`

  Print cmake-modules manual and exit.The `cmake-modules(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-policy <cmp> [<f>]`

  Print help for one policy and exit.The `cmake-policies(7)` manual entry for `<cmp>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-policy-list [<f>]`

  List policies with help available and exit.The list contains all policies for which help may be obtained by using the `--help-policy` option followed by a policy name. The help is printed to a named ile if given.

- `--help-policies [<f>]`

  Print cmake-policies manual and exit.The `cmake-policies(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-property <prop> [<f>]`

  Print help for one property and exit.The `cmake-properties(7)` manual entries for `<prop>` are printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-property-list [<f>]`

  List properties with help available and exit.The list contains all properties for which help may be obtained by using the `--help-property` option followed by a property name. The help is printed to a named ile if given.

- `--help-properties [<f>]`

  Print cmake-properties manual and exit.The `cmake-properties(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-variable <var> [<f>]`

  Print help for one variable and exit.The `cmake-variables(7)` manual entry for `<var>` is printed in a human-readable text format. The help is printed to a named ile if given.

- `--help-variable-list [<f>]`

  List variables with help available and exit.The list contains all variables for which help may be obtained by using the `--help-variable` option followed by a variable name. The help is printed to a named ile if given.

- `--help-variables [<f>]`

  Print cmake-variables manual and exit.The `cmake-variables(7)` manual is printed in a human-readable text format. The help is printed to a named ile if given.

See Also

The following resources are available to get help using CMake:

- Home Page

  https://cmake.orgThe primary starting point for learning about CMake.

- Online Documentation and Community Resources

  https://cmake.org/documentationLinks to available documentation and community resources may be found on this web page.

- Mailing List

  https://cmake.org/mailing-listsFor help and discussion about using cmake, a mailing list is provided at cmake@cmake.org. The list is member-post-only but one may sign up on the CMake web page. Please first read the full documentation at https://cmake.org before posting questions to the list.

==========

cmake-developer(7)

Contents

Introduction

This manual is intended for reference by developers modifying the CMake source tree itself, and by those authoring externally-maintained modules.

Adding Compile Features

CMake reports an error if a compiler whose features are known does not report support for a particular requested feature. A compiler is considered to have known features if it reports support for at least one feature.

When adding a new compile feature to CMake, it is therefore necessary to list support for the feature for all CompilerIds which already have one or more feature supported, if the new feature is available for any version of the compiler.

When adding the first supported feature to a particular CompilerId, it is necessary to list support for all features known to cmake (See `CMAKE_C_COMPILE_FEATURES` and `CMAKE_CXX_COMPILE_FEATURES` as appropriate), where available for the compiler. Ensure that the `CMAKE_<LANG>_STANDARD_DEFAULT` is set to the computed internal variable `CMAKE_<LANG>_STANDARD_COMPUTED_DEFAULT` for compiler versions which should be supported.

It is sensible to record the features for the most recent version of a particular CompilerId first, and then work backwards. It is sensible to try to create a continuous range of versions of feature releases of the compiler. Gaps in the range indicate incorrect features recorded for intermediate releases.

Generally, features are made available for a particular version if the compiler vendor documents availability of the feature with that version. Note that sometimes partially implemented features appear to be functional in previous releases (such as `cxx_constexpr` in GNU 4.6, though availability is documented in GNU 4.7), and sometimes compiler vendors document availability of features, though supporting infrastructure is not available (such as `__has_feature(cxx_generic_lambdas)` indicating non-availability in Clang 3.4, though it is documented as available, and fixed in Clang 3.5). Similar cases for other compilers and versions need to be investigated when extending CMake to support them.

When a vendor releases a new version of a known compiler which supports a previously unsupported feature, and there are already known features for that compiler, the feature should be listed as supported in CMake for that version of the compiler as soon as reasonably possible.

Standard-specific/compiler-specific variables such `CMAKE_CXX98_COMPILE_FEATURES` are deliberately not documented. They only exist for the compiler-specific implementation of adding the `-std` compile flag for compilers which need that.

Help

The `Help` directory contains CMake help manual source files. They are written using the reStructuredText markup syntax and processed by Sphinx to generate the CMake help manuals.

Markup Constructs

In addition to using Sphinx to generate the CMake help manuals, we also use a C++-implemented document processor to print documents for the `--help-*` command-line help options. It supports a subset of reStructuredText markup. When authoring or modifying documents, please verify that the command-line help looks good in addition to the Sphinx-generated html and man pages.

The command-line help processor supports the following constructs defined by reStructuredText, Sphinx, and a CMake extension to Sphinx.

- CMake Domain directives

  Directives defined in the CMake Domain for defining CMake documentation objects are printed in command-line help output as if the lines were normal paragraph text with interpretation.

- CMake Domain interpreted text roles

  Interpreted text roles defined in the CMake Domain for cross-referencing CMake documentation objects are replaced by their link text in command-line help output. Other roles are printed literally and not processed.

- `code-block` directive

  Add a literal code block without interpretation. The command-line help processor prints the block content without the leading directive line and with common indentation replaced by one space.

- `include` directive

  Include another document source file. The command-line help processor prints the included document inline with the referencing document.

- literal block after `::`

  A paragraph ending in `::` followed by a blank line treats the following indented block as literal text without interpretation. The command-line help processor prints the `::` literally and prints the block content with common indentation replaced by one space.

- `note` directive

  Call out a side note. The command-line help processor prints the block content as if the lines were normal paragraph text with interpretation.

- `parsed-literal` directive

  Add a literal block with markup interpretation. The command-line help processor prints the block content without the leading directive line and with common indentation replaced by one space.

- `productionlist` directive

  Render context-free grammar productions. The command-line help processor prints the block content as if the lines were normal paragraph text with interpretation.

- `replace` directive

  Define a `|substitution|` replacement. The command-line help processor requires a substitution replacement to be defined before it is referenced.

- `|substitution|` reference

  Reference a substitution replacement previously defined by the `replace` directive. The command-line help processor performs the substitution and replaces all newlines in the replacement text with spaces.

- `toctree` directive

  Include other document sources in the Table-of-Contents document tree. The command-line help processor prints the referenced documents inline as part of the referencing document.

Inline markup constructs not listed above are printed literally in the command-line help output. We prefer to use inline markup constructs that look correct in source form, so avoid use of -escapes in favor of inline literals when possible.

Explicit markup blocks not matching directives listed above are removed from command-line help output. Do not use them, except for plain `..` comments that are removed by Sphinx too.

Note that nested indentation of blocks is not recognized by the command-line help processor. Therefore:

- Explicit markup blocks are recognized only when not indented inside other blocks.
- Literal blocks after paragraphs ending in `::` but not at the top indentation level may consume all indented lines following them.

Try to avoid these cases in practice.

CMake Domain

CMake adds a Sphinx Domain called `cmake`, also called the "CMake Domain". It defines several "object" types for CMake documentation:

- `command`

  A CMake language command.

- `generator`

  A CMake native build system generator. See the `cmake(1)` command-line tool's `-G` option.

- `manual`

  A CMake manual page, like this `cmake-developer(7)` manual.

- `module`

  A CMake module. See the `cmake-modules(7)` manual and the `include()` command.

- `policy`

  A CMake policy. See the `cmake-policies(7)` manual and the `cmake_policy()` command.

- `prop_cache, prop_dir, prop_gbl, prop_sf, prop_inst, prop_test, prop_tgt`

  A CMake cache, directory, global, source file, installed file, test, or target property, respectively. See the `cmake-properties(7)` manual and the `set_property()` command.

- `variable`

  A CMake language variable. See the `cmake-variables(7)` manual and the `set()` command.

Documentation objects in the CMake Domain come from two sources. First, the CMake extension to Sphinx transforms every document named with the form `Help/<type>/<file-name>.rst` to a domain object with type `<type>`. The object name is extracted from the document title, which is expected to be of the form:

```
<object-name>
-------------
```

and to appear at or near the top of the `.rst` file before any other lines starting in a letter, digit, or `<`. If no such title appears literally in the `.rst` file, the object name is the `<file-name>`. If a title does appear, it is expected that `<file-name>` is equal to `<object-name>` with any `<` and `>` characters removed.

Second, the CMake Domain provides directives to define objects inside other documents:

```
.. command:: <command-name>

 This indented block documents <command-name>.

.. variable:: <variable-name>

 This indented block documents <variable-name>.
```

Object types for which no directive is available must be defined using the first approach above.

Cross-References

Sphinx uses reStructuredText interpreted text roles to provide cross-reference syntax. The CMake Domainprovides for each domain object type a role of the same name to cross-reference it. CMake Domain roles are inline markup of the forms:

```
:type:`name`
:type:`text <name>`
```

where `type` is the domain object type and `name` is the domain object name. In the first form the link text will be `name` (or `name()` if the type is `command`) and in the second form the link text will be the explicit `text`. For example, the code:

```
* The :command:`list` command.
* The :command:`list(APPEND)` sub-command.
* The :command:`list() command <list>`.
* The :command:`list(APPEND) sub-command <list>`.
* The :variable:`CMAKE_VERSION` variable.
* The :prop_tgt:`OUTPUT_NAME_<CONFIG>` target property.
```

produces:

- The `list()` command.
- The `list(APPEND)` sub-command.
- The `list() command`.
- The `list(APPEND) sub-command`.
- The `CMAKE_VERSION` variable.
- The `OUTPUT_NAME_` target property.

Note that CMake Domain roles differ from Sphinx and reStructuredText convention in that the form `a<b>`, without a space preceding `<`, is interpreted as a name instead of link text with an explicit target. This is necessary because we use `<placeholders>` frequently in object names like `OUTPUT_NAME_<CONFIG>`. The form `a <b>`, with a space preceding `<`, is still interpreted as a link text with an explicit target.

Style

Style: Section Headers

When marking section titles, make the section decoration line as long as the title text. Use only a line below the title, not above. For example:

```
Title Text
----------
```

Capitalize the first letter of each non-minor word in the title.

The section header underline character hierarchy is

- `#`: Manual group (part) in the master document
- `*`: Manual (chapter) title
- `=`: Section within a manual
- `-`: Subsection or CMake Domain object document title
- `^`: Subsubsection or CMake Domain object document section
- `"`: Paragraph or CMake Domain object document subsection

Style: Whitespace

Use two spaces for indentation. Use two spaces between sentences in prose.

Style: Line Length

Prefer to restrict the width of lines to 75-80 columns. This is not a hard restriction, but writing new paragraphs wrapped at 75 columns allows space for adding minor content without significant re-wrapping of content.

Style: Prose

Use American English spellings in prose.

Style: Starting Literal Blocks

Prefer to mark the start of literal blocks with `::` at the end of the preceding paragraph. In cases where the following block gets a `code-block` marker, put a single `:` at the end of the preceding paragraph.

Style: CMake Command Signatures

Command signatures should be marked up as plain literal blocks, not as cmake `code-blocks`.

Signatures are separated from preceding content by a section header. That is, use:

```
... preceding paragraph.

Normal Libraries
^^^^^^^^^^^^^^^^

::

  add_library(<lib> ...)

This signature is used for ...
```

Signatures of commands should wrap optional parts with square brackets, and should mark list of optional arguments with an ellipsis (`...`). Elements of the signature which are specified by the user should be specified with angle brackets, and may be referred to in prose using `inline-literal` syntax.

Style: Boolean Constants

Use "`OFF`" and "`ON`" for boolean values which can be modified by the user, such as `POSITION_INDEPENDENT_CODE`. Such properties may be "enabled" and "disabled". Use "`True`" and "`False`" for inherent values which can't be modified after being set, such as the `IMPORTED` property of a build target.

Style: Inline Literals

Mark up references to keywords in signatures, file names, and other technical terms with `inline-literal` syntax, for example:

```
If ``WIN32`` is used with :command:`add_executable`, the
:prop_tgt:`WIN32_EXECUTABLE` target property is enabled. That command
creates the file ``<name>.exe`` on Windows.
```

Style: Cross-References

Mark up linkable references as links, including repeats. An alternative, which is used by wikipedia (http://en.wikipedia.org/wiki/WP:REPEATLINK), is to link to a reference only once per article. That style is not used in CMake documentation.

Style: Referencing CMake Concepts

If referring to a concept which corresponds to a property, and that concept is described in a high-level manual, prefer to link to the manual section instead of the property. For example:

```
This command creates an :ref:`Imported Target <Imported Targets>`.
```

instead of:

```
This command creates an :prop_tgt:`IMPORTED` target.
```

The latter should be used only when referring specifically to the property.

References to manual sections are not automatically created by creating a section, but code such as:

```
.. _`Imported Targets`:
```

creates a suitable anchor. Use an anchor name which matches the name of the corresponding section. Refer to the anchor using a cross-reference with specified text.

Imported Targets need the `IMPORTED` term marked up with care in particular because the term may refer to a command keyword (`IMPORTED`), a target property (`IMPORTED`), or a concept (Imported Targets).

Where a property, command or variable is related conceptually to others, by for example, being related to the buildsystem description, generator expressions or Qt, each relevant property, command or variable should link to the primary manual, which provides high-level information. Only particular information relating to the command should be in the documentation of the command.

Style: Referencing CMake Domain Objects

When referring to CMake Domain objects such as properties, variables, commands etc, prefer to link to the target object and follow that with the type of object it is. For example:

```
Set the :prop_tgt:`AUTOMOC` target property to ``ON``.
```

Instead of

```
Set the target property :prop_tgt:`AUTOMOC` to ``ON``.
```

The `policy` directive is an exception, and the type us usually referred to before the link:

```
If policy :prop_tgt:`CMP0022` is set to ``NEW`` the behavior is ...
```

However, markup self-references with `inline-literal` syntax. For example, within the `add_executable()` command documentation, use

```
``add_executable``
```

not

```
:command:`add_executable`
```

which is used elsewhere.

Modules

The `Modules` directory contains CMake-language `.cmake` module files.

Module Documentation

To document CMake module `Modules/<module-name>.cmake`, modify `Help/manual/cmake-modules.7.rst` to reference the module in the `toctree` directive, in sorted order, as:

```
/module/<module-name>
```

Then add the module document file `Help/module/<module-name>.rst` containing just the line:

```
.. cmake-module:: ../../Modules/<module-name>.cmake
```

The `cmake-module` directive will scan the module file to extract reStructuredText markup from comment blocks that start in `.rst:`. At the top of `Modules/<module-name>.cmake`, begin with the following license notice:

```
# Distributed under the OSI-approved BSD 3-Clause License.  See accompanying
# file Copyright.txt or https://cmake.org/licensing for details.
```

After this notice, add a *BLANK* line. Then, add documentation using a Line Comment block of the form:

```
#.rst:
# <module-name>
# -------------
#
# <reStructuredText documentation of module>
```

or a Bracket Comment of the form:

```
#[[.rst:
<module-name>
-------------

<reStructuredText documentation of module>
#]]
```

Any number of `=` may be used in the opening and closing brackets as long as they match. Content on the line containing the closing bracket is excluded if and only if the line starts in `#`.

Additional such `.rst:` comments may appear anywhere in the module file. All such comments must start with `#` in the first column.

For example, a `Modules/Findxxx.cmake` module may contain:

```
# Distributed under the OSI-approved BSD 3-Clause License.  See accompanying
# file Copyright.txt or https://cmake.org/licensing for details.

#.rst:
# FindXxx
# -------
#
# This is a cool module.
# This module does really cool stuff.
# It can do even more than you think.
#
# It even needs two paragraphs to tell you about it.
# And it defines the following variables:
#
# * VAR_COOL: this is great isn't it?
# * VAR_REALLY_COOL: cool right?

<code>

#[=======================================[.rst:
.. command:: xxx_do_something

 This command does something for Xxx::

  xxx_do_something(some arguments)
#]=======================================]
macro(xxx_do_something)
```

```
  <code>
endmacro()
```

Test the documentation formatting by running `cmake --help-module <module-name>`, and also by enabling the `SPHINX_HTML` and `SPHINX_MAN` options to build the documentation. Edit the comments until generated documentation looks satisfactory. To have a .cmake file in this directory NOT show up in the modules documentation, simply leave out the `Help/module/<module-name>.rst` file and the `Help/manual/cmake-modules.7.rst` toctree entry.

Find Modules

A "find module" is a `Modules/Find<PackageName>.cmake` file to be loaded by the `find_package()` command when invoked for `<PackageName>`.

The primary task of a find module is to determine whether a package exists on the system, set the `<PackageName>_FOUND` variable to reflect this and provide any variables, macros and imported targets required to use the package. A find module is useful in cases where an upstream library does not provide a config file package.

The traditional approach is to use variables for everything, including libraries and executables: see the Standard Variable Names section below. This is what most of the existing find modules provided by CMake do.

The more modern approach is to behave as much like config file packages files as possible, by providing imported target. This has the advantage of propagating Transitive Usage Requirements to consumers.

In either case (or even when providing both variables and imported targets), find modules should provide backwards compatibility with old versions that had the same name.

A FindFoo.cmake module will typically be loaded by the command:

```
find_package(Foo [major[.minor[.patch[.tweak]]]]
             [EXACT] [QUIET] [REQUIRED]
             [[COMPONENTS] [components...]]
             [OPTIONAL_COMPONENTS components...]
             [NO_POLICY_SCOPE])
```

See the `find_package()` documentation for details on what variables are set for the find module. Most of these are dealt with by using `FindPackageHandleStandardArgs`.

Briefly, the module should only locate versions of the package compatible with the requested version, as described by the `Foo_FIND_VERSION` family of variables. If `Foo_FIND_QUIETLY` is set to true, it should avoid printing messages, including anything complaining about the package not being found. If `Foo_FIND_REQUIRED` is set to true, the module should issue a `FATAL_ERROR` if the package cannot be found. If neither are set to true, it should print a non-fatal message if it cannot find the package.

Packages that find multiple semi-independent parts (like bundles of libraries) should search for the components listed in `Foo_FIND_COMPONENTS` if it is set , and only set `Foo_FOUND` to true if for each searched-for component `<c>` that was not found, `Foo_FIND_REQUIRED_<c>` is not set to true. The `HANDLE_COMPONENTS` argument of `find_package_handle_standard_args()` can be used to implement this.

If `Foo_FIND_COMPONENTS` is not set, which modules are searched for and required is up to the find module, but should be documented.

For internal implementation, it is a generally accepted convention that variables starting with underscore are for temporary use only.

Like all modules, find modules should be properly documented. To add a module to the CMake documentation, follow the steps in the Module Documentation section above.

Standard Variable Names

For a `FindXxx.cmake` module that takes the approach of setting variables (either instead of or in addition to creating imported targets), the following variable names should be used to keep things consistent between find modules. Note that all variables start with `Xxx_` to make sure they do not interfere with other find modules; the same consideration applies to macros, functions and imported targets.

- `Xxx_INCLUDE_DIRS`

  The final set of include directories listed in one variable for use by client code. This should not be a cache entry.

- `Xxx_LIBRARIES`

  The libraries to link against to use Xxx. These should include full paths. This should not be a cache entry.

- `Xxx_DEFINITIONS`

  Definitions to use when compiling code that uses Xxx. This really shouldn't include options such as `-DHAS_JPEG` that a client source-code file uses to decide whether to `#include <jpeg.h>`

- `Xxx_EXECUTABLE`

  Where to find the Xxx tool.

- `Xxx_Yyy_EXECUTABLE`

  Where to find the Yyy tool that comes with Xxx.

- `Xxx_LIBRARY_DIRS`

  Optionally, the final set of library directories listed in one variable for use by client code. This should not be a cache entry.

- `Xxx_ROOT_DIR`

  Where to find the base directory of Xxx.

- `Xxx_VERSION_Yy`

  Expect Version Yy if true. Make sure at most one of these is ever true.

- `Xxx_WRAP_Yy`

  If False, do not try to use the relevant CMake wrapping command.

- `Xxx_Yy_FOUND`

  If False, optional Yy part of Xxx system is not available.

- `Xxx_FOUND`

  Set to false, or undefined, if we haven't found, or don't want to use Xxx.

- `Xxx_NOT_FOUND_MESSAGE`

  Should be set by config-files in the case that it has set `Xxx_FOUND` to FALSE. The contained message will be printed by the `find_package()` command and by `find_package_handle_standard_args()` to inform the user about the problem.

- `Xxx_RUNTIME_LIBRARY_DIRS`

  Optionally, the runtime library search path for use when running an executable linked to shared libraries. The list should be used by user code to create the `PATH` on windows or `LD_LIBRARY_PATH` on UNIX. This should not be a cache entry.

- `Xxx_VERSION`

  The full version string of the package found, if any. Note that many existing modules provide `Xxx_VERSION_STRING` instead.

- `Xxx_VERSION_MAJOR`

  The major version of the package found, if any.

- `Xxx_VERSION_MINOR`

  The minor version of the package found, if any.

- `Xxx_VERSION_PATCH`

  The patch version of the package found, if any.

The following names should not usually be used in CMakeLists.txt files, but are typically cache variables for users to edit and control the behaviour of find modules (like entering the path to a library manually)

- `Xxx_LIBRARY`

  The path of the Xxx library (as used with `find_library()`, for example).

- `Xxx_Yy_LIBRARY`

  The path of the Yy library that is part of the Xxx system. It may or may not be required to use Xxx.

- `Xxx_INCLUDE_DIR`

  Where to find headers for using the Xxx library.

- `Xxx_Yy_INCLUDE_DIR`

  Where to find headers for using the Yy library of the Xxx system.

To prevent users being overwhelmed with settings to configure, try to keep as many options as possible out of the cache, leaving at least one option which can be used to disable use of the module, or locate a not-found library (e.g. `Xxx_ROOT_DIR`). For the same reason, mark most cache options as advanced. For packages which provide both debug and release binaries, it is common to create cache variables with a `_LIBRARY_<CONFIG>` suffix, such as `Foo_LIBRARY_RELEASE` and `Foo_LIBRARY_DEBUG`.

While these are the standard variable names, you should provide backwards compatibility for any old names that were actually in use. Make sure you comment them as deprecated, so that no-one starts using them.

A Sample Find Module

We will describe how to create a simple find module for a library `Foo`.

The first thing that is needed is a license notice.

```
# Distributed under the OSI-approved BSD 3-Clause License.  See accompanying
# file Copyright.txt or https://cmake.org/licensing for details.
```

Next we need module documentation. CMake's documentation system requires you to follow the license notice with a blank line and then with a documentation marker and the name of the module. You should follow this with a simple statement of what the module does.

```
#.rst:
# FindFoo
# -------
#
# Finds the Foo library
#
```

More description may be required for some packages. If there are caveats or other details users of the module should be aware of, you can add further paragraphs below this. Then you need to document what variables and imported targets are set by the module, such as

```
# This will define the following variables::
#
#   Foo_FOUND    - True if the system has the Foo library
#   Foo_VERSION  - The version of the Foo library which was found
#
# and the following imported targets::
#
#   Foo::Foo   - The Foo library
```

If the package provides any macros, they should be listed here, but can be documented where they are defined. See the Module Documentation section above for more details.

Now the actual libraries and so on have to be found. The code here will obviously vary from module to module (dealing with that, after all, is the point of find modules), but there tends to be a common pattern for libraries.

First, we try to use `pkg-config` to find the library. Note that we cannot rely on this, as it may not be available, but it provides a good starting point.

```
find_package(PkgConfig)
pkg_check_modules(PC_Foo QUIET Foo)
```

This should define some variables starting `PC_Foo_` that contain the information from the `Foo.pc` file.

Now we need to find the libraries and include files; we use the information from `pkg-config` to provide hints to CMake about where to look.

```
find_path(Foo_INCLUDE_DIR
  NAMES foo.h
  PATHS ${PC_Foo_INCLUDE_DIRS}
  PATH_SUFFIXES Foo
)
find_library(Foo_LIBRARY
  NAMES foo
  PATHS ${PC_Foo_LIBRARY_DIRS}
)
```

If you have a good way of getting the version (from a header file, for example), you can use that information to set `Foo_VERSION` (although note that find modules have traditionally used `Foo_VERSION_STRING`, so you may want to set both). Otherwise, attempt to use the information from `pkg-config`

```
set(Foo_VERSION ${PC_Foo_VERSION})
```

Now we can use `FindPackageHandleStandardArgs` to do most of the rest of the work for us

```
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(Foo
  FOUND_VAR Foo_FOUND
  REQUIRED_VARS
    Foo_LIBRARY
    Foo_INCLUDE_DIR
  VERSION_VAR Foo_VERSION
)
```

This will check that the `REQUIRED_VARS` contain values (that do not end in `-NOTFOUND`) and set `Foo_FOUND` appropriately. It will also cache those values. If `Foo_VERSION` is set, and a required version was passed to `find_package()`, it will check the requested version against the one in `Foo_VERSION`. It will also print messages as appropriate; note that if the package was found, it will print the contents of the first required variable to indicate where it was found.

At this point, we have to provide a way for users of the find module to link to the library or libraries that were found. There are two approaches, as discussed in the Find Modules section above. The traditional variable approach looks like

```
if(Foo_FOUND)
  set(Foo_LIBRARIES ${Foo_LIBRARY})
  set(Foo_INCLUDE_DIRS ${Foo_INCLUDE_DIR})
  set(Foo_DEFINITIONS ${PC_Foo_CFLAGS_OTHER})
endif()
```

If more than one library was found, all of them should be included in these variables (see the Standard Variable Names section for more information).

When providing imported targets, these should be namespaced (hence the `Foo::` prefix); CMake will recognize that values passed to `target_link_libraries()` that contain `::` in their name are supposed to be imported targets (rather than just library names), and will produce appropriate diagnostic messages if that target does not exist (see policy `CMP0028`).

```
if(Foo_FOUND AND NOT TARGET Foo::Foo)
  add_library(Foo::Foo UNKNOWN IMPORTED)
  set_target_properties(Foo::Foo PROPERTIES
    IMPORTED_LOCATION "${Foo_LIBRARY}"
    INTERFACE_COMPILE_OPTIONS "${PC_Foo_CFLAGS_OTHER}"
    INTERFACE_INCLUDE_DIRECTORIES "${Foo_INCLUDE_DIR}"
  )
endif()
```

One thing to note about this is that the `INTERFACE_INCLUDE_DIRECTORIES` and similar properties should only contain information about the target itself, and not any of its dependencies. Instead, those dependencies should also be targets, and CMake should be told that they are dependencies of this target. CMake will then combine all the necessary information automatically.

The type of the `IMPORTED` target created in the `add_library()` command can always be specified as `UNKNOWN` type. This simplifies the code in cases where static or shared variants may be found, and CMake will determine the type by inspecting the files.

If the library is available with multiple configurations, the `IMPORTED_CONFIGURATIONS` target property should also be populated:

```
if(Foo_FOUND)
  if (NOT TARGET Foo::Foo)
    add_library(Foo::Foo UNKNOWN IMPORTED)
  endif()
  if (Foo_LIBRARY_RELEASE)
    set_property(TARGET Foo::Foo APPEND PROPERTY
      IMPORTED_CONFIGURATIONS RELEASE
    )
    set_target_properties(Foo::Foo PROPERTIES
      IMPORTED_LOCATION_RELEASE "${Foo_LIBRARY_RELEASE}"
    )
  endif()
  if (Foo_LIBRARY_DEBUG)
    set_property(TARGET Foo::Foo APPEND PROPERTY
      IMPORTED_CONFIGURATIONS DEBUG
    )
    set_target_properties(Foo::Foo PROPERTIES
      IMPORTED_LOCATION_DEBUG "${Foo_LIBRARY_DEBUG}"
    )
  endif()
  set_target_properties(Foo::Foo PROPERTIES
    INTERFACE_COMPILE_OPTIONS "${PC_Foo_CFLAGS_OTHER}"
    INTERFACE_INCLUDE_DIRECTORIES "${Foo_INCLUDE_DIR}"
  )
endif()
```

The `RELEASE` variant should be listed first in the property so that the variant is chosen if the user uses a configuration which is not an exact match for any listed `IMPORTED_CONFIGURATIONS`.

Most of the cache variables should be hidden in the `ccmake` interface unless the user explicitly asks to edit them.

```
mark_as_advanced(
  Foo_INCLUDE_DIR
  Foo_LIBRARY
)
```

If this module replaces an older version, you should set compatibility variables to cause the least disruption possible.

```
# compatibility variables
set(Foo_VERSION_STRING ${Foo_VERSION})
```

==========

cmake-buildsystem(7)

Contents

Introduction

A CMake-based buildsystem is organized as a set of high-level logical targets. Each target corresponds to an executable or library, or is a custom target containing custom commands. Dependencies between the targets are expressed in the buildsystem to determine the build order and the rules for regeneration in response to change.

Binary Targets

Executables and libraries are defined using the `add_executable()` and `add_library()` commands. The resulting binary files have appropriate prefixes, suffixes and extensions for the platform targeted. Dependencies between binary targets are expressed using the `target_link_libraries()` command:

```
add_library(archive archive.cpp zip.cpp lzma.cpp)
add_executable(zipapp zipapp.cpp)
target_link_libraries(zipapp archive)
```

`archive` is defined as a static library – an archive containing objects compiled from `archive.cpp`, `zip.cpp`, and `lzma.cpp`. `zipapp` is defined as an executable formed by compiling and linking `zipapp.cpp`. When linking the `zipapp` executable, the `archive` static library is linked in.

Binary Executables

The `add_executable()` command defines an executable target:

```
add_executable(mytool mytool.cpp)
```

Commands such as `add_custom_command()`, which generates rules to be run at build time can transparently use an `EXECUTABLE` target as a `COMMAND` executable. The buildsystem rules will ensure that the executable is built before attempting to run the command.

Binary Library Types

Normal Libraries

By default, the `add_library()` command defines a static library, unless a type is specified. A type may be specified when using the command:

```
add_library(archive SHARED archive.cpp zip.cpp lzma.cpp)
add_library(archive STATIC archive.cpp zip.cpp lzma.cpp)
```

The `BUILD_SHARED_LIBS` variable may be enabled to change the behavior of `add_library()` to build shared libraries by default.

In the context of the buildsystem definition as a whole, it is largely irrelevant whether particular libraries are `SHARED` or `STATIC` – the commands, dependency specifications and other APIs work similarly regardless of the library type. The `MODULE` library type is dissimilar in that it is generally not linked to – it is not used in the right-hand-side of the `target_link_libraries()` command. It is a type which is loaded as a plugin using runtime techniques. If the library does not export any unmanaged symbols (e.g. Windows resource DLL, C++/CLI DLL), it is required that the library not be a `SHARED` library because CMake expects `SHARED` libraries to export at least one symbol.

```
add_library(archive MODULE 7z.cpp)
```

Apple Frameworks

A `SHARED` library may be marked with the `FRAMEWORK` target property to create an macOS or iOS Framework Bundle. The `MACOSX_FRAMEWORK_IDENTIFIER` sets `CFBundleIdentifier` key and it uniquely identifies the bundle.

```
add_library(MyFramework SHARED MyFramework.cpp)
set_target_properties(MyFramework PROPERTIES
  FRAMEWORK TRUE
  FRAMEWORK_VERSION A
  MACOSX_FRAMEWORK_IDENTIFIER org.cmake.MyFramework
)
```

Object Libraries

The `OBJECT` library type defines a non-archival collection of object files resulting from compiling the given source files. The object files collection may be used as source inputs to other targets:

```
add_library(archive OBJECT archive.cpp zip.cpp lzma.cpp)

add_library(archiveExtras STATIC $<TARGET_OBJECTS:archive> extras.cpp)

add_executable(test_exe $<TARGET_OBJECTS:archive> test.cpp)
```

The link (or archiving) step of those other targets will use the object files collection in addition to those from their own sources.

Alternatively, object libraries may be linked into other targets:

```
add_library(archive OBJECT archive.cpp zip.cpp lzma.cpp)

add_library(archiveExtras STATIC extras.cpp)
target_link_libraries(archiveExtras PUBLIC archive)

add_executable(test_exe test.cpp)
target_link_libraries(test_exe archive)
```

The link (or archiving) step of those other targets will use the object files from object libraries that are *directly* linked. Additionally, usage requirements of the object libraries will be honored when compiling sources in those other targets. Furthermore, those usage requirements will propagate transitively to dependents of those other targets.

Object libraries may not be used as the `TARGET` in a use of the `add_custom_command(TARGET)` command signature. However, the list of objects can be used by `add_custom_command(OUTPUT)` or `file(GENERATE)` by using `$<TARGET_OBJECTS:objlib>`.

Build Specification and Usage Requirements

The `target_include_directories()`, `target_compile_definitions()` and `target_compile_options()` commands specify the build specifications and the usage requirements of binary targets. The commands populate the `INCLUDE_DIRECTORIES`, `COMPILE_DEFINITIONS` and `COMPILE_OPTIONS` target properties respectively, and/or the `INTERFACE_INCLUDE_DIRECTORIES`, `INTERFACE_COMPILE_DEFINITIONS` and `INTERFACE_COMPILE_OPTIONS` target properties.

Each of the commands has a `PRIVATE`, `PUBLIC` and `INTERFACE` mode. The `PRIVATE` mode populates only the non-`INTERFACE_` variant of the target property and the `INTERFACE` mode populates only the `INTERFACE_` variants. The `PUBLIC` mode populates both variants of the respective target property. Each command may be invoked with multiple uses of each keyword:

```
target_compile_definitions(archive
  PRIVATE BUILDING_WITH_LZMA
  INTERFACE USING_ARCHIVE_LIB
)
```

Note that usage requirements are not designed as a way to make downstreams use particular `COMPILE_OPTIONS` or `COMPILE_DEFINITIONS` etc for convenience only. The contents of the properties must be **requirements**, not merely recommendations or convenience.

See the Creating Relocatable Packages section of the `cmake-packages(7)` manual for discussion of additional care that must be taken when specifying usage requirements while creating packages for redistribution.

Target Properties

The contents of the `INCLUDE_DIRECTORIES`, `COMPILE_DEFINITIONS` and `COMPILE_OPTIONS` target properties are used appropriately when compiling the source files of a binary target.

Entries in the `INCLUDE_DIRECTORIES` are added to the compile line with `-I` or `-isystem` prefixes and in the order of appearance in the property value.

Entries in the `COMPILE_DEFINITIONS` are prefixed with `-D` or `/D` and added to the compile line in an unspecified order. The `DEFINE_SYMBOL` target property is also added as a compile definition as a special convenience case for `SHARED` and `MODULE` library targets.

Entries in the `COMPILE_OPTIONS` are escaped for the shell and added in the order of appearance in the property value. Several compile options have special separate handling, such as `POSITION_INDEPENDENT_CODE`.

The contents of the `INTERFACE_INCLUDE_DIRECTORIES`, `INTERFACE_COMPILE_DEFINITIONS` and `INTERFACE_COMPILE_OPTIONS` target properties are *Usage Requirements* – they specify content which consumers must use to correctly compile and link with the target they appear on. For any binary target, the contents of each `INTERFACE_` property on each target specified in a `target_link_libraries()` command is consumed:

```
set(srcs archive.cpp zip.cpp)
if (LZMA_FOUND)
  list(APPEND srcs lzma.cpp)
endif()
add_library(archive SHARED ${srcs})
if (LZMA_FOUND)
  # The archive library sources are compiled with -DBUILDING_WITH_LZMA
  target_compile_definitions(archive PRIVATE BUILDING_WITH_LZMA)
endif()
target_compile_definitions(archive INTERFACE USING_ARCHIVE_LIB)

add_executable(consumer)
# Link consumer to archive and consume its usage requirements. The consumer
# executable sources are compiled with -DUSING_ARCHIVE_LIB.
target_link_libraries(consumer archive)
```

Because it is common to require that the source directory and corresponding build directory are added to the `INCLUDE_DIRECTORIES`, the `CMAKE_INCLUDE_CURRENT_DIR` variable can be enabled to conveniently add the corresponding directories to the `INCLUDE_DIRECTORIES` of all targets. The variable `CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE` can be enabled to add the corresponding directories to the `INTERFACE_INCLUDE_DIRECTORIES` of all targets. This makes use of targets in multiple different directories convenient through use of the `target_link_libraries()` command.

Transitive Usage Requirements

The usage requirements of a target can transitively propagate to dependents. The `target_link_libraries()` command has `PRIVATE`, `INTERFACE` and `PUBLIC` keywords to control the propagation.

```
add_library(archive archive.cpp)
target_compile_definitions(archive INTERFACE USING_ARCHIVE_LIB)

add_library(serialization serialization.cpp)
target_compile_definitions(serialization INTERFACE USING_SERIALIZATION_LIB)

add_library(archiveExtras extras.cpp)
```

```
target_link_libraries(archiveExtras PUBLIC archive)
target_link_libraries(archiveExtras PRIVATE serialization)
# archiveExtras is compiled with -DUSING_ARCHIVE_LIB
# and -DUSING_SERIALIZATION_LIB

add_executable(consumer consumer.cpp)
# consumer is compiled with -DUSING_ARCHIVE_LIB
target_link_libraries(consumer archiveExtras)
```

Because `archive` is a `PUBLIC` dependency of `archiveExtras`, the usage requirements of it are propagated to `consumer` too. Because `serialization` is a `PRIVATE` dependency of `archiveExtras`, the usage requirements of it are not propagated to `consumer`.

Generally, a dependency should be specified in a use of `target_link_libraries()` with the `PRIVATE` keyword if it is used by only the implementation of a library, and not in the header files. If a dependency is additionally used in the header files of a library (e.g. for class inheritance), then it should be specified as a `PUBLIC` dependency. A dependency which is not used by the implementation of a library, but only by its headers should be specified as an `INTERFACE` dependency. The `target_link_libraries()` command may be invoked with multiple uses of each keyword:

```
target_link_libraries(archiveExtras
  PUBLIC archive
  PRIVATE serialization
)
```

Usage requirements are propagated by reading the `INTERFACE_` variants of target properties from dependencies and appending the values to the non-`INTERFACE_` variants of the operand. For example, the `INTERFACE_INCLUDE_DIRECTORIES` of dependencies is read and appended to the `INCLUDE_DIRECTORIES` of the operand. In cases where order is relevant and maintained, and the order resulting from the `target_link_libraries()` calls does not allow correct compilation, use of an appropriate command to set the property directly may update the order.

For example, if the linked libraries for a target must be specified in the order `lib1` `lib2` `lib3`, but the include directories must be specified in the order `lib3` `lib1` `lib2`:

```
target_link_libraries(myExe lib1 lib2 lib3)
target_include_directories(myExe
  PRIVATE $<TARGET_PROPERTY:lib3,INTERFACE_INCLUDE_DIRECTORIES>)
```

Note that care must be taken when specifying usage requirements for targets which will be exported for installation using the `install(EXPORT)` command. See Creating Packages for more.

Compatible Interface Properties

Some target properties are required to be compatible between a target and the interface of each dependency. For example, the `POSITION_INDEPENDENT_CODE` target property may specify a boolean value of whether a target should be compiled as position-independent-code, which has platform-specific consequences. A target may also specify the usage requirement `INTERFACE_POSITION_INDEPENDENT_CODE` to communicate that

consumers must be compiled as position-independent-code.

```
add_executable(exe1 exe1.cpp)
set_property(TARGET exe1 PROPERTY POSITION_INDEPENDENT_CODE ON)

add_library(lib1 SHARED lib1.cpp)
set_property(TARGET lib1 PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE ON)

add_executable(exe2 exe2.cpp)
target_link_libraries(exe2 lib1)
```

Here, both `exe1` and `exe2` will be compiled as position-independent-code. `lib1` will also be compiled as position-independent-code because that is the default setting for `SHARED` libraries. If dependencies have conflicting, non-compatible requirements `cmake(1)` issues a diagnostic:

```
add_library(lib1 SHARED lib1.cpp)
set_property(TARGET lib1 PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE ON)

add_library(lib2 SHARED lib2.cpp)
set_property(TARGET lib2 PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE OFF)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1)
set_property(TARGET exe1 PROPERTY POSITION_INDEPENDENT_CODE OFF)

add_executable(exe2 exe2.cpp)
target_link_libraries(exe2 lib1 lib2)
```

The `lib1` requirement `INTERFACE_POSITION_INDEPENDENT_CODE` is not "compatible" with the `POSITION_INDEPENDENT_CODE` property of the `exe1` target. The library requires that consumers are built as position-independent-code, while the executable specifies to not built as position-independent-code, so a diagnostic is issued.

The `lib1` and `lib2` requirements are not "compatible". One of them requires that consumers are built as position-independent-code, while the other requires that consumers are not built as position-independent-code. Because `exe2` links to both and they are in conflict, a diagnostic is issued.

To be "compatible", the `POSITION_INDEPENDENT_CODE` property, if set must be either the same, in a boolean sense, as the `INTERFACE_POSITION_INDEPENDENT_CODE` property of all transitively specified dependencies on which that property is set.

This property of "compatible interface requirement" may be extended to other properties by specifying the property in the content of the `COMPATIBLE_INTERFACE_BOOL` target property. Each specified property must be compatible between the consuming target and the corresponding property with an `INTERFACE_` prefix from each dependency:

```
add_library(lib1Version2 SHARED lib1_v2.cpp)
set_property(TARGET lib1Version2 PROPERTY INTERFACE_CUSTOM_PROP ON)
set_property(TARGET lib1Version2 APPEND PROPERTY
  COMPATIBLE_INTERFACE_BOOL CUSTOM_PROP
)
```

```
add_library(lib1Version3 SHARED lib1_v3.cpp)
set_property(TARGET lib1Version3 PROPERTY INTERFACE_CUSTOM_PROP OFF)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1Version2) # CUSTOM_PROP will be ON

add_executable(exe2 exe2.cpp)
target_link_libraries(exe2 lib1Version2 lib1Version3) # Diagnostic
```

Non-boolean properties may also participate in "compatible interface" computations. Properties specified in the `COMPATIBLE_INTERFACE_STRING` property must be either unspecified or compare to the same string among all transitively specified dependencies. This can be useful to ensure that multiple incompatible versions of a library are not linked together through transitive requirements of a target:

```
add_library(lib1Version2 SHARED lib1_v2.cpp)
set_property(TARGET lib1Version2 PROPERTY INTERFACE_LIB_VERSION 2)
set_property(TARGET lib1Version2 APPEND PROPERTY
  COMPATIBLE_INTERFACE_STRING LIB_VERSION
)

add_library(lib1Version3 SHARED lib1_v3.cpp)
set_property(TARGET lib1Version3 PROPERTY INTERFACE_LIB_VERSION 3)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1Version2) # LIB_VERSION will be "2"

add_executable(exe2 exe2.cpp)
target_link_libraries(exe2 lib1Version2 lib1Version3) # Diagnostic
```

The `COMPATIBLE_INTERFACE_NUMBER_MAX` target property specifies that content will be evaluated numerically and the maximum number among all specified will be calculated:

```
add_library(lib1Version2 SHARED lib1_v2.cpp)
set_property(TARGET lib1Version2 PROPERTY INTERFACE_CONTAINER_SIZE_REQUIRED 200)
set_property(TARGET lib1Version2 APPEND PROPERTY
  COMPATIBLE_INTERFACE_NUMBER_MAX CONTAINER_SIZE_REQUIRED
)

add_library(lib1Version3 SHARED lib1_v3.cpp)
set_property(TARGET lib1Version3 PROPERTY INTERFACE_CONTAINER_SIZE_REQUIRED 1000)

add_executable(exe1 exe1.cpp)
# CONTAINER_SIZE_REQUIRED will be "200"
target_link_libraries(exe1 lib1Version2)

add_executable(exe2 exe2.cpp)
# CONTAINER_SIZE_REQUIRED will be "1000"
target_link_libraries(exe2 lib1Version2 lib1Version3)
```

Similarly, the `COMPATIBLE_INTERFACE_NUMBER_MIN` may be used to calculate the numeric minimum value for a property from dependencies.

Each calculated "compatible" property value may be read in the consumer at generate-time using generator expressions.

Note that for each dependee, the set of properties specified in each compatible interface property must not intersect with the set specified in any of the other properties.

Property Origin Debugging

Because build specifications can be determined by dependencies, the lack of locality of code which creates a target and code which is responsible for setting build specifications may make the code more difficult to reason about. `cmake(1)` provides a debugging facility to print the origin of the contents of properties which may be determined by dependencies. The properties which can be debugged are listed in the `CMAKE_DEBUG_TARGET_PROPERTIES` variable documentation:

```
set(CMAKE_DEBUG_TARGET_PROPERTIES
  INCLUDE_DIRECTORIES
  COMPILE_DEFINITIONS
  POSITION_INDEPENDENT_CODE
  CONTAINER_SIZE_REQUIRED
  LIB_VERSION
)
add_executable(exe1 exe1.cpp)
```

In the case of properties listed in `COMPATIBLE_INTERFACE_BOOL` or `COMPATIBLE_INTERFACE_STRING`, the debug output shows which target was responsible for setting the property, and which other dependencies also defined the property. In the case of `COMPATIBLE_INTERFACE_NUMBER_MAX` and `COMPATIBLE_INTERFACE_NUMBER_MIN`, the debug output shows the value of the property from each dependency, and whether the value determines the new extreme.

Build Specification with Generator Expressions

Build specifications may use `generator expressions` containing content which may be conditional or known only at generate-time. For example, the calculated "compatible" value of a property may be read with the `TARGET_PROPERTY` expression:

```
add_library(lib1Version2 SHARED lib1_v2.cpp)
set_property(TARGET lib1Version2 PROPERTY
  INTERFACE_CONTAINER_SIZE_REQUIRED 200)
set_property(TARGET lib1Version2 APPEND PROPERTY
  COMPATIBLE_INTERFACE_NUMBER_MAX CONTAINER_SIZE_REQUIRED
)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1Version2)
target_compile_definitions(exe1 PRIVATE
    CONTAINER_SIZE=$<TARGET_PROPERTY:CONTAINER_SIZE_REQUIRED>
)
```

In this case, the `exe1` source files will be compiled with `-DCONTAINER_SIZE=200`.

Configuration determined build specifications may be conveniently set using the `CONFIG` generator expression.

```
target_compile_definitions(exe1 PRIVATE
    $<$<CONFIG:Debug>:DEBUG_BUILD>
)
```

The `CONFIG` parameter is compared case-insensitively with the configuration being built. In the presence of `IMPORTED` targets, the content of `MAP_IMPORTED_CONFIG_DEBUG` is also accounted for by this expression.

Some buildsystems generated by `cmake(1)` have a predetermined build-configuration set in the `CMAKE_BUILD_TYPE` variable. The buildsystem for the IDEs such as Visual Studio and Xcode are generated independent of the build-configuration, and the actual build configuration is not known until build-time. Therefore, code such as

```
string(TOLOWER ${CMAKE_BUILD_TYPE} _type)
if (_type STREQUAL debug)
  target_compile_definitions(exe1 PRIVATE DEBUG_BUILD)
endif()
```

may appear to work for `Makefile` based and `Ninja` generators, but is not portable to IDE generators. Additionally, the `IMPORTED` configuration-mappings are not accounted for with code like this, so it should be avoided.

The unary `TARGET_PROPERTY` generator expression and the `TARGET_POLICY` generator expression are evaluated with the consuming target context. This means that a usage requirement specification may be evaluated differently based on the consumer:

```
add_library(lib1 lib1.cpp)
target_compile_definitions(lib1 INTERFACE
  $<$<STREQUAL:$<TARGET_PROPERTY:TYPE>,EXECUTABLE>:LIB1_WITH_EXE>
  $<$<STREQUAL:$<TARGET_PROPERTY:TYPE>,SHARED_LIBRARY>:LIB1_WITH_SHARED_LIB>
  $<$<TARGET_POLICY:CMP0041>:CONSUMER_CMP0041_NEW>
)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1)

cmake_policy(SET CMP0041 NEW)

add_library(shared_lib shared_lib.cpp)
target_link_libraries(shared_lib lib1)
```

The `exe1` executable will be compiled with `-DLIB1_WITH_EXE`, while the `shared_lib` shared library will be compiled with `-DLIB1_WITH_SHARED_LIB` and `-DCONSUMER_CMP0041_NEW`, because policy `CMP0041` is `NEW` at the point where the `shared_lib` target is created.

The `BUILD_INTERFACE` expression wraps requirements which are only used when consumed from a target in the same buildsystem, or when consumed from a target exported to the build directory using the `export()` command. The `INSTALL_INTERFACE` expression wraps requirements which are only used when consumed from a target which has been installed and exported with the `install(EXPORT)` command:

```
add_library(ClimbingStats climbingstats.cpp)
target_compile_definitions(ClimbingStats INTERFACE
  $<BUILD_INTERFACE:ClimbingStats_FROM_BUILD_LOCATION>
  $<INSTALL_INTERFACE:ClimbingStats_FROM_INSTALLED_LOCATION>
)
install(TARGETS ClimbingStats EXPORT libExport ${InstallArgs})
install(EXPORT libExport NAMESPACE Upstream::
        DESTINATION lib/cmake/ClimbingStats)
export(EXPORT libExport NAMESPACE Upstream::)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 ClimbingStats)
```

In this case, the `exe1` executable will be compiled with `-DClimbingStats_FROM_BUILD_LOCATION`. The exporting commands generate `IMPORTED` targets with either the `INSTALL_INTERFACE` or the `BUILD_INTERFACE` omitted, and the `*_INTERFACE` marker stripped away. A separate project consuming the `ClimbingStats` package would contain:

```
find_package(ClimbingStats REQUIRED)

add_executable(Downstream main.cpp)
target_link_libraries(Downstream Upstream::ClimbingStats)
```

Depending on whether the `ClimbingStats` package was used from the build location or the install location, the `Downstream` target would be compiled with either `-DClimbingStats_FROM_BUILD_LOCATION` or `-DClimbingStats_FROM_INSTALL_LOCATION`. For more about packages and exporting see the `cmake-packages(7)` manual.

Include Directories and Usage Requirements

Include directories require some special consideration when specified as usage requirements and when used with generator expressions. The `target_include_directories()` command accepts both relative and absolute include directories:

```
add_library(lib1 lib1.cpp)
target_include_directories(lib1 PRIVATE
  /absolute/path
  relative/path
)
```

Relative paths are interpreted relative to the source directory where the command appears. Relative paths are not allowed in the `INTERFACE_INCLUDE_DIRECTORIES` of `IMPORTED` targets.

In cases where a non-trivial generator expression is used, the `INSTALL_PREFIX` expression may be used within the argument of an `INSTALL_INTERFACE` expression. It is a replacement marker which expands to the installation prefix when imported by a consuming project.

Include directories usage requirements commonly differ between the build-tree and the install-tree. The `BUILD_INTERFACE` and `INSTALL_INTERFACE` generator expressions can be used to describe separate usage requirements based on the usage location. Relative paths are allowed within the `INSTALL_INTERFACE` expression and are interpreted relative to the installation prefix. For example:

```
add_library(ClimbingStats climbingstats.cpp)
target_include_directories(ClimbingStats INTERFACE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}/generated>
  $<INSTALL_INTERFACE:/absolute/path>
  $<INSTALL_INTERFACE:relative/path>
  $<INSTALL_INTERFACE:$<INSTALL_PREFIX>/$<CONFIG>/generated>
)
```

Two convenience APIs are provided relating to include directories usage requirements. The `CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE` variable may be enabled, with an equivalent effect to:

```
set_property(TARGET tgt APPEND PROPERTY INTERFACE_INCLUDE_DIRECTORIES
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR};${CMAKE_CURRENT_BINARY_DIR}>
)
```

for each target affected. The convenience for installed targets is an `INCLUDES DESTINATION` component with the `install(TARGETS)` command:

```
install(TARGETS foo bar bat EXPORT tgts ${dest_args}
  INCLUDES DESTINATION include
)
install(EXPORT tgts ${other_args})
install(FILES ${headers} DESTINATION include)
```

This is equivalent to appending `${CMAKE_INSTALL_PREFIX}/include` to the `INTERFACE_INCLUDE_DIRECTORIES` of each of the installed `IMPORTED` targets when generated by `install(EXPORT)`.

When the `INTERFACE_INCLUDE_DIRECTORIES` of an imported target is consumed, the entries in the property are treated as `SYSTEM` include directories, as if they were listed in the `INTERFACE_SYSTEM_INCLUDE_DIRECTORIES` of the dependency. This can result in omission of compiler warnings for headers found in those directories. This behavior for Imported Targets may be controlled by setting the `NO_SYSTEM_FROM_IMPORTED` target property on the *consumers* of imported targets.

If a binary target is linked transitively to a Mac OX framework, the `Headers` directory of the framework is also treated as a usage requirement. This has the same effect as passing the framework directory as an include directory.

Link Libraries and Generator Expressions

Like build specifications, `link_libraries` may be specified with generator expression conditions. However, as consumption of usage requirements is based on collection from linked dependencies, there is an additional limitation that the link dependencies must form a "directed acyclic graph". That is, if linking to a target is dependent on the value of a target property, that target property may not be dependent on the linked dependencies:

```
add_library(lib1 lib1.cpp)
add_library(lib2 lib2.cpp)
target_link_libraries(lib1 PUBLIC
  $<$<TARGET_PROPERTY:POSITION_INDEPENDENT_CODE>:lib2>
)
add_library(lib3 lib3.cpp)
set_property(TARGET lib3 PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE ON)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1 lib3)
```

As the value of the `POSITION_INDEPENDENT_CODE` property of the `exe1` target is dependent on the linked libraries (`lib3`), and the edge of linking `exe1` is determined by the same `POSITION_INDEPENDENT_CODE` property, the dependency graph above contains a cycle. `cmake(1)` issues a diagnostic in this case.

Output Artifacts

The buildsystem targets created by the `add_library()` and `add_executable()` commands create rules to create binary outputs. The exact output location of the binaries can only be determined at generate-time because it can depend on the build-configuration and the link-language of linked dependencies etc. `TARGET_FILE`, `TARGET_LINKER_FILE` and related expressions can be used to access the name and location of generated binaries. These expressions do not work for `OBJECT` libraries however, as there is no single file generated by such libraries which is relevant to the expressions.

There are three kinds of output artifacts that may be build by targets as detailed in the following sections. Their classification differs between DLL platforms and non-DLL platforms. All Windows-based systems including Cygwin are DLL platforms.

Runtime Output Artifacts

A *runtime* output artifact of a buildsystem target may be:

- The executable file (e.g. `.exe`) of an executable target created by the `add_executable()` command.
- On DLL platforms: the executable file (e.g. `.dll`) of a shared library target created by the `add_library()` command with the `SHARED` option.

The `RUNTIME_OUTPUT_DIRECTORY` and `RUNTIME_OUTPUT_NAME` target properties may be used to control runtime output artifact locations and names in the build tree.

Library Output Artifacts

A *library* output artifact of a buildsystem target may be:

- The loadable module file (e.g. `.dll` or `.so`) of a module library target created by the `add_library()` command with the `MODULE` option.
- On non-DLL platforms: the shared library file (e.g. `.so` or `.dylib`) of a shared library target created by the `add_library()` command with the `SHARED` option.

The `LIBRARY_OUTPUT_DIRECTORY` and `LIBRARY_OUTPUT_NAME` target properties may be used to control library output artifact locations and names in the build tree.

Archive Output Artifacts

An *archive* output artifact of a buildsystem target may be:

- The static library file (e.g. `.lib` or `.a`) of a static library target created by the `add_library()` command with the `STATIC` option.
- On DLL platforms: the import library file (e.g. `.lib`) of a shared library target created by the `add_library()` command with the `SHARED` option. This file is only guaranteed to exist if the library exports at least one unmanaged symbol.
- On DLL platforms: the import library file (e.g. `.lib`) of an executable target created by the `add_executable()` command when its `ENABLE_EXPORTS` target property is set.

The `ARCHIVE_OUTPUT_DIRECTORY` and `ARCHIVE_OUTPUT_NAME` target properties may be used to control archive output artifact locations and names in the build tree.

Directory-Scoped Commands

The `target_include_directories()`, `target_compile_definitions()` and `target_compile_options()` commands have an effect on only one target at a time. The commands `add_compile_definitions()`, `add_compile_options()` and `include_directories()` have a similar function, but operate at directory scope instead of target scope for convenience.

Pseudo Targets

Some target types do not represent outputs of the buildsystem, but only inputs such as external dependencies, aliases or other non-build artifacts. Pseudo targets are not represented in the generated buildsystem.

Imported Targets

An `IMPORTED` target represents a pre-existing dependency. Usually such targets are defined by an upstream package and should be treated as immutable. After declaring an `IMPORTED` target one can adjust its target properties by using the customary commands such as `target_compile_definitions()`, `target_include_directories()`, `target_compile_options()` or `target_link_libraries()` just like with any other regular target.

`IMPORTED` targets may have the same usage requirement properties populated as binary targets, such as `INTERFACE_INCLUDE_DIRECTORIES`, `INTERFACE_COMPILE_DEFINITIONS`, `INTERFACE_COMPILE_OPTIONS`, `INTERFACE_LINK_LIBRARIES`, and `INTERFACE_POSITION_INDEPENDENT_CODE`.

The `LOCATION` may also be read from an IMPORTED target, though there is rarely reason to do so. Commands such as `add_custom_command()` can transparently use an `IMPORTED` `EXECUTABLE` target as a `COMMAND` executable.

The scope of the definition of an `IMPORTED` target is the directory where it was defined. It may be accessed and used from subdirectories, but not from parent directories or sibling directories. The scope is similar to the scope of a cmake variable.

It is also possible to define a `GLOBAL` `IMPORTED` target which is accessible globally in the buildsystem.

See the `cmake-packages(7)` manual for more on creating packages with `IMPORTED` targets.

Alias Targets

An `ALIAS` target is a name which may be used interchangeably with a binary target name in read-only contexts. A primary use-case for `ALIAS` targets is for example or unit test executables accompanying a library, which may be part of the same buildsystem or built separately based on user configuration.

```
add_library(lib1 lib1.cpp)
install(TARGETS lib1 EXPORT lib1Export ${dest_args})
install(EXPORT lib1Export NAMESPACE Upstream:: ${other_args})

add_library(Upstream::lib1 ALIAS lib1)
```

In another directory, we can link unconditionally to the `Upstream::lib1` target, which may be an `IMPORTED` target from a package, or an `ALIAS` target if built as part of the same buildsystem.

```
if (NOT TARGET Upstream::lib1)
  find_package(lib1 REQUIRED)
endif()
add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 Upstream::lib1)
```

`ALIAS` targets are not mutable, installable or exportable. They are entirely local to the buildsystem description. A name can be tested for whether it is an `ALIAS` name by reading the `ALIASED_TARGET` property from it:

```
get_target_property(_aliased Upstream::lib1 ALIASED_TARGET)
if(_aliased)
  message(STATUS "The name Upstream::lib1 is an ALIAS for ${_aliased}.")
endif()
```

Interface Libraries

An `INTERFACE` target has no `LOCATION` and is mutable, but is otherwise similar to an `IMPORTED` target.

It may specify usage requirements such as `INTERFACE_INCLUDE_DIRECTORIES`, `INTERFACE_COMPILE_DEFINITIONS`, `INTERFACE_COMPILE_OPTIONS`, `INTERFACE_LINK_LIBRARIES`, `INTERFACE_SOURCES`, and `INTERFACE_POSITION_INDEPENDENT_CODE`. Only the `INTERFACE` modes of the `target_include_directories()`, `target_compile_definitions()`, `target_compile_options()`, `target_sources()`, and `target_link_libraries()` commands may be used with `INTERFACE` libraries.

A primary use-case for `INTERFACE` libraries is header-only libraries.

```
add_library(Eigen INTERFACE)
target_include_directories(Eigen INTERFACE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/src>
  $<INSTALL_INTERFACE:include/Eigen>
)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 Eigen)
```

Here, the usage requirements from the `Eigen` target are consumed and used when compiling, but it has no effect on linking.

Another use-case is to employ an entirely target-focussed design for usage requirements:

```
add_library(pic_on INTERFACE)
set_property(TARGET pic_on PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE ON)
add_library(pic_off INTERFACE)
set_property(TARGET pic_off PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE OFF)

add_library(enable_rtti INTERFACE)
target_compile_options(enable_rtti INTERFACE
  $<$<OR:$<COMPILER_ID:GNU>,$<COMPILER_ID:Clang>>:-rtti>
)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 pic_on enable_rtti)
```

This way, the build specification of `exe1` is expressed entirely as linked targets, and the complexity of compiler-specific flags is encapsulated in an `INTERFACE` library target.

The properties permitted to be set on or read from an `INTERFACE` library are:

- Properties matching `INTERFACE_*`
- Built-in properties matching `COMPATIBLE_INTERFACE_*`
- `EXPORT_NAME`
- `IMPORTED`
- `NAME`
- Properties matching `IMPORTED_LIBNAME_*`
- Properties matching `MAP_IMPORTED_CONFIG_*`

`INTERFACE` libraries may be installed and exported. Any content they refer to must be installed separately:

```
add_library(Eigen INTERFACE)
target_include_directories(Eigen INTERFACE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/src>
  $<INSTALL_INTERFACE:include/Eigen>
)

install(TARGETS Eigen EXPORT eigenExport)
install(EXPORT eigenExport NAMESPACE Upstream::
  DESTINATION lib/cmake/Eigen
)
install(FILES
    ${CMAKE_CURRENT_SOURCE_DIR}/src/eigen.h
    ${CMAKE_CURRENT_SOURCE_DIR}/src/vector.h
    ${CMAKE_CURRENT_SOURCE_DIR}/src/matrix.h
  DESTINATION include/Eigen
)
```

==========

cmake-compile-features(7)

Contents

Introduction

Project source code may depend on, or be conditional on, the availability of certain features of the compiler. There are three use-cases which arise: Compile Feature Requirements, Optional Compile Features and Conditional Compilation Options.

While features are typically specified in programming language standards, CMake provides a primary user interface based on granular handling of the features, not the language standard that introduced the feature.

The `CMAKE_C_KNOWN_FEATURES` and `CMAKE_CXX_KNOWN_FEATURES` global properties contain all the features known to CMake, regardless of compiler support for the feature. The `CMAKE_C_COMPILE_FEATURES` and `CMAKE_CXX_COMPILE_FEATURES` variables contain all features CMake knows are known to the compiler, regardless of language standard or compile flags needed to use them.

Features known to CMake are named mostly following the same convention as the Clang feature test macros. The are some exceptions, such as CMake using `cxx_final` and `cxx_override` instead of the single `cxx_override_control` used by Clang.

Compile Feature Requirements

Compile feature requirements may be specified with the `target_compile_features()` command. For example, if a target must be compiled with compiler support for the `cxx_constexpr` feature:

```
add_library(mylib requires_constexpr.cpp)
target_compile_features(mylib PRIVATE cxx_constexpr)
```

In processing the requirement for the `cxx_constexpr` feature, `cmake(1)` will ensure that the in-use C++ compiler is capable of the feature, and will add any necessary flags such as `-std=gnu++11` to the compile lines of C++ files in the `mylib` target. A `FATAL_ERROR` is issued if the compiler is not capable of the feature.

The exact compile flags and language standard are deliberately not part of the user interface for this use-case. CMake will compute the appropriate compile flags to use by considering the features specified for each target.

Such compile flags are added even if the compiler supports the particular feature without the flag. For example, the GNU compiler supports variadic templates (with a warning) even if `-std=gnu++98` is used. CMake adds the `-std=gnu++11` flag if `cxx_variadic_templates` is specified as a requirement.

In the above example, `mylib` requires `cxx_constexpr` when it is built itself, but consumers of `mylib` are not required to use a compiler which supports `cxx_constexpr`. If the interface of `mylib` does require the `cxx_constexpr` feature (or any other known feature), that may be specified with the `PUBLIC` or `INTERFACE` signatures of `target_compile_features()`:

```
add_library(mylib requires_constexpr.cpp)
# cxx_constexpr is a usage-requirement
target_compile_features(mylib PUBLIC cxx_constexpr)

# main.cpp will be compiled with -std=gnu++11 on GNU for cxx_constexpr.
add_executable(myexe main.cpp)
target_link_libraries(myexe mylib)
```

Feature requirements are evaluated transitively by consuming the link implementation. See `cmake-buildsystem(7)` for more on transitive behavior of build properties and usage requirements.

Requiring Language Standards

In projects that use a large number of commonly available features from a particular language standard (e.g. C++ 11) one may specify a meta-feature (e.g. `cxx_std_11`) that requires use of a compiler mode aware of that standard. This is simpler than specifying all the features individually, but does not guarantee the existence of any particular feature. Diagnosis of use of unsupported features will be delayed until compile time.

For example, if C++ 11 features are used extensively in a project's header files, then clients must use a compiler mode aware of C++ 11 or above. This can be requested with the code:

```
target_compile_features(mylib PUBLIC cxx_std_11)
```

In this example, CMake will ensure the compiler is invoked in a mode that is aware of C++ 11 (or above), adding flags such as `-std=gnu++11` if necessary. This applies to sources within `mylib` as well as any dependents (that may include headers from `mylib`).

Availability of Compiler Extensions

Because the `CXX_EXTENSIONS` target property is `ON` by default, CMake uses extended variants of language dialects by default, such as `-std=gnu++11` instead of `-std=c++11`. That target property may be set to `OFF` to use the non-extended variant of the dialect flag. Note that because most compilers enable extensions by default, this could expose cross-platform bugs in user code or in the headers of third-party dependencies.

Optional Compile Features

Compile features may be preferred if available, without creating a hard requirement. For example, a library may provides alternative implementations depending on whether the `cxx_variadic_templates` feature is available:

```
#if Foo_COMPILER_CXX_VARIADIC_TEMPLATES
template<int I, int... Is>
struct Interface;

template<int I>
struct Interface<I>
{
  static int accumulate()
  {
    return I;
  }
};

template<int I, int... Is>
struct Interface
{
  static int accumulate()
  {
    return I + Interface<Is...>::accumulate();
  }
};
#else
template<int I1, int I2 = 0, int I3 = 0, int I4 = 0>
struct Interface
{
  static int accumulate() { return I1 + I2 + I3 + I4; }
};
#endif
```

Such an interface depends on using the correct preprocessor defines for the compiler features. CMake can generate a header file containing such defines using the `WriteCompilerDetectionHeader` module. The module contains the `write_compiler_detection_header` function which accepts parameters to control the content of the generated header file:

```
write_compiler_detection_header(
  FILE "${CMAKE_CURRENT_BINARY_DIR}/foo_compiler_detection.h"
  PREFIX Foo
  COMPILERS GNU
  FEATURES
    cxx_variadic_templates
)
```

Such a header file may be used internally in the source code of a project, and it may be installed and used in the interface of library code.

For each feature listed in `FEATURES`, a preprocessor definition is created in the header file, and defined to either `1` or `0`.

Additionally, some features call for additional defines, such as the `cxx_final` and `cxx_override` features. Rather than being used in `#ifdef` code, the `final` keyword is abstracted by a symbol which is defined to either `final`, a compiler-specific equivalent, or to empty. That way, C++ code can be written to unconditionally use the symbol, and compiler support determines what it is expanded to:

```
struct Interface {
  virtual void Execute() = 0;
};

struct Concrete Foo_FINAL {
  void Execute() Foo_OVERRIDE;
};
```

In this case, `Foo_FINAL` will expand to `final` if the compiler supports the keyword, or to empty otherwise.

In this use-case, the CMake code will wish to enable a particular language standard if available from the compiler. The `CXX_STANDARD` target property variable may be set to the desired language standard for a particular target, and the `CMAKE_CXX_STANDARD` may be set to influence all following targets:

```
write_compiler_detection_header(
  FILE "${CMAKE_CURRENT_BINARY_DIR}/foo_compiler_detection.h"
  PREFIX Foo
  COMPILERS GNU
  FEATURES
    cxx_final cxx_override
)

# Includes foo_compiler_detection.h and uses the Foo_FINAL symbol
# which will expand to 'final' if the compiler supports the requested
# CXX_STANDARD.
add_library(foo foo.cpp)
set_property(TARGET foo PROPERTY CXX_STANDARD 11)
```

```
# Includes foo_compiler_detection.h and uses the Foo_FINAL symbol
# which will expand to 'final' if the compiler supports the feature,
# even though CXX_STANDARD is not set explicitly.  The requirement of
# cxx_constexpr causes CMake to set CXX_STANDARD internally, which
# affects the compile flags.
add_library(foo_impl foo_impl.cpp)
target_compile_features(foo_impl PRIVATE cxx_constexpr)
```

The `write_compiler_detection_header` function also creates compatibility code for other features which have standard equivalents. For example, the `cxx_static_assert` feature is emulated with a template and abstracted via the `<PREFIX>_STATIC_ASSERT` and `<PREFIX>_STATIC_ASSERT_MSG` function-macros.

Conditional Compilation Options

Libraries may provide entirely different header files depending on requested compiler features.

For example, a header at `with_variadics/interface.h` may contain:

```
template<int I, int... Is>
struct Interface;

template<int I>
struct Interface<I>
{
  static int accumulate()
  {
    return I;
  }
};

template<int I, int... Is>
struct Interface
{
  static int accumulate()
  {
    return I + Interface<Is...>::accumulate();
  }
};
```

while a header at `no_variadics/interface.h` may contain:

```
template<int I1, int I2 = 0, int I3 = 0, int I4 = 0>
struct Interface
{
  static int accumulate() { return I1 + I2 + I3 + I4; }
};
```

It would be possible to write a abstraction `interface.h` header containing something like:

```
#include "foo_compiler_detection.h"
#if Foo_COMPILER_CXX_VARIADIC_TEMPLATES
#include "with_variadics/interface.h"
#else
#include "no_variadics/interface.h"
#endif
```

However this could be unmaintainable if there are many files to abstract. What is needed is to use alternative include directories depending on the compiler capabilities.

CMake provides a `COMPILE_FEATURES` `generator expression` to implement such conditions. This may be used with the build-property commands such as `target_include_directories()` and `target_link_libraries()` to set the appropriate `buildsystem` properties:

```
add_library(foo INTERFACE)
set(with_variadics ${CMAKE_CURRENT_SOURCE_DIR}/with_variadics)
set(no_variadics ${CMAKE_CURRENT_SOURCE_DIR}/no_variadics)
target_include_directories(foo
  INTERFACE
    "$<$<COMPILE_FEATURES:cxx_variadic_templates>:${with_variadics}>"
    "$<$<NOT:$<COMPILE_FEATURES:cxx_variadic_templates>>:${no_variadics}>"
  )
```

Consuming code then simply links to the `foo` target as usual and uses the feature-appropriate include directory

```
add_executable(consumer_with consumer_with.cpp)
target_link_libraries(consumer_with foo)
set_property(TARGET consumer_with CXX_STANDARD 11)

add_executable(consumer_no consumer_no.cpp)
target_link_libraries(consumer_no foo)
```

Supported Compilers

CMake is currently aware of the `C++ standards` and `compile features` available from the following `compiler ids` as of the versions specified for each:

- `AppleClang`: Apple Clang for Xcode versions 4.4 though 9.2.
- `Clang`: Clang compiler versions 2.9 through 6.0.
- `GNU`: GNU compiler versions 4.4 through 8.0.
- `MSVC`: Microsoft Visual Studio versions 2010 through 2017.
- `SunPro`: Oracle SolarisStudio versions 12.4 through 12.6.
- `Intel`: Intel compiler versions 12.1 through 17.0.

CMake is currently aware of the `C standards` and `compile features` available from the following `compiler ids` as of the versions specified for each:

- all compilers and versions listed above for C++.
- `GNU`: GNU compiler versions 3.4 through 8.0.

CMake is currently aware of the `C++ standards` and their associated meta-features (e.g. `cxx_std_11`) available from the following `compiler ids` as of the versions specified for each:

- `Cray`: Cray Compiler Environment version 8.1 through 8.5.8.
- `PGI`: PGI version 12.10 through 17.5.
- `XL`: IBM XL version 10.1 through 13.1.5.

CMake is currently aware of the `C standards` and their associated meta-features (e.g. `c_std_99`) available from the following `compiler ids` as of the versions specified for each:

- all compilers and versions listed above with only meta-features for C++.
- `TI`: Texas Instruments compiler.

CMake is currently aware of the `CUDA standards` from the following `compiler ids` as of the versions specified for each:

- `NVIDIA`: NVIDIA nvcc compiler 7.5 though 9.1.

==========

cmake-generator-expressions(7)

Contents

Introduction

Generator expressions are evaluated during build system generation to produce information specific to each build configuration.

Generator expressions are allowed in the context of many target properties, such as `LINK_LIBRARIES`, `INCLUDE_DIRECTORIES`, `COMPILE_DEFINITIONS` and others. They may also be used when using commands to populate those properties, such as `target_link_libraries()`, `target_include_directories()`, `target_compile_definitions()` and others.

This means that they enable conditional linking, conditional definitions used when compiling, and conditional include directories and more. The conditions may be based on the build configuration, target properties, platform information or any other queryable information.

Logical Expressions

Logical expressions are used to create conditional output. The basic expressions are the `0` and `1` expressions. Because other logical expressions evaluate to either `0` or `1`, they can be composed to create conditional output:

```
$<$<CONFIG:Debug>:DEBUG_MODE>
```

expands to `DEBUG_MODE` when the `Debug` configuration is used, and otherwise expands to nothing.

Available logical expressions are:

- `$<BOOL:...>`

  `1` if the `...` is true, else `0`

- `$<AND:?[,?]...>`

  `1` if all `?` are `1`, else `0`The `?` must always be either `0` or `1` in boolean expressions.

- `$<OR:?[,?]...>`

  `0` if all `?` are `0`, else `1`

- `$<NOT:?>`

  `0` if `?` is `1`, else `1`

- `$<IF:?,true-value...,false-value...>`

  `true-value...` if `?` is `1`, `false-value...` if `?` is `0`

- `$<STREQUAL:a,b>`

  `1` if `a` is STREQUAL `b`, else `0`

- `$<EQUAL:a,b>`

  `1` if `a` is EQUAL `b` in a numeric comparison, else `0`

- `$<IN_LIST:a,b>`

  `1` if `a` is IN_LIST `b`, else `0`

- `$<TARGET_EXISTS:tgt>`

  `1` if `tgt` is an existed target name, else `0`.

- `$<CONFIG:cfg>`

  `1` if config is `cfg`, else `0`. This is a case-insensitive comparison. The mapping in `MAP_IMPORTED_CONFIG_` is also considered by this expression when it is evaluated on a property on an `IMPORTED` target.

- `$<PLATFORM_ID:comp>`

  `1` if the CMake-id of the platform matches `comp`, otherwise `0`. See also the `CMAKE_SYSTEM_NAME` variable.

- `$<C_COMPILER_ID:comp>`

  `1` if the CMake-id of the C compiler matches `comp`, otherwise `0`. See also the `CMAKE_<LANG>_COMPILER_ID` variable.

- `$<CXX_COMPILER_ID:comp>`

`1` if the CMake-id of the CXX compiler matches `comp`, otherwise `0`. See also the `CMAKE_<LANG>_COMPILER_ID` variable.

- `$<VERSION_LESS:v1,v2>`

  `1` if `v1` is a version less than `v2`, else `0`.

- `$<VERSION_GREATER:v1,v2>`

  `1` if `v1` is a version greater than `v2`, else `0`.

- `$<VERSION_EQUAL:v1,v2>`

  `1` if `v1` is the same version as `v2`, else `0`.

- `$<VERSION_LESS_EQUAL:v1,v2>`

  `1` if `v1` is a version less than or equal to `v2`, else `0`.

- `$<VERSION_GREATER_EQUAL:v1,v2>`

  `1` if `v1` is a version greater than or equal to `v2`, else `0`.

- `$<C_COMPILER_VERSION:ver>`

  `1` if the version of the C compiler matches `ver`, otherwise `0`. See also the `CMAKE_<LANG>_COMPILER_VERSION` variable.

- `$<CXX_COMPILER_VERSION:ver>`

  `1` if the version of the CXX compiler matches `ver`, otherwise `0`. See also the `CMAKE_<LANG>_COMPILER_VERSION` variable.

- `$<TARGET_POLICY:pol>`

  `1` if the policy `pol` was NEW when the 'head' target was created, else `0`. If the policy was not set, the warning message for the policy will be emitted. This generator expression only works for a subset of policies.

- `$<COMPILE_FEATURES:feature[,feature]...>`

  `1` if all of the `feature` features are available for the 'head' target, and `0` otherwise. If this expression is used while evaluating the link implementation of a target and if any dependency transitively increases the required `C_STANDARD` or `CXX_STANDARD` for the 'head' target, an error is reported. See the `cmake-compile-features(7)` manual for information on compile features and a list of supported compilers.

- `$<COMPILE_LANGUAGE:lang>`

  `1` when the language used for compilation unit matches `lang`, otherwise `0`. This expression may be used to specify compile options, compile definitions, and include directories for source files of a particular language in a target. For example: `add_executable(myapp main.cpp foo.c bar.cpp zot.cu) target_compile_options(myapp PRIVATE $<$<COMPILE_LANGUAGE:CXX>:-fno-exceptions>) target_compile_definitions(myapp PRIVATE $<$<COMPILE_LANGUAGE:CXX>:COMPILING_CXX> $<$<COMPILE_LANGUAGE:CUDA>:COMPILING_CUDA>) target_include_directories(myapp PRIVATE $<$<COMPILE_LANGUAGE:CXX>:/opt/foo/cxx_headers>)` This specifies the use of the `-fno-exceptions` compile option, `COMPILING_CXX` compile definition, and `cxx_headers` include directory for C++ only (compiler id checks elided). It also specifies a `COMPILING_CUDA` compile definition for CUDA.Note that with Visual Studio Generators and `Xcode` there is no way to represent target-wide compile definitions or include directories separately for `C` and `CXX` languages. Also, with Visual Studio Generatorsthere is no way to represent target-wide flags separately for `C` and `CXX` languages. Under these generators,

expressions for both C and C++ sources will be evaluated using `CXX` if there are any C++ sources and otherwise using `C`. A workaround is to create separate libraries for each source file language instead: `add_library(myapp_c foo.c) add_library(myapp_cxx bar.cpp) target_compile_options(myapp_cxx PUBLIC -fno-exceptions) add_executable(myapp main.cpp) target_link_libraries(myapp myapp_c myapp_cxx)`

Informational Expressions

These expressions expand to some information. The information may be used directly, eg:

```
include_directories(/usr/include/$<CXX_COMPILER_ID>/)
```

expands to `/usr/include/GNU/` or `/usr/include/Clang/` etc, depending on the Id of the compiler.

These expressions may also may be combined with logical expressions:

```
$<$<VERSION_LESS:$<CXX_COMPILER_VERSION>,4.2.0>:OLD_COMPILER>
```

expands to `OLD_COMPILER` if the `CMAKE_CXX_COMPILER_VERSION` is less than 4.2.0.

Available informational expressions are:

- `$<CONFIGURATION>`

  Configuration name. Deprecated. Use `CONFIG` instead.

- `$<CONFIG>`

  Configuration name

- `$<PLATFORM_ID>`

  The CMake-id of the platform. See also the `CMAKE_SYSTEM_NAME` variable.

- `$<C_COMPILER_ID>`

  The CMake-id of the C compiler used. See also the `CMAKE_<LANG>_COMPILER_ID` variable.

- `$<CXX_COMPILER_ID>`

  The CMake-id of the CXX compiler used. See also the `CMAKE_<LANG>_COMPILER_ID` variable.

- `$<C_COMPILER_VERSION>`

  The version of the C compiler used. See also the `CMAKE_<LANG>_COMPILER_VERSION` variable.

- `$<CXX_COMPILER_VERSION>`

  The version of the CXX compiler used. See also the `CMAKE_<LANG>_COMPILER_VERSION` variable.

- `$<TARGET_FILE:tgt>`

  Full path to main file (.exe, .so.1.2, .a) where `tgt` is the name of a target.

- `$<TARGET_FILE_NAME:tgt>`

  Name of main file (.exe, .so.1.2, .a).

- `$<TARGET_FILE_DIR:tgt>`

  Directory of main file (.exe, .so.1.2, .a).

- `$<TARGET_LINKER_FILE:tgt>`

  File used to link (.a, .lib, .so) where `tgt` is the name of a target.

- `$<TARGET_LINKER_FILE_NAME:tgt>`

  Name of file used to link (.a, .lib, .so).

- `$<TARGET_LINKER_FILE_DIR:tgt>`

  Directory of file used to link (.a, .lib, .so).

- `$<TARGET_SONAME_FILE:tgt>`

  File with soname (.so.3) where `tgt` is the name of a target.

- `$<TARGET_SONAME_FILE_NAME:tgt>`

  Name of file with soname (.so.3).

- `$<TARGET_SONAME_FILE_DIR:tgt>`

  Directory of with soname (.so.3).

- `$<TARGET_PDB_FILE:tgt>`

  Full path to the linker generated program database file (.pdb) where `tgt` is the name of a target.See also the `PDB_NAME` and `PDB_OUTPUT_DIRECTORY` target properties and their configuration specific variants `PDB_NAME_` and `PDB_OUTPUT_DIRECTORY_`.

- `$<TARGET_PDB_FILE_NAME:tgt>`

  Name of the linker generated program database file (.pdb).

- `$<TARGET_PDB_FILE_DIR:tgt>`

  Directory of the linker generated program database file (.pdb).

- `$<TARGET_BUNDLE_DIR:tgt>`

  Full path to the bundle directory (`my.app`, `my.framework`, or `my.bundle`) where `tgt` is the name of a target.

- `$<TARGET_BUNDLE_CONTENT_DIR:tgt>`

  Full path to the bundle content directory where `tgt` is the name of a target. For the macOS SDK it leads to `my.app/Contents`, `my.framework`, or `my.bundle/Contents`. For all other SDKs (e.g. iOS) it leads to `my.app`, `my.framework`, or `my.bundle` due to the flat bundle structure.

- `$<TARGET_PROPERTY:tgt,prop>`

  Value of the property `prop` on the target `tgt`.Note that `tgt` is not added as a dependency of the target this expression is evaluated on.

- `$<TARGET_PROPERTY:prop>`

  Value of the property `prop` on the target on which the generator expression is evaluated.

- `$<INSTALL_PREFIX>`

  Content of the install prefix when the target is exported via `install(EXPORT)` and empty otherwise.

- `$<COMPILE_LANGUAGE>`

  The compile language of source files when evaluating compile options. See the unary version for notes about portability of this generator expression.

Output Expressions

These expressions generate output, in some cases depending on an input. These expressions may be combined with other expressions for information or logical comparison:

```
-I$<JOIN:$<TARGET_PROPERTY:INCLUDE_DIRECTORIES>, -I>
```

generates a string of the entries in the `INCLUDE_DIRECTORIES` target property with each entry preceded by `-I`. Note that a more-complete use in this situation would require first checking if the INCLUDE_DIRECTORIES property is non-empty:

```
$<$<BOOL:${prop}>:-I$<JOIN:${prop}, -I>>
```

where `${prop}` refers to a helper variable:

```
set(prop "$<TARGET_PROPERTY:INCLUDE_DIRECTORIES>")
```

Available output expressions are:

- `$<0:...>`

  Empty string (ignores `...`)

- `$<1:...>`

  Content of `...`

- `$<JOIN:list,...>`

  Joins the list with the content of `...`

- `$<ANGLE-R>`

  A literal `>`. Used to compare strings which contain a `>` for example.

- `$<COMMA>`

  A literal `,`. Used to compare strings which contain a `,` for example.

- `$<SEMICOLON>`

  A literal `;`. Used to prevent list expansion on an argument with `;`.

- `$<TARGET_NAME:...>`

  Marks `...` as being the name of a target. This is required if exporting targets to multiple dependent export sets. The `...` must be a literal name of a target- it may not contain generator expressions.

- `$<TARGET_NAME_IF_EXISTS:...>`

  Expands to the `...` if the given target exists, an empty string otherwise.

- `$<LINK_ONLY:...>`

  Content of `...` except when evaluated in a link interface while propagating Transitive Usage Requirements, in which case it is the empty string. Intended for use only in an `INTERFACE_LINK_LIBRARIES` target property, perhaps via the `target_link_libraries()` command, to specify private link dependencies without other usage requirements.

- `$<INSTALL_INTERFACE:...>`

  Content of `...` when the property is exported using `install(EXPORT)`, and empty otherwise.

- `$<BUILD_INTERFACE:...>`

  Content of `...` when the property is exported using `export()`, or when the target is used by another target in the same buildsystem. Expands to the empty string otherwise.

- `$<LOWER_CASE:...>`

  Content of `...` converted to lower case.

- `$<UPPER_CASE:...>`

  Content of `...` converted to upper case.

- `$<MAKE_C_IDENTIFIER:...>`

  Content of `...` converted to a C identifier. The conversion follows the same behavior as `string(MAKE_C_IDENTIFIER)`.

- `$<TARGET_OBJECTS:objLib>`

  List of objects resulting from build of `objLib`. `objLib` must be an object of type `OBJECT_LIBRARY`.

- `$<SHELL_PATH:...>`

  Content of `...` converted to shell path style. For example, slashes are converted to backslashes in Windows shells and drive letters are converted to posix paths in MSYS shells. The `...` must be an absolute path.

- `$<GENEX_EVAL:...>`

  Content of `...` evaluated as a generator expression in the current context. This enables consumption of generator expressions whose evaluation results itself in generator expressions.

- `$<TARGET_GENEX_EVAL:tgt,...>`

  Content of `...` evaluated as a generator expression in the context of `tgt` target. This enables consumption of custom target properties that themselves contain generator expressions.Having the capability to evaluate generator expressions is very useful when you want to manage custom properties supporting generator expressions. For example:`add_library(foo ...) set_property(TARGET foo PROPERTY CUSTOM_KEYS $<$<CONFIG:DEBUG>:FOO_EXTRA_THINGS> ) add_custom_target(printFooKeys COMMAND ${CMAKE_COMMAND} -E echo $<TARGET_PROPERTY:foo,CUSTOM_KEYS> )` This naive implementation of the `printFooKeys` custom command is wrong because `CUSTOM_KEYS` target property is not evaluated and the content is passed as is (i.e. `$<$<CONFIG:DEBUG>:FOO_EXTRA_THINGS>`).To have the expected result (i.e. `FOO_EXTRA_THINGS` if config is `Debug`), it is required to evaluate the output of `$<TARGET_PROPERTY:foo,CUSTOM_KEYS>`:`add_custom_target(printFooKeys COMMAND ${CMAKE_COMMAND} -E echo $<TARGET_GENEX_EVAL:foo,$<TARGET_PROPERTY:foo,CUSTOM_KEYS>> )`

==========

---

cmake-generators(7)

Contents

Introduction

A *CMake Generator* is responsible for writing the input files for a native build system. Exactly one of the CMake Generators must be selected for a build tree to determine what native build system is to be used. Optionally one of the Extra Generators may be selected as a variant of some of the Command-Line Build Tool Generators to produce project files for an auxiliary IDE.

CMake Generators are platform-specific so each may be available only on certain platforms. The `cmake(1)` command-line tool `--help` output lists available generators on the current platform. Use its `-G` option to specify the generator for a new build tree. The `cmake-gui(1)` offers interactive selection of a generator when creating a new build tree.

CMake Generators

Command-Line Build Tool Generators

These generators support command-line build tools. In order to use them, one must launch CMake from a command-line prompt whose environment is already configured for the chosen compiler and build tool.

Makefile Generators

- Borland Makefiles
- MSYS Makefiles
- MinGW Makefiles
- NMake Makefiles
- NMake Makefiles JOM
- Unix Makefiles
- Watcom WMake

Ninja Generator

- Ninja

IDE Build Tool Generators

These generators support Integrated Development Environment (IDE) project files. Since the IDEs configure their own environment one may launch CMake from any environment.

Visual Studio Generators

- Visual Studio 6
- Visual Studio 7
- Visual Studio 7 .NET 2003
- Visual Studio 8 2005
- Visual Studio 9 2008
- Visual Studio 10 2010
- Visual Studio 11 2012
- Visual Studio 12 2013
- Visual Studio 14 2015
- Visual Studio 15 2017

Other Generators

- Green Hills MULTI
- Xcode

Extra Generators

Some of the CMake Generators listed in the `cmake(1)` command-line tool `--help` output may have variants that specify an extra generator for an auxiliary IDE tool. Such generator names have the form `<extra-generator> - <main-generator>`. The following extra generators are known to CMake.

- CodeBlocks
- CodeLite
- Eclipse CDT4
- Kate
- Sublime Text 2

==========

cmake-language(7)

Contents

Organization

CMake input files are written in the "CMake Language" in source files named `CMakeLists.txt` or ending in a `.cmake` file name extension.

CMake Language source files in a project are organized into:

- Directories (`CMakeLists.txt`),
- Scripts (`<script>.cmake`), and
- Modules (`<module>.cmake`).

Directories

When CMake processes a project source tree, the entry point is a source file called `CMakeLists.txt` in the top-level source directory. This file may contain the entire build specification or use the `add_subdirectory()` command to add subdirectories to the build. Each subdirectory added by the command must also contain a `CMakeLists.txt` file as the entry point to that directory. For each source directory whose `CMakeLists.txt` file is processed CMake generates a corresponding directory in the build tree to act as the default working and output directory.

Scripts

An individual `<script>.cmake` source file may be processed in *script mode* by using the `cmake(1)` command-line tool with the `-P` option. Script mode simply runs the commands in the given CMake Language source file and does not generate a build system. It does not allow CMake commands that define build targets or actions.

Modules

CMake Language code in either Directories or Scripts may use the `include()` command to load a `<module>.cmake` source file in the scope of the including context. See the `cmake-modules(7)` manual page for documentation of modules included with the CMake distribution. Project source trees may also provide their own modules and specify their location(s) in the `CMAKE_MODULE_PATH` variable.

Syntax

Encoding

A CMake Language source file may be written in 7-bit ASCII text for maximum portability across all supported platforms. Newlines may be encoded as either `\n` or `\r\n` but will be converted to `\n` as input files are read.

Note that the implementation is 8-bit clean so source files may be encoded as UTF-8 on platforms with system APIs supporting this encoding. In addition, CMake 3.2 and above support source files encoded in UTF-8 on Windows (using UTF-16 to call system APIs). Furthermore, CMake 3.0 and above allow a leading UTF-8 Byte-Order Mark in source files.

Source Files

A CMake Language source file consists of zero or more Command Invocations separated by newlines and optionally spaces and Comments:

```
file         ::=  file_element*
file_element ::=  command_invocation line_ending |
                  (bracket_comment|space)* line_ending
line_ending  ::=  line_comment? newline
space        ::=  <match '[ \t]+'>
newline      ::=  <match '\n'>
```

Note that any source file line not inside Command Arguments or a Bracket Comment can end in a Line Comment.

Command Invocations

A *command invocation* is a name followed by paren-enclosed arguments separated by whitespace:

```
command_invocation  ::=  space* identifier space* '(' arguments ')'
identifier          ::=  <match '[A-Za-z_][A-Za-z0-9_]*'>
arguments           ::=  argument? separated_arguments*
separated_arguments ::=  separation+ argument? |
                         separation* '(' arguments ')'
separation          ::=  space | line_ending
```

For example:

```
add_executable(hello world.c)
```

Command names are case-insensitive. Nested unquoted parentheses in the arguments must balance. Each `(` or `)` is given to the command invocation as a literal Unquoted Argument. This may be used in calls to the `if()` command to enclose conditions. For example:

```
if(FALSE AND (FALSE OR TRUE)) # evaluates to FALSE
```

Note

CMake versions prior to 3.0 require command name identifiers to be at least 2 characters.

CMake versions prior to 2.8.12 silently accept an Unquoted Argument or a Quoted Argument immediately following a Quoted Argument and not separated by any whitespace. For compatibility, CMake 2.8.12 and higher accept such code but produce a warning.

Command Arguments

There are three types of arguments within Command Invocations:

```
argument ::=  bracket_argument | quoted_argument | unquoted_argument
```

Bracket Argument

A *bracket argument*, inspired by Lua long bracket syntax, encloses content between opening and closing "brackets" of the same length:

```
bracket_argument ::=  bracket_open bracket_content bracket_close
bracket_open     ::=  '[' '='* '['
bracket_content  ::=  <any text not containing a bracket_close with
                       the same number of '=' as the bracket_open>
bracket_close    ::=  ']' '='* ']'
```

An opening bracket is written `[` followed by zero or more `=` followed by `[`. The corresponding closing bracket is written `]` followed by the same number of `=` followed by `]`. Brackets do not nest. A unique length may always be chosen for the opening and closing brackets to contain closing brackets of other lengths.

Bracket argument content consists of all text between the opening and closing brackets, except that one newline immediately following the opening bracket, if any, is ignored. No evaluation of the enclosed content, such as Escape Sequences or Variable References, is performed. A bracket argument is always given to the command invocation as exactly one argument.

For example:

```
message([=[
This is the first line in a bracket argument with bracket length 1.
No \-escape sequences or ${variable} references are evaluated.
This is always one argument even though it contains a ; character.
The text does not end on a closing bracket of length 0 like ]].
It does end in a closing bracket of length 1.
]=])
```

Note

CMake versions prior to 3.0 do not support bracket arguments. They interpret the opening bracket as the start of an Unquoted Argument.

Quoted Argument

A *quoted argument* encloses content between opening and closing double-quote characters:

```
quoted_argument     ::=  '"' quoted_element* '"'
quoted_element      ::=  <any character except '\' or '"'> |
                         escape_sequence |
                         quoted_continuation
quoted_continuation ::=  '\' newline
```

Quoted argument content consists of all text between opening and closing quotes. Both Escape Sequences and Variable References are evaluated. A quoted argument is always given to the command invocation as exactly one argument.

For example:

```
message("This is a quoted argument containing multiple lines.
This is always one argument even though it contains a ; character.
Both \\-escape sequences and ${variable} references are evaluated.
The text does not end on an escaped double-quote like \".
It does end in an unescaped double quote.
")
```

The final `\` on any line ending in an odd number of backslashes is treated as a line continuation and ignored along with the immediately following newline character. For example:

```
message("\
This is the first line of a quoted argument. \
In fact it is the only line but since it is long \
the source code uses line continuation.\
")
```

Note

CMake versions prior to 3.0 do not support continuation with `\`. They report errors in quoted arguments containing lines ending in an odd number of `\` characters.

Unquoted Argument

An *unquoted argument* is not enclosed by any quoting syntax. It may not contain any whitespace, `(`, `)`, `#`, `"`, or `\` except when escaped by a backslash:

```
unquoted_argument ::=  unquoted_element+ | unquoted_legacy
unquoted_element  ::=  <any character except whitespace or one of '()#"\'> |
                       escape_sequence
unquoted_legacy   ::=  <see note in text>
```

Unquoted argument content consists of all text in a contiguous block of allowed or escaped characters. Both Escape Sequences and Variable References are evaluated. The resulting value is divided in the same way Listsdivide into elements. Each non-empty element is given to the command invocation as an argument. Therefore an unquoted argument may be given to a command invocation as zero or more arguments.

For example:

```
foreach(arg
    NoSpace
    Escaped\ Space
    This;Divides;Into;Five;Arguments
    Escaped\;Semicolon
    )
  message("${arg}")
endforeach()
```

Note

To support legacy CMake code, unquoted arguments may also contain double-quoted strings (`"..."`, possibly enclosing horizontal whitespace), and make-style variable references (`$(MAKEVAR)`).

Unescaped double-quotes must balance, may not appear at the beginning of an unquoted argument, and are treated as part of the content. For example, the unquoted arguments `-Da="b c"`, `-Da=$(v)`, and `a" "b"c"d` are each interpreted literally. They may instead be written as quoted arguments `"-Da=\"b c\""`, `"-Da=$(v)"`, and `"a\" \"b\"c\"d"`, respectively.

Make-style references are treated literally as part of the content and do not undergo variable expansion. They are treated as part of a single argument (rather than as separate `$`, `(`, `MAKEVAR`, and `)` arguments).

The above "unquoted_legacy" production represents such arguments. We do not recommend using legacy unquoted arguments in new code. Instead use a Quoted Argument or a Bracket Argument to represent the content.

Escape Sequences

An *escape sequence* is a `\` followed by one character:

```
escape_sequence  ::=  escape_identity | escape_encoded | escape_semicolon
escape_identity  ::=  '\' <match '[^A-Za-z0-9;]'>
escape_encoded   ::=  '\t' | '\r' | '\n'
escape_semicolon ::=  '\;'
```

A `\` followed by a non-alphanumeric character simply encodes the literal character without interpreting it as syntax. A `\t`, `\r`, or `\n` encodes a tab, carriage return, or newline character, respectively. A `\;` outside of any Variable References encodes itself but may be used in an Unquoted Argument to encode the `;` without dividing the argument value on it. A `\;` inside Variable References encodes the literal `;` character. (See also policy `CMP0053` documentation for historical considerations.)

Variable References

A *variable reference* has the form `${variable_name}` and is evaluated inside a Quoted Argument or an Unquoted Argument. A variable reference is replaced by the value of the variable, or by the empty string if the variable is not set. Variable references can nest and are evaluated from the inside out, e.g. `${outer_${inner_variable}_variable}`.

Literal variable references may consist of alphanumeric characters, the characters `/_.+-`, and Escape Sequences. Nested references may be used to evaluate variables of any name. See also policy `CMP0053` documentation for historical considerations and reasons why the `$` is also technically permitted but is discouraged.

The Variables section documents the scope of variable names and how their values are set.

An *environment variable reference* has the form `$ENV{VAR}` and is evaluated in the same contexts as a normal variable reference. See `ENV` for more information.

A *cache variable reference* has the form `$CACHE{VAR}` and is evaluated in the same contexts as a normal variable reference. See `CACHE` for more information.

Comments

A comment starts with a `#` character that is not inside a Bracket Argument, Quoted Argument, or escaped with `\` as part of an Unquoted Argument. There are two types of comments: a Bracket Comment and a Line Comment.

Bracket Comment

A `#` immediately followed by a Bracket Argument forms a *bracket comment* consisting of the entire bracket enclosure:

```
bracket_comment ::=  '#' bracket_argument
```

For example:

```
#[[This is a bracket comment.
It runs until the close bracket.]]
message("First Argument\n" #[[Bracket Comment]] "Second Argument")
```

Note

CMake versions prior to 3.0 do not support bracket comments. They interpret the opening `#` as the start of a Line Comment.

Line Comment

A `#` not immediately followed by a Bracket Argument forms a *line comment* that runs until the end of the line:

```
line_comment ::=  '#' <any text not starting in a bracket_argument
                       and not containing a newline>
```

For example:

```
# This is a line comment.
message("First Argument\n" # This is a line comment :)
        "Second Argument") # This is a line comment.
```

Control Structures

Conditional Blocks

The `if()` / `elseif()` / `else()` / `endif()` commands delimit code blocks to be executed conditionally.

Loops

The `foreach()` / `endforeach()` and `while()` / `endwhile()` commands delimit code blocks to be executed in a loop. Inside such blocks the `break()` command may be used to terminate the loop early whereas the `continue()` command may be used to start with the next iteration immediately.

Command Definitions

The `macro()` / `endmacro()`, and `function()` / `endfunction()` commands delimit code blocks to be recorded for later invocation as commands.

Variables

Variables are the basic unit of storage in the CMake Language. Their values are always of string type, though some commands may interpret the strings as values of other types. The `set()` and `unset()` commands explicitly set or unset a variable, but other commands have semantics that modify variables as well. Variable names are case-sensitive and may consist of almost any text, but we recommend sticking to names consisting only of alphanumeric characters plus `_` and `-`.

Variables have dynamic scope. Each variable "set" or "unset" creates a binding in the current scope:

- Function Scope

  Command Definitions created by the `function()` command create commands that, when invoked, process the recorded commands in a new variable binding scope. A variable "set" or "unset" binds in this scope and is visible for the current function and any nested calls within it, but not after the function returns.

- Directory Scope

  Each of the Directories in a source tree has its own variable bindings. Before processing the `CMakeLists.txt` file for a directory, CMake copies all variable bindings currently defined in the parent directory, if any, to initialize the new directory scope. CMake Scripts, when processed with `cmake -P`, bind variables in one "directory" scope.A variable "set" or "unset" not inside a function call binds to the current directory scope.

- Persistent Cache

  CMake stores a separate set of "cache" variables, or "cache entries", whose values persist across multiple runs within a project build tree. Cache entries have an isolated binding scope modified only by explicit request, such as by the `CACHE` option of the `set()` and `unset()` commands.

When evaluating Variable References, CMake first searches the function call stack, if any, for a binding and then falls back to the binding in the current directory scope, if any. If a "set" binding is found, its value is used. If an "unset" binding is found, or no binding is found, CMake then searches for a cache entry. If a cache entry is found, its value is used. Otherwise, the variable reference evaluates to an empty string. The `$CACHE{VAR}` syntax can be used to do direct cache entry lookups.

The `cmake-variables(7)` manual documents many variables that are provided by CMake or have meaning to CMake when set by project code.

Lists

Although all values in CMake are stored as strings, a string may be treated as a list in certain contexts, such as during evaluation of an Unquoted Argument. In such contexts, a string is divided into list elements by splitting on `;` characters not following an unequal number of `[` and `]` characters and not immediately preceded by a `\`. The sequence `\;` does not divide a value but is replaced by `;` in the resulting element.

A list of elements is represented as a string by concatenating the elements separated by `;`. For example, the `set()` command stores multiple values into the destination variable as a list:

```
set(srcs a.c b.c c.c) # sets "srcs" to "a.c;b.c;c.c"
```

Lists are meant for simple use cases such as a list of source files and should not be used for complex data processing tasks. Most commands that construct lists do not escape `;` characters in list elements, thus flattening nested lists:

```
set(x a "b;c") # sets "x" to "a;b;c", not "a;b\;c"
```

==========

cmake-packages(7)

Contents

Introduction

Packages provide dependency information to CMake based buildsystems. Packages are found with the `find_package()` command. The result of using `find_package` is either a set of `IMPORTED` targets, or a set of variables corresponding to build-relevant information.

Using Packages

CMake provides direct support for two forms of packages, Config-file Packages and Find-module Packages. Indirect support for `pkg-config` packages is also provided via the `FindPkgConfig` module. In all cases, the basic form of `find_package()` calls is the same:

```
find_package(Qt4 4.7.0 REQUIRED) # CMake provides a Qt4 find-module
find_package(Qt5Core 5.1.0 REQUIRED) # Qt provides a Qt5 package config file.
find_package(LibXml2 REQUIRED) # Use pkg-config via the LibXml2 find-module
```

In cases where it is known that a package configuration file is provided by upstream, and only that should be used, the `CONFIG` keyword may be passed to `find_package()`:

```
find_package(Qt5Core 5.1.0 CONFIG REQUIRED)
find_package(Qt5Gui 5.1.0 CONFIG)
```

Similarly, the `MODULE` keyword says to use only a find-module:

```
find_package(Qt4 4.7.0 MODULE REQUIRED)
```

Specifying the type of package explicitly improves the error message shown to the user if it is not found.

Both types of packages also support specifying components of a package, either after the `REQUIRED` keyword:

```
find_package(Qt5 5.1.0 CONFIG REQUIRED Widgets Xml Sql)
```

or as a separate `COMPONENTS` list:

```
find_package(Qt5 5.1.0 COMPONENTS Widgets Xml Sql)
```

or as a separate `OPTIONAL_COMPONENTS` list:

```
find_package(Qt5 5.1.0 COMPONENTS Widgets
                       OPTIONAL_COMPONENTS Xml Sql
)
```

Handling of `COMPONENTS` and `OPTIONAL_COMPONENTS` is defined by the package.

By setting the `CMAKE_DISABLE_FIND_PACKAGE_` variable to `TRUE`, the `<PackageName>` package will not be searched, and will always be `NOTFOUND`.

Config-file Packages

A config-file package is a set of files provided by upstreams for downstreams to use. CMake searches in a number of locations for package configuration files, as described in the `find_package()` documentation. The most simple way for a CMake user to tell `cmake(1)` to search in a non-standard prefix for a package is to set the `CMAKE_PREFIX_PATH` cache variable.

Config-file packages are provided by upstream vendors as part of development packages, that is, they belong with the header files and any other files provided to assist downstreams in using the package.

A set of variables which provide package status information are also set automatically when using a config-file package. The `<PackageName>_FOUND` variable is set to true or false, depending on whether the package was found. The `<PackageName>_DIR` cache variable is set to the location of the package configuration file.

Find-module Packages

A find module is a file with a set of rules for finding the required pieces of a dependency, primarily header files and libraries. Typically, a find module is needed when the upstream is not built with CMake, or is not CMake-aware enough to otherwise provide a package configuration file. Unlike a package configuration file, it is not shipped with upstream, but is used by downstream to find the files by guessing locations of files with platform-specific hints.

Unlike the case of an upstream-provided package configuration file, no single point of reference identifies the package as being found, so the `<PackageName>_FOUND` variable is not automatically set by the `find_package()` command. It can still be expected to be set by convention however and should be set by the author of the Find-module. Similarly there is no `<PackageName>_DIR` variable, but each of the artifacts such as library locations and header file locations provide a separate cache variable.

See the `cmake-developer(7)` manual for more information about creating Find-module files.

Package Layout

A config-file package consists of a Package Configuration File and optionally a Package Version File provided with the project distribution.

Package Configuration File

Consider a project `Foo` that installs the following files:

```
<prefix>/include/foo-1.2/foo.h
<prefix>/lib/foo-1.2/libfoo.a
```

It may also provide a CMake package configuration file:

```
<prefix>/lib/cmake/foo-1.2/FooConfig.cmake
```

with content defining `IMPORTED` targets, or defining variables, such as:

```
# ...
# (compute PREFIX relative to file location)
# ...
set(Foo_INCLUDE_DIRS ${PREFIX}/include/foo-1.2)
set(Foo_LIBRARIES ${PREFIX}/lib/foo-1.2/libfoo.a)
```

If another project wishes to use `Foo` it need only to locate the `FooConfig.cmake` file and load it to get all the information it needs about package content locations. Since the package configuration file is provided by the package installation it already knows all the file locations.

The `find_package()` command may be used to search for the package configuration file. This command constructs a set of installation prefixes and searches under each prefix in several locations. Given the name `Foo`, it looks for a file called `FooConfig.cmake` or `foo-config.cmake`. The full set of locations is specified in the `find_package()` command documentation. One place it looks is:

```
<prefix>/lib/cmake/Foo*/
```

where `Foo*` is a case-insensitive globbing expression. In our example the globbing expression will match `<prefix>/lib/cmake/foo-1.2` and the package configuration file will be found.

Once found, a package configuration file is immediately loaded. It, together with a package version file, contains all the information the project needs to use the package.

Package Version File

When the `find_package()` command finds a candidate package configuration file it looks next to it for a version file. The version file is loaded to test whether the package version is an acceptable match for the version requested. If the version file claims compatibility the configuration file is accepted. Otherwise it is ignored.

The name of the package version file must match that of the package configuration file but has either `-version` or `Version` appended to the name before the `.cmake` extension. For example, the files:

```
<prefix>/lib/cmake/foo-1.3/foo-config.cmake
<prefix>/lib/cmake/foo-1.3/foo-config-version.cmake
```

and:

```
<prefix>/lib/cmake/bar-4.2/BarConfig.cmake
<prefix>/lib/cmake/bar-4.2/BarConfigVersion.cmake
```

are each pairs of package configuration files and corresponding package version files.

When the `find_package()` command loads a version file it first sets the following variables:

- `PACKAGE_FIND_NAME`

  The `<PackageName>`
- `PACKAGE_FIND_VERSION`

  Full requested version string
- `PACKAGE_FIND_VERSION_MAJOR`

  Major version if requested, else 0
- `PACKAGE_FIND_VERSION_MINOR`

  Minor version if requested, else 0
- `PACKAGE_FIND_VERSION_PATCH`

  Patch version if requested, else 0
- `PACKAGE_FIND_VERSION_TWEAK`

Tweak version if requested, else 0

- `PACKAGE_FIND_VERSION_COUNT`

  Number of version components, 0 to 4

The version file must use these variables to check whether it is compatible or an exact match for the requested version and set the following variables with results:

- `PACKAGE_VERSION`

  Full provided version string

- `PACKAGE_VERSION_EXACT`

  True if version is exact match

- `PACKAGE_VERSION_COMPATIBLE`

  True if version is compatible

- `PACKAGE_VERSION_UNSUITABLE`

  True if unsuitable as any version

Version files are loaded in a nested scope so they are free to set any variables they wish as part of their computation. The find_package command wipes out the scope when the version file has completed and it has checked the output variables. When the version file claims to be an acceptable match for the requested version the find_package command sets the following variables for use by the project:

- `<PackageName>_VERSION`

  Full provided version string

- `<PackageName>_VERSION_MAJOR`

  Major version if provided, else 0

- `<PackageName>_VERSION_MINOR`

  Minor version if provided, else 0

- `<PackageName>_VERSION_PATCH`

  Patch version if provided, else 0

- `<PackageName>_VERSION_TWEAK`

  Tweak version if provided, else 0

- `<PackageName>_VERSION_COUNT`

  Number of version components, 0 to 4

The variables report the version of the package that was actually found. The `<PackageName>` part of their name matches the argument given to the `find_package()` command.

Creating Packages

Usually, the upstream depends on CMake itself and can use some CMake facilities for creating the package files. Consider an upstream which provides a single shared library:

```
project(UpstreamLib)

set(CMAKE_INCLUDE_CURRENT_DIR ON)
set(CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE ON)

set(Upstream_VERSION 3.4.1)

include(GenerateExportHeader)

add_library(ClimbingStats SHARED climbingstats.cpp)
generate_export_header(ClimbingStats)
set_property(TARGET ClimbingStats PROPERTY VERSION ${Upstream_VERSION})
set_property(TARGET ClimbingStats PROPERTY SOVERSION 3)
set_property(TARGET ClimbingStats PROPERTY
  INTERFACE_ClimbingStats_MAJOR_VERSION 3)
set_property(TARGET ClimbingStats APPEND PROPERTY
  COMPATIBLE_INTERFACE_STRING ClimbingStats_MAJOR_VERSION
)

install(TARGETS ClimbingStats EXPORT ClimbingStatsTargets
  LIBRARY DESTINATION lib
  ARCHIVE DESTINATION lib
  RUNTIME DESTINATION bin
  INCLUDES DESTINATION include
)
install(
  FILES
    climbingstats.h
    "${CMAKE_CURRENT_BINARY_DIR}/climbingstats_export.h"
  DESTINATION
    include
  COMPONENT
    Devel
)

include(CMakePackageConfigHelpers)
write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/ClimbingStats/ClimbingStatsConfigVersion.cmake"
  VERSION ${Upstream_VERSION}
  COMPATIBILITY AnyNewerVersion
)

export(EXPORT ClimbingStatsTargets
  FILE "${CMAKE_CURRENT_BINARY_DIR}/ClimbingStats/ClimbingStatsTargets.cmake"
  NAMESPACE Upstream::
)
configure_file(cmake/ClimbingStatsConfig.cmake
  "${CMAKE_CURRENT_BINARY_DIR}/ClimbingStats/ClimbingStatsConfig.cmake"
  COPYONLY
)

set(ConfigPackageLocation lib/cmake/ClimbingStats)
install(EXPORT ClimbingStatsTargets
```

```
    FILE
      ClimbingStatsTargets.cmake
    NAMESPACE
      Upstream::
    DESTINATION
      ${ConfigPackageLocation}
  )
  install(
    FILES
      cmake/ClimbingStatsConfig.cmake
      "${CMAKE_CURRENT_BINARY_DIR}/ClimbingStats/ClimbingStatsConfigVersion.cmake"
    DESTINATION
      ${ConfigPackageLocation}
    COMPONENT
      Devel
  )
```

The `CMakePackageConfigHelpers` module provides a macro for creating a simple `ConfigVersion.cmake` file. This file sets the version of the package. It is read by CMake when `find_package()` is called to determine the compatibility with the requested version, and to set some version-specific variables `<PackageName>_VERSION`, `<PackageName>_VERSION_MAJOR`, `<PackageName>_VERSION_MINOR` etc. The `install(EXPORT)` command is used to export the targets in the `ClimbingStatsTargets` export-set, defined previously by the `install(TARGETS)` command. This command generates the `ClimbingStatsTargets.cmake` file to contain `IMPORTED` targets, suitable for use by downstreams and arranges to install it to `lib/cmake/ClimbingStats`. The generated `ClimbingStatsConfigVersion.cmake` and a `cmake/ClimbingStatsConfig.cmake` are installed to the same location, completing the package.

The generated `IMPORTED` targets have appropriate properties set to define their usage requirements, such as `INTERFACE_INCLUDE_DIRECTORIES`, `INTERFACE_COMPILE_DEFINITIONS` and other relevant built-in `INTERFACE_` properties. The `INTERFACE` variant of user-defined properties listed in `COMPATIBLE_INTERFACE_STRING` and other Compatible Interface Properties are also propagated to the generated `IMPORTED` targets. In the above case, `ClimbingStats_MAJOR_VERSION` is defined as a string which must be compatible among the dependencies of any depender. By setting this custom defined user property in this version and in the next version of `ClimbingStats`, `cmake(1)` will issue a diagnostic if there is an attempt to use version 3 together with version 4. Packages can choose to employ such a pattern if different major versions of the package are designed to be incompatible.

A `NAMESPACE` with double-colons is specified when exporting the targets for installation. This convention of double-colons gives CMake a hint that the name is an `IMPORTED` target when it is used by downstreams with the `target_link_libraries()` command. This way, CMake can issue a diagnostic if the package providing it has not yet been found.

In this case, when using `install(TARGETS)` the `INCLUDES DESTINATION` was specified. This causes the `IMPORTED` targets to have their `INTERFACE_INCLUDE_DIRECTORIES` populated with the `include` directory in the `CMAKE_INSTALL_PREFIX`. When the `IMPORTED` target is used by downstream, it automatically consumes the entries from that property.

Creating a Package Configuration File

In this case, the `ClimbingStatsConfig.cmake` file could be as simple as:

```
include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStatsTargets.cmake")
```

As this allows downstreams to use the `IMPORTED` targets. If any macros should be provided by the `ClimbingStats` package, they should be in a separate file which is installed to the same location as the `ClimbingStatsConfig.cmake` file, and included from there.

This can also be extended to cover dependencies:

```
# ...
add_library(ClimbingStats SHARED climbingstats.cpp)
generate_export_header(ClimbingStats)

find_package(Stats 2.6.4 REQUIRED)
target_link_libraries(ClimbingStats PUBLIC Stats::Types)
```

As the `Stats::Types` target is a `PUBLIC` dependency of `ClimbingStats`, downstreams must also find the `Stats` package and link to the `Stats::Types` library. The `Stats` package should be found in the `ClimbingStatsConfig.cmake` file to ensure this. The `find_dependency` macro from the `CMakeFindDependencyMacro` helps with this by propagating whether the package is `REQUIRED`, or `QUIET` etc. All `REQUIRED` dependencies of a package should be found in the `Config.cmake` file:

```
include(CMakeFindDependencyMacro)
find_dependency(Stats 2.6.4)

include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStatsTargets.cmake")
include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStatsMacros.cmake")
```

The `find_dependency` macro also sets `ClimbingStats_FOUND` to `False` if the dependency is not found, along with a diagnostic that the `ClimbingStats` package can not be used without the `Stats` package.

If `COMPONENTS` are specified when the downstream uses `find_package()`, they are listed in the `<PackageName>_FIND_COMPONENTS` variable. If a particular component is non-optional, then the `<PackageName>_FIND_REQUIRED_<comp>` will be true. This can be tested with logic in the package configuration file:

```
include(CMakeFindDependencyMacro)
find_dependency(Stats 2.6.4)

include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStatsTargets.cmake")
include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStatsMacros.cmake")

set(_supported_components Plot Table)

foreach(_comp ${ClimbingStats_FIND_COMPONENTS})
  if (NOT ";${_supported_components};" MATCHES _comp)
    set(ClimbingStats_FOUND False)
    set(ClimbingStats_NOT_FOUND_MESSAGE "Unsupported component: ${_comp}")
  endif()
  include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStats${_comp}Targets.cmake")
endforeach()
```

Here, the `ClimbingStats_NOT_FOUND_MESSAGE` is set to a diagnosis that the package could not be found because an invalid component was specified. This message variable can be set for any case where the `_FOUND` variable is set to `False`, and will be displayed to the user.

Creating a Package Configuration File for the Build Tree

The `export(EXPORT)` command creates an `IMPORTED` targets definition file which is specific to the build-tree, and is not relocatable. This can similarly be used with a suitable package configuration file and package version file to define a package for the build tree which may be used without installation. Consumers of the build tree can simply ensure that the `CMAKE_PREFIX_PATH` contains the build directory, or set the `ClimbingStats_DIR` to `<build_dir>/ClimbingStats` in the cache.

Creating Relocatable Packages

A relocatable package must not reference absolute paths of files on the machine where the package is built that will not exist on the machines where the package may be installed.

Packages created by `install(EXPORT)` are designed to be relocatable, using paths relative to the location of the package itself. When defining the interface of a target for `EXPORT`, keep in mind that the include directories should be specified as relative paths which are relative to the `CMAKE_INSTALL_PREFIX`:

```
target_include_directories(tgt INTERFACE
  # Wrong, not relocatable:
  $<INSTALL_INTERFACE:${CMAKE_INSTALL_PREFIX}/include/TgtName>
)

target_include_directories(tgt INTERFACE
  # Ok, relocatable:
  $<INSTALL_INTERFACE:include/TgtName>
)
```

The `$<INSTALL_PREFIX>` `generator expression` may be used as a placeholder for the install prefix without resulting in a non-relocatable package. This is necessary if complex generator expressions are used:

```
target_include_directories(tgt INTERFACE
  # Ok, relocatable:
  $<INSTALL_INTERFACE:$<$<CONFIG:Debug>:$<INSTALL_PREFIX>/include/TgtName>>
)
```

This also applies to paths referencing external dependencies. It is not advisable to populate any properties which may contain paths, such as `INTERFACE_INCLUDE_DIRECTORIES` and `INTERFACE_LINK_LIBRARIES`, with paths relevant to dependencies. For example, this code may not work well for a relocatable package:

```
target_link_libraries(ClimbingStats INTERFACE
  ${Foo_LIBRARIES} ${Bar_LIBRARIES}
  )
target_include_directories(ClimbingStats INTERFACE
  "$<INSTALL_INTERFACE:${Foo_INCLUDE_DIRS};${Bar_INCLUDE_DIRS}>"
  )
```

The referenced variables may contain the absolute paths to libraries and include directories **as found on the machine the package was made on**. This would create a package with hard-coded paths to dependencies and not suitable for relocation.

Ideally such dependencies should be used through their own IMPORTED targets that have their own `IMPORTED_LOCATION` and usage requirement properties such as `INTERFACE_INCLUDE_DIRECTORIES` populated appropriately. Those imported targets may then be used with the `target_link_libraries()` command for `ClimbingStats`:

```
target_link_libraries(ClimbingStats INTERFACE Foo::Foo Bar::Bar)
```

With this approach the package references its external dependencies only through the names of IMPORTED targets. When a consumer uses the installed package, the consumer will run the appropriate `find_package()` commands (via the `find_dependency` macro described above) to find the dependencies and populate the imported targets with appropriate paths on their own machine.

Unfortunately many `modules` shipped with CMake do not yet provide IMPORTED targets because their development pre-dated this approach. This may improve incrementally over time. Workarounds to create relocatable packages using such modules include:

- When building the package, specify each `Foo_LIBRARY` cache entry as just a library name, e.g. `-DFoo_LIBRARY=foo`. This tells the corresponding find module to populate the `Foo_LIBRARIES` with just `foo` to ask the linker to search for the library instead of hard-coding a path.
- Or, after installing the package content but before creating the package installation binary for redistribution, manually replace the absolute paths with placeholders for substitution by the installation tool when the package is installed.

Package Registry

CMake provides two central locations to register packages that have been built or installed anywhere on a system:

- User Package Registry
- System Package Registry

The registries are especially useful to help projects find packages in non-standard install locations or directly in their own build trees. A project may populate either the user or system registry (using its own means, see below) to refer to its location. In either case the package should store at the registered location a Package Configuration File (`<PackageName>Config.cmake`) and optionally a Package Version File (`<PackageName>ConfigVersion.cmake`).

The `find_package()` command searches the two package registries as two of the search steps specified in its documentation. If it has sufficient permissions it also removes stale package registry entries that refer to directories that do not exist or do not contain a matching package configuration file.

User Package Registry

The User Package Registry is stored in a per-user location. The `export(PACKAGE)` command may be used to register a project build tree in the user package registry. CMake currently provides no interface to add install trees to the user package registry. Installers must be manually taught to register their packages if desired.

On Windows the user package registry is stored in the Windows registry under a key in `HKEY_CURRENT_USER`.

A `<PackageName>` may appear under registry key:

```
HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\<PackageName>
```

as a `REG_SZ` value, with arbitrary name, that specifies the directory containing the package configuration file.

On UNIX platforms the user package registry is stored in the user home directory under `~/.cmake/packages`. A `<PackageName>` may appear under the directory:

```
~/.cmake/packages/<PackageName>
```

as a file, with arbitrary name, whose content specifies the directory containing the package configuration file.

System Package Registry

The System Package Registry is stored in a system-wide location. CMake currently provides no interface to add to the system package registry. Installers must be manually taught to register their packages if desired.

On Windows the system package registry is stored in the Windows registry under a key in `HKEY_LOCAL_MACHINE`. A `<PackageName>` may appear under registry key:

```
HKEY_LOCAL_MACHINE\Software\Kitware\CMake\Packages\<PackageName>
```

as a `REG_SZ` value, with arbitrary name, that specifies the directory containing the package configuration file.

There is no system package registry on non-Windows platforms.

Disabling the Package Registry

In some cases using the Package Registries is not desirable. CMake allows one to disable them using the following variables:

> - `CMAKE_EXPORT_NO_PACKAGE_REGISTRY` disables the `export(PACKAGE)` command.

> - `CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY` disables the User Package Registry in all the `find_package()` calls.
> - `CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY` disables the System Package Registry in all the `find_package()` calls.

Package Registry Example

A simple convention for naming package registry entries is to use content hashes. They are deterministic and unlikely to collide (`export(PACKAGE)` uses this approach). The name of an entry referencing a specific directory is simply the content hash of the directory path itself.

If a project arranges for package registry entries to exist, such as:

```
> reg query HKCU\Software\Kitware\CMake\Packages\MyPackage
HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\MyPackage
 45e7d55f13b87179bb12f907c8de6fc4 REG_SZ c:/Users/Me/Work/lib/cmake/MyPackage
 7b4a9844f681c80ce93190d4e3185db9 REG_SZ c:/Users/Me/Work/MyPackage-build
```

or:

```
$ cat ~/.cmake/packages/MyPackage/7d1fb77e07ce59a81bed093bbee945bd
/home/me/work/lib/cmake/MyPackage
$ cat ~/.cmake/packages/MyPackage/f92c1db873a1937f3100706657c63e07
/home/me/work/MyPackage-build
```

then the `CMakeLists.txt` code:

```
find_package(MyPackage)
```

will search the registered locations for package configuration files (`MyPackageConfig.cmake`). The search order among package registry entries for a single package is unspecified and the entry names (hashes in this example) have no meaning. Registered locations may contain package version files (`MyPackageConfigVersion.cmake`) to tell `find_package()` whether a specific location is suitable for the version requested.

Package Registry Ownership

Package registry entries are individually owned by the project installations that they reference. A package installer is responsible for adding its own entry and the corresponding uninstaller is responsible for removing it.

The `export(PACKAGE)` command populates the user package registry with the location of a project build tree. Build trees tend to be deleted by developers and have no “uninstall” event that could trigger removal of their entries. In order to keep the registries clean the `find_package()` command automatically removes stale entries it encounters if it has sufficient permissions. CMake provides no interface to remove an entry referencing an existing build tree once `export(PACKAGE)` has been invoked. However, if the project removes its package configuration file from the build tree then the entry referencing the location will be considered stale.

==========

cmake-qt(7)

Contents

Introduction

CMake can find and use Qt 4 and Qt 5 libraries. The Qt 4 libraries are found by the `FindQt4` find-module shipped with CMake, whereas the Qt 5 libraries are found using "Config-file Packages" shipped with Qt 5. See `cmake-packages(7)` for more information about CMake packages, and see the Qt cmake manual for your Qt version.

Qt 4 and Qt 5 may be used together in the same `CMake buildsystem`:

```
cmake_minimum_required(VERSION 3.8.0 FATAL_ERROR)

project(Qt4And5)

set(CMAKE_AUTOMOC ON)

find_package(Qt5 COMPONENTS Widgets DBus REQUIRED)
add_executable(publisher publisher.cpp)
target_link_libraries(publisher Qt5::Widgets Qt5::DBus)

find_package(Qt4 REQUIRED)
add_executable(subscriber subscriber.cpp)
target_link_libraries(subscriber Qt4::QtGui Qt4::QtDBus)
```

A CMake target may not link to both Qt 4 and Qt 5. A diagnostic is issued if this is attempted or results from transitive target dependency evaluation.

Qt Build Tools

Qt relies on some bundled tools for code generation, such as `moc` for meta-object code generation, `uic` for widget layout and population, and `rcc` for virtual filesystem content generation. These tools may be automatically invoked by `cmake(1)` if the appropriate conditions are met. The automatic tool invocation may be used with both Qt 4 and Qt 5.

The tools are executed as part of a synthesized custom target generated by CMake. Target dependencies may be added to that custom target by adding them to the `AUTOGEN_TARGET_DEPENDS` target property.

AUTOMOC

The `AUTOMOC` target property controls whether `cmake(1)` inspects the C++ files in the target to determine if they require `moc` to be run, and to create rules to execute `moc` at the appropriate time.

If a macro from `AUTOMOC_MACRO_NAMES` is found in a header file, `moc` will be run on the file. The result will be put into a file named according to `moc_<basename>.cpp`. If the macro is found in a C++ implementation file, the moc output will be put into a file named according to `<basename>.moc`, following the Qt conventions. The `<basename>.moc` must be included by the user in the C++ implementation file with a preprocessor `#include`.

Included `moc_*.cpp` and `*.moc` files will be generated in the `<AUTOGEN_BUILD_DIR>/include` directory which is automatically added to the target's `INCLUDE_DIRECTORIES`.

- This differs from CMake 3.7 and below; see their documentation for details.
- For `multi configuration generators`, the include directory is `<AUTOGEN_BUILD_DIR>/include_<CONFIG>`.
- See `AUTOGEN_BUILD_DIR`.

Not included `moc_<basename>.cpp` files will be generated in custom folders to avoid name collisions and included in a separate `<AUTOGEN_BUILD_DIR>/mocs_compilation.cpp` file which is compiled into the target.

- See `AUTOGEN_BUILD_DIR`.

The `moc` command line will consume the `COMPILE_DEFINITIONS` and `INCLUDE_DIRECTORIES` target properties from the target it is being invoked for, and for the appropriate build configuration.

The `AUTOMOC` target property may be pre-set for all following targets by setting the `CMAKE_AUTOMOC` variable. The `AUTOMOC_MOC_OPTIONS` target property may be populated to set options to pass to `moc`. The `CMAKE_AUTOMOC_MOC_OPTIONS` variable may be populated to pre-set the options for all following targets.

Additional macro names to search for can be added to `AUTOMOC_MACRO_NAMES`.

Additional `moc` dependency file names can be extracted from source code by using `AUTOMOC_DEPEND_FILTERS`.

Source C++ files can be excluded from `AUTOMOC` processing by enabling `SKIP_AUTOMOC` or the broader `SKIP_AUTOGEN`.

AUTOUIC

The `AUTOUIC` target property controls whether `cmake(1)` inspects the C++ files in the target to determine if they require `uic` to be run, and to create rules to execute `uic` at the appropriate time.

If a preprocessor `#include` directive is found which matches `<path>ui_<basename>.h`, and a `<basename>.ui` file exists, then `uic` will be executed to generate the appropriate file. The `<basename>.ui` file is searched for in the following places

1. `<source_dir>/<basename>.ui`

2. `<source_dir>/<path><basename>.ui`
3. `<AUTOUIC_SEARCH_PATHS>/<basename>.ui`
4. `<AUTOUIC_SEARCH_PATHS>/<path><basename>.ui`

where `<source_dir>` is the directory of the C++ file and `AUTOUIC_SEARCH_PATHS` is a list of additional search paths.

The generated generated `ui_*.h` files are placed in the `<AUTOGEN_BUILD_DIR>/include` directory which is automatically added to the target's `INCLUDE_DIRECTORIES`.

- This differs from CMake 3.7 and below; see their documentation for details.
- For `multi configuration generators`, the include directory is `<AUTOGEN_BUILD_DIR>/include_<CONFIG>`.
- See `AUTOGEN_BUILD_DIR`.

The `AUTOUIC` target property may be pre-set for all following targets by setting the `CMAKE_AUTOUIC` variable. The `AUTOUIC_OPTIONS` target property may be populated to set options to pass to `uic`. The `CMAKE_AUTOUIC_OPTIONS` variable may be populated to pre-set the options for all following targets. The `AUTOUIC_OPTIONS` source file property may be set on the `<basename>.ui` file to set particular options for the file. This overrides options from the `AUTOUIC_OPTIONS` target property.

A target may populate the `INTERFACE_AUTOUIC_OPTIONS` target property with options that should be used when invoking `uic`. This must be consistent with the `AUTOUIC_OPTIONS` target property content of the depender target. The `CMAKE_DEBUG_TARGET_PROPERTIES` variable may be used to track the origin target of such `INTERFACE_AUTOUIC_OPTIONS`. This means that a library which provides an alternative translation system for Qt may specify options which should be used when running `uic`:

```
add_library(KI18n klocalizedstring.cpp)
target_link_libraries(KI18n Qt5::Core)

# KI18n uses the tr2i18n() function instead of tr().  That function is
# declared in the klocalizedstring.h header.
set(autouic_options
  -tr tr2i18n
  -include klocalizedstring.h
)

set_property(TARGET KI18n APPEND PROPERTY
  INTERFACE_AUTOUIC_OPTIONS ${autouic_options}
)
```

A consuming project linking to the target exported from upstream automatically uses appropriate options when `uic` is run by `AUTOUIC`, as a result of linking with the `IMPORTED` target:

```
set(CMAKE_AUTOUIC ON)
# Uses a libwidget.ui file:
add_library(LibWidget libwidget.cpp)
target_link_libraries(LibWidget
  KF5::KI18n
  Qt5::Widgets
)
```

Source files can be excluded from AUTOUIC processing by enabling SKIP_AUTOUIC or the broader SKIP_AUTOGEN.

AUTORCC

The AUTORCC target property controls whether cmake(1) creates rules to execute rcc at the appropriate time on source files which have the suffix .qrc.

```
add_executable(myexe main.cpp resource_file.qrc)
```

The AUTORCC target property may be pre-set for all following targets by setting the CMAKE_AUTORCC variable. The AUTORCC_OPTIONS target property may be populated to set options to pass to rcc. The CMAKE_AUTORCC_OPTIONS variable may be populated to pre-set the options for all following targets. The AUTORCC_OPTIONS source file property may be set on the <name>.qrc file to set particular options for the file. This overrides options from the AUTORCC_OPTIONS target property.

Source files can be excluded from AUTORCC processing by enabling SKIP_AUTORCC or the broader SKIP_AUTOGEN.

Visual Studio Generators

When using the Visual Studio generators, CMake uses a PRE_BUILD custom command for AUTOMOC and AUTOUIC. If the AUTOMOC and AUTOUIC processing depends on files, a custom target is used instead. This happens when

- The origin target depends on GENERATED files which aren't excluded from AUTOMOC and AUTOUIC by SKIP_AUTOMOC, SKIP_AUTOUIC, SKIP_AUTOGEN or CMP0071
- AUTOGEN_TARGET_DEPENDS lists a source file

qtmain.lib on Windows

The Qt 4 and 5 IMPORTED targets for the QtGui libraries specify that the qtmain.lib static library shipped with Qt will be linked by all dependent executables which have the WIN32_EXECUTABLE enabled.

To disable this behavior, enable the Qt5_NO_LINK_QTMAIN target property for Qt 5 based targets or QT4_NO_LINK_QTMAIN target property for Qt 4 based targets.

```
add_executable(myexe WIN32 main.cpp)
target_link_libraries(myexe Qt4::QtGui)

add_executable(myexe_no_qtmain WIN32 main_no_qtmain.cpp)
set_property(TARGET main_no_qtmain PROPERTY QT4_NO_LINK_QTMAIN ON)
target_link_libraries(main_no_qtmain Qt4::QtGui)
```

==========

cmake-server(7)

Contents

Introduction

`cmake(1)` is capable of providing semantic information about CMake code it executes to generate a buildsystem. If executed with the `-E server` command line options, it starts in a long running mode and allows a client to request the available information via a JSON protocol.

The protocol is designed to be useful to IDEs, refactoring tools, and other tools which have a need to understand the buildsystem in entirety.

A single `cmake-buildsystem(7)` may describe buildsystem contents and build properties which differ based on `generation-time context` including:

- The Platform (eg, Windows, APPLE, Linux).
- The build configuration (eg, Debug, Release, Coverage).
- The Compiler (eg, MSVC, GCC, Clang) and compiler version.
- The language of the source files compiled.
- Available compile features (eg CXX variadic templates).
- CMake policies.

The protocol aims to provide information to tooling to satisfy several needs:

1. Provide a complete and easily parsed source of all information relevant to the tooling as it relates to the source code. There should be no need for tooling to parse generated buildsystems to access include directories or compile definitions for example.
2. Semantic information about the CMake buildsystem itself.
3. Provide a stable interface for reading the information in the CMake cache.
4. Information for determining when cmake needs to be re-run as a result of file changes.

Operation

Start `cmake(1)` in the server command mode, supplying the path to the build directory to process:

```
cmake -E server (--debug|--pipe=<NAMED_PIPE>)
```

The server will communicate using stdin/stdout (with the `--debug` parameter) or using a named pipe (with the `--pipe=<NAMED_PIPE>` parameter). Note that "named pipe" refers to a local domain socket on Unix and to a named pipe on Windows.

When connecting to the server (via named pipe or by starting it in `--debug` mode), the server will reply with a hello message:

```
[== "CMake Server" ==[
{"supportedProtocolVersions":[{"major":1,"minor":0}],"type":"hello"}
]== "CMake Server" ==]
```

Messages sent to and from the process are wrapped in magic strings:

```
[== "CMake Server" ==[
{
  ... some JSON message ...
}
]== "CMake Server" ==]
```

The server is now ready to accept further requests via the named pipe or stdin.

Debugging

CMake server mode can be asked to provide statistics on execution times, etc. or to dump a copy of the response into a file. This is done passing a "debug" JSON object as a child of the request.

The debug object supports the “showStats” key, which takes a boolean and makes the server mode return a “zzzDebug” object with stats as part of its response. “dumpToFile” takes a string value and will cause the cmake server to copy the response into the given filename.

This is a response from the cmake server with “showStats” set to true:

```
[== "CMake Server" ==[
{
  "cookie":"",
  "errorMessage":"Waiting for type \"handshake\".",
  "inReplyTo":"unknown",
 "type":"error",
  "zzzDebug": {
    "dumpFile":"/tmp/error.txt",
    "jsonSerialization":0.011016,
    "size":111,
    "totalTime":0.025995
  }
}
]== "CMake Server" ==]
```

The server has made a copy of this response into the file /tmp/error.txt and took 0.011 seconds to turn the JSON response into a string, and it took 0.025 seconds to process the request in total. The reply has a size of 111 bytes.

Protocol API

General Message Layout

All messages need to have a “type” value, which identifies the type of message that is passed back or forth. E.g. the initial message sent by the server is of type “hello”. Messages without a type will generate an response of type “error”.

All requests sent to the server may contain a “cookie” value. This value will he handed back unchanged in all responses triggered by the request.

All responses will contain a value “inReplyTo”, which may be empty in case of parse errors, but will contain the type of the request message in all other cases.

Type “reply”

This type is used by the server to reply to requests.

The message may – depending on the type of the original request – contain values.

Example:

```
[== "CMake Server" ==[
{"cookie":"zimtstern","inReplyTo":"handshake","type":"reply"}
]== "CMake Server" ==]
```

Type “error”

This type is used to return an error condition to the client. It will contain an “errorMessage”.

Example:

```
[== "CMake Server" ==[
{"cookie":"","errorMessage":"Protocol version not
supported.","inReplyTo":"handshake","type":"error"}
]== "CMake Server" ==]
```

Type “progress”

When the server is busy for a long time, it is polite to send back replies of type “progress” to the client. These will contain a “progressMessage” with a string describing the action currently taking place as well as “progressMinimum”, “progressMaximum” and “progressCurrent” with integer values describing the range of progress.

Messages of type “progress” will be followed by more “progress” messages or with a message of type “reply” or “error” that complete the request.

“progress” messages may not be emitted after the “reply” or “error” message for the request that triggered the responses was delivered.

Type “message”

A message is triggered when the server processes a request and produces some form of output that should be displayed to the user. A Message has a “message” with the actual text to display as well as a “title” with a suggested dialog box title.

Example:

```
[== "CMake Server" ==[
{"cookie":"","message":"Something happened.","title":"Title
Text","inReplyTo":"handshake","type":"message"}
]== "CMake Server" ==]
```

Type “signal”

The server can send signals when it detects changes in the system state. Signals are of type “signal”, have an empty “cookie” and “inReplyTo” field and always have a “name” set to show which signal was sent.

Specific Signals

The cmake server may sent signals with the following names:

“dirty” Signal

The “dirty” signal is sent whenever the server determines that the configuration of the project is no longer up-to-date. This happens when any of the files that have an influence on the build system is changed.

The “dirty” signal may look like this:

```
[== "CMake Server" ==[
{
  "cookie":"",
  "inReplyTo":"",
  "name":"dirty",
  "type":"signal"}
]== "CMake Server" ==]
```

"fileChange" Signal

The "fileChange" signal is sent whenever a watched file is changed. It contains the "path" that has changed and a list of "properties" with the kind of change that was detected. Possible changes are "change" and "rename".

The "fileChange" signal looks like this:

```
[== "CMake Server" ==[
{
  "cookie":"",
  "inReplyTo":"",
  "name":"fileChange",
  "path":"/absolute/CMakeLists.txt",
  "properties":["change"],
  "type":"signal"}
]== "CMake Server" ==]
```

Specific Message Types

Type "hello"

The initial message send by the cmake server on startup is of type "hello". This is the only message ever sent by the server that is not of type "reply", "progress" or "error".

It will contain "supportedProtocolVersions" with an array of server protocol versions supported by the cmake server. These are JSON objects with "major" and "minor" keys containing non-negative integer values. Some versions may be marked as experimental. These will contain the "isExperimental" key set to true. Enabling these requires a special command line argument when starting the cmake server mode.

Within a "major" version all "minor" versions are fully backwards compatible. New "minor" versions may introduce functionality in such a way that existing clients of the same "major" version will continue to work, provided they ignore keys in the output that they do not know about.

Example:

```
[== "CMake Server" ==[
{"supportedProtocolVersions":[{"major":0,"minor":1}],"type":"hello"}
]== "CMake Server" ==]
```

Type "handshake"

The first request that the client may send to the server is of type "handshake".

This request needs to pass one of the “supportedProtocolVersions” of the “hello” type response received earlier back to the server in the “protocolVersion” field. Giving the “major” version of the requested protocol version will make the server use the latest minor version of that protocol. Use this if you do not explicitly need to depend on a specific minor version.

Protocol version 1.0 requires the following attributes to be set:

> - “sourceDirectory” with a path to the sources
> - “buildDirectory” with a path to the build directory
> - “generator” with the generator name
> - “extraGenerator” (optional!) with the extra generator to be used
> - “platform” with the generator platform (if supported by the generator)
> - “toolset” with the generator toolset (if supported by the generator)

Protocol version 1.2 makes all but the build directory optional, provided there is a valid cache in the build directory that contains all the other information already.

Example:

```
[== "CMake Server" ==[
{"cookie":"zimtstern","type":"handshake","protocolVersion":{"major":0},
 "sourceDirectory":"/home/code/cmake", "buildDirectory":"/tmp/testbuild",
 "generator":"Ninja"}
]== "CMake Server" ==]
```

which will result in a response type “reply”:

```
[== "CMake Server" ==[
{"cookie":"zimtstern","inReplyTo":"handshake","type":"reply"}
]== "CMake Server" ==]
```

indicating that the server is ready for action.

Type “globalSettings”

This request can be sent after the initial handshake. It will return a JSON structure with information on cmake state.

Example:

```
[== "CMake Server" ==[
{"type":"globalSettings"}
]== "CMake Server" ==]
```

which will result in a response type “reply”:

```
[== "CMake Server" ==[
{
  "buildDirectory": "/tmp/test-build",
  "capabilities": {
    "generators": [
      {
```

```
          "extraGenerators": [],
          "name": "Watcom WMake",
          "platformSupport": false,
          "toolsetSupport": false
        },
        <...>
      ],
      "serverMode": false,
      "version": {
        "isDirty": false,
        "major": 3,
        "minor": 6,
        "patch": 20160830,
        "string": "3.6.20160830-gd6abad",
        "suffix": "gd6abad"
      }
    },
    "checkSystemVars": false,
    "cookie": "",
    "extraGenerator": "",
    "generator": "Ninja",
    "debugOutput": false,
    "inReplyTo": "globalSettings",
    "sourceDirectory": "/home/code/cmake",
    "trace": false,
    "traceExpand": false,
    "type": "reply",
    "warnUninitialized": false,
    "warnUnused": false,
    "warnUnusedCli": true
  }
  ]== "CMake Server" ==]
```

Type “setGlobalSettings”

This request can be sent to change the global settings attributes. Unknown attributes are going to be ignored. Read-only attributes reported by “globalSettings” are all capabilities, buildDirectory, generator, extraGenerator and sourceDirectory. Any attempt to set these will be ignored, too.

All other settings will be changed.

The server will respond with an empty reply message or an error.

Example:

```
[== "CMake Server" ==[
{"type":"setGlobalSettings","debugOutput":true}
]== "CMake Server" ==]
```

CMake will reply to this with:

```
[== "CMake Server" ==[
{"inReplyTo":"setGlobalSettings","type":"reply"}
]== "CMake Server" ==]
```

Type “configure”

This request will configure a project for build.

To configure a build directory already containing cmake files, it is enough to set “buildDirectory” via “setGlobalSettings”. To create a fresh build directory you also need to set “currentGenerator” and “sourceDirectory” via “setGlobalSettings” in addition to “buildDirectory”.

You may a list of strings to “configure” via the “cacheArguments” key. These strings will be interpreted similar to command line arguments related to cache handling that are passed to the cmake command line client.

Example:

```
[== "CMake Server" ==[
{"type":"configure", "cacheArguments":["-Dsomething=else"]}
]== "CMake Server" ==]
```

CMake will reply like this (after reporting progress for some time):

```
[== "CMake Server" ==[
{"cookie":"","inReplyTo":"configure","type":"reply"}
]== "CMake Server" ==]
```

Type “compute”

This request will generate build system files in the build directory and is only available after a project was successfully “configure”d.

Example:

```
[== "CMake Server" ==[
{"type":"compute"}
]== "CMake Server" ==]
```

CMake will reply (after reporting progress information):

```
[== "CMake Server" ==[
{"cookie":"","inReplyTo":"compute","type":"reply"}
]== "CMake Server" ==]
```

Type “codemodel”

The “codemodel” request can be used after a project was “compute”d successfully.

It will list the complete project structure as it is known to cmake.

The reply will contain a key “configurations”, which will contain a list of configuration objects. Configuration objects are used to destinquish between different configurations the build directory might have enabled. While most generators only support one configuration, others might support several.

Each configuration object can have the following keys:

- “name”

  contains the name of the configuration. The name may be empty.
- “projects”

  contains a list of project objects, one for each build project.

Project objects define one (sub-)project defined in the cmake build system.

Each project object can have the following keys:

- “name”

  contains the (sub-)projects name.
- “minimumCMakeVersion”

  contains the minimum cmake version allowed for this project, null if the project doesn’t specify one.
- “hasInstallRule”

  true if the project contains any install rules, false otherwise.
- “sourceDirectory”

  contains the current source directory
- “buildDirectory”

  contains the current build directory.
- “targets”

  contains a list of build system target objects.

Target objects define individual build targets for a certain configuration.

Each target object can have the following keys:

- “name”

  contains the name of the target.
- “type”

  defines the type of build of the target. Possible values are “STATIC_LIBRARY”, “MODULE_LIBRARY”, “SHARED_LIBRARY”, “OBJECT_LIBRARY”, “EXECUTABLE”, “UTILITY” and “INTERFACE_LIBRARY”.
- “fullName”

  contains the full name of the build result (incl. extensions, etc.).
- “sourceDirectory”

  contains the current source directory.
- “buildDirectory”

  contains the current build directory.
- “isGeneratorProvided”

true if the target is auto-created by a generator, false otherwise

- “hasInstallRule”

  true if the target contains any install rules, false otherwise.

- “installPaths”

  full path to the destination directories defined by target install rules.

- “artifacts”

  with a list of build artifacts. The list is sorted with the most important artifacts first (e.g. a .DLL file is listed before a .PDB file on windows).

- “linkerLanguage”

  contains the language of the linker used to produce the artifact.

- “linkLibraries”

  with a list of libraries to link to. This value is encoded in the system’s native shell format.

- “linkFlags”

  with a list of flags to pass to the linker. This value is encoded in the system’s native shell format.

- “linkLanguageFlags”

  with the flags for a compiler using the linkerLanguage. This value is encoded in the system’s native shell format.

- “frameworkPath”

  with the framework path (on Apple computers). This value is encoded in the system’s native shell format.

- “linkPath”

  with the link path. This value is encoded in the system’s native shell format.

- “sysroot”

  with the sysroot path.

- “fileGroups”

  contains the source files making up the target.

FileGroups are used to group sources using similar settings together.

Each fileGroup object may contain the following keys:

- “language”

  contains the programming language used by all files in the group.

- “compileFlags”

  with a string containing all the flags passed to the compiler when building any of the files in this group. This value is encoded in the system’s native shell format.

- “includePath”

  with a list of include paths. Each include path is an object containing a “path” with the actual include path and “isSystem” with a bool value informing whether this is a normal include or a system include. This value is encoded in the system’s native shell format.

- “defines”

with a list of defines in the form “SOMEVALUE” or “SOMEVALUE=42”. This value is encoded in the system’s native shell format.

- “sources”

  with a list of source files.

All file paths in the fileGroup are either absolute or relative to the sourceDirectory of the target.

Example:

```
[== "CMake Server" ==[
{"type":"codemodel"}
]== "CMake Server" ==]
```

CMake will reply:

```
[== "CMake Server" ==[
{
  "configurations": [
    {
      "name": "",
      "projects": [
        {
          "buildDirectory": "/tmp/build/Source/CursesDialog/form",
          "name": "CMAKE_FORM",
          "sourceDirectory": "/home/code/src/cmake/Source/CursesDialog/form",
          "targets": [
            {
              "artifacts": [ "/tmp/build/Source/CursesDialog/form/libcmForm.a" ],
              "buildDirectory": "/tmp/build/Source/CursesDialog/form",
              "fileGroups": [
                {
                  "compileFlags": "  -std=gnu11",
                  "defines": [ "CURL_STATICLIB", "LIBARCHIVE_STATIC" ],
                  "includePath": [ { "path": "/tmp/build/Utilities" }, <...> ],
                  "isGenerated": false,
                  "language": "C",
                  "sources": [ "fld_arg.c", <...> ]
                }
              ],
              "fullName": "libcmForm.a",
              "linkerLanguage": "C",
              "name": "cmForm",
              "sourceDirectory": "/home/code/src/cmake/Source/CursesDialog/form",
              "type": "STATIC_LIBRARY"
            }
          ]
        },
        <...>
      ]
    }
  ],
  "cookie": "",
```

```
   "inReplyTo": "codemodel",
   "type": "reply"
 }
 ]== "CMake Server" ==]
```

Type “ctestInfo”

The “ctestInfo” request can be used after a project was “compute”d successfully.

It will list the complete project test structure as it is known to cmake.

The reply will contain a key “configurations”, which will contain a list of configuration objects. Configuration objects are used to destinquish between different configurations the build directory might have enabled. While most generators only support one configuration, others might support several.

Each configuration object can have the following keys:

- “name”

  contains the name of the configuration. The name may be empty.
- “projects”

  contains a list of project objects, one for each build project.

Project objects define one (sub-)project defined in the cmake build system.

Each project object can have the following keys:

- “name”

  contains the (sub-)projects name.
- “ctestInfo”

  contains a list of test objects.

Each test object can have the following keys:

- “ctestName”

  contains the name of the test.
- “ctestCommand”

  contains the test command.
- “properties”

  contains a list of test property objects.

Each test property object can have the following keys:

- “key”

  contains the test property key.
- “value”

  contains the test property value.

Type “cmakeInputs”

The “cmakeInputs” requests will report files used by CMake as part of the build system itself.

This request is only available after a project was successfully “configure”d.

Example:

```
[== "CMake Server" ==[
{"type":"cmakeInputs"}
]== "CMake Server" ==]
```

CMake will reply with the following information:

```
[== "CMake Server" ==[
{"buildFiles":
  [
    {"isCMake":true,"isTemporary":false,"sources":["/usr/lib/cmake/...", ... ]},
    {"isCMake":false,"isTemporary":false,"sources":["CMakeLists.txt", ...]},
    {"isCMake":false,"isTemporary":true,"sources":["/tmp/build/CMakeFiles/...", ...]}
  ],
  "cmakeRootDirectory":"/usr/lib/cmake",
  "sourceDirectory":"/home/code/src/cmake",
  "cookie":"",
  "inReplyTo":"cmakeInputs",
  "type":"reply"
}
]== "CMake Server" ==]
```

All file names are either relative to the top level source directory or absolute.

The list of files which “isCMake” set to true are part of the cmake installation.

The list of files witch “isTemporary” set to true are part of the build directory and will not survive the build directory getting cleaned out.

Type “cache”

The “cache” request will list the cached configuration values.

Example:

```
[== "CMake Server" ==[
{"type":"cache"}
]== "CMake Server" ==]
```

CMake will respond with the following output:

```
[== "CMake Server" ==[
{
  "cookie":"","inReplyTo":"cache","type":"reply",
  "cache":
  [
    {
      "key":"SOMEVALUE",
      "properties":
```

```
      {
        "ADVANCED":"1",
        "HELPSTRING":"This is not helpful"
      }
      "type":"STRING",
      "value":"TEST"}
  ]
}
]== "CMake Server" ==]
```

The output can be limited to a list of keys by passing an array of key names to the “keys” optional field of the “cache” request.

Type “fileSystemWatchers”

The server can watch the filesystem for changes. The “fileSystemWatchers” command will report on the files and directories watched.

Example:

```
[== "CMake Server" ==[
{"type":"fileSystemWatchers"}
]== "CMake Server" ==]
```

CMake will respond with the following output:

```
[== "CMake Server" ==[
{
  "cookie":"","inReplyTo":"fileSystemWatchers","type":"reply",
  "watchedFiles": [ "/absolute/path" ],
  "watchedDirectories": [ "/absolute" ]
}
]== "CMake Server" ==]
```

==========

cmake-toolchains(7)

Contents

Introduction

CMake uses a toolchain of utilities to compile, link libraries and create archives, and other tasks to drive the build. The toolchain utilities available are determined by the languages enabled. In normal builds, CMake automatically determines the toolchain for host builds based on system introspection and defaults. In cross-compiling scenarios, a toolchain file may be specified with information about compiler and utility paths.

Languages

Languages are enabled by the `project()` command. Language-specific built-in variables, such as `CMAKE_CXX_COMPILER`, `CMAKE_CXX_COMPILER_ID` etc are set by invoking the `project()` command. If no project command is in the top-level CMakeLists file, one will be implicitly generated. By default the enabled languages are C and CXX:

```
project(C_Only C)
```

A special value of NONE can also be used with the `project()` command to enable no languages:

```
project(MyProject NONE)
```

The `enable_language()` command can be used to enable languages after the `project()` command:

```
enable_language(CXX)
```

When a language is enabled, CMake finds a compiler for that language, and determines some information, such as the vendor and version of the compiler, the target architecture and bitwidth, the location of corresponding utilities etc.

The `ENABLED_LANGUAGES` global property contains the languages which are currently enabled.

Variables and Properties

Several variables relate to the language components of a toolchain which are enabled. `CMAKE_<LANG>_COMPILER` is the full path to the compiler used for `<LANG>`. `CMAKE_<LANG>_COMPILER_ID` is the identifier used by CMake for the compiler and `CMAKE_<LANG>_COMPILER_VERSION` is the version of the compiler.

The `CMAKE_<LANG>_FLAGS` variables and the configuration-specific equivalents contain flags that will be added to the compile command when compiling a file of a particular language.

As the linker is invoked by the compiler driver, CMake needs a way to determine which compiler to use to invoke the linker. This is calculated by the `LANGUAGE` of source files in the target, and in the case of static libraries, the language of the dependent libraries. The choice CMake makes may be overridden with the `LINKER_LANGUAGE` target property.

Toolchain Features

CMake provides the `try_compile()` command and wrapper macros such as `CheckCXXSourceCompiles`, `CheckCXXSymbolExists` and `CheckIncludeFile` to test capability and availability of various toolchain features. These APIs test the toolchain in some way and cache the result so that the test does not have to be performed again the next time CMake runs.

Some toolchain features have built-in handling in CMake, and do not require compile-tests. For example, `POSITION_INDEPENDENT_CODE` allows specifying that a target should be built as position-independent code, if the compiler supports that feature. The `<LANG>_VISIBILITY_PRESET` and `VISIBILITY_INLINES_HIDDEN` target properties add flags for hidden visibility, if supported by the compiler.

Cross Compiling

If `cmake(1)` is invoked with the command line parameter `-DCMAKE_TOOLCHAIN_FILE=path/to/file`, the file will be loaded early to set values for the compilers. The `CMAKE_CROSSCOMPILING` variable is set to true when CMake is cross-compiling.

Cross Compiling for Linux

A typical cross-compiling toolchain for Linux has content such as:

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(CMAKE_SYSROOT /home/devel/rasp-pi-rootfs)
set(CMAKE_STAGING_PREFIX /home/devel/stage)

set(tools /home/devel/gcc-4.7-linaro-rpi-gnueabihf)
set(CMAKE_C_COMPILER ${tools}/bin/arm-linux-gnueabihf-gcc)
set(CMAKE_CXX_COMPILER ${tools}/bin/arm-linux-gnueabihf-g++)

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
```

The `CMAKE_SYSTEM_NAME` is the CMake-identifier of the target platform to build for.

The `CMAKE_SYSTEM_PROCESSOR` is the CMake-identifier of the target architecture to build for.

The `CMAKE_SYSROOT` is optional, and may be specified if a sysroot is available.

The `CMAKE_STAGING_PREFIX` is also optional. It may be used to specify a path on the host to install to. The `CMAKE_INSTALL_PREFIX` is always the runtime installation location, even when cross-compiling.

The `CMAKE__COMPILER` variables may be set to full paths, or to names of compilers to search for in standard locations. For toolchains that do not support linking binaries without custom flags or scripts one may set the `CMAKE_TRY_COMPILE_TARGET_TYPE` variable to `STATIC_LIBRARY` to tell CMake not to try to link executables during its checks.

CMake `find_*` commands will look in the sysroot, and the `CMAKE_FIND_ROOT_PATH` entries by default in all cases, as well as looking in the host system root prefix. Although this can be controlled on a case-by-case basis, when cross-compiling, it can be useful to exclude looking in either the host or the target for particular artifacts. Generally, includes, libraries and packages should be found in the target system prefixes, whereas executables which must be run as part of the build should be found only on the host and not on the target. This is the purpose of the `CMAKE_FIND_ROOT_PATH_MODE_*` variables.

Cross Compiling for the Cray Linux Environment

Cross compiling for compute nodes in the Cray Linux Environment can be done without needing a separate toolchain file. Specifying `-DCMAKE_SYSTEM_NAME=CrayLinuxEnvironment` on the CMake command line will ensure that the appropriate build settings and search paths are configured. The platform will pull its configuration from the current environment variables and will configure a project to use the compiler wrappers from the Cray Programming Environment's `PrgEnv-*` modules if present and loaded.

The default configuration of the Cray Programming Environment is to only support static libraries. This can be overridden and shared libraries enabled by setting the `CRAYPE_LINK_TYPE` environment variable to `dynamic`.

Running CMake without specifying `CMAKE_SYSTEM_NAME` will run the configure step in host mode assuming a standard Linux environment. If not overridden, the `PrgEnv-*` compiler wrappers will end up getting used, which if targeting the either the login node or compute node, is likely not the desired behavior. The exception to this would be if you are building directly on a NID instead of cross-compiling from a login node. If trying to build software for a login node, you will need to either first unload the currently loaded `PrgEnv-*` module or explicitly tell CMake to use the system compilers in `/usr/bin` instead of the Cray wrappers. If instead targeting a compute node is desired, just specify the `CMAKE_SYSTEM_NAME` as mentioned above.

Cross Compiling using Clang

Some compilers such as Clang are inherently cross compilers. The `CMAKE__COMPILER_TARGET` can be set to pass a value to those supported compilers when compiling:

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(triple arm-linux-gnueabihf)

set(CMAKE_C_COMPILER clang)
set(CMAKE_C_COMPILER_TARGET ${triple})
set(CMAKE_CXX_COMPILER clang++)
set(CMAKE_CXX_COMPILER_TARGET ${triple})
```

Similarly, some compilers do not ship their own supplementary utilities such as linkers, but provide a way to specify the location of the external toolchain which will be used by the compiler driver.
The `CMAKE_<LANG>_COMPILER_EXTERNAL_TOOLCHAIN` variable can be set in a toolchain file to pass the path to the compiler driver.

Cross Compiling for QNX

As the Clang compiler the QNX QCC compile is inherently a cross compiler. And the `CMAKE_<LANG>_COMPILER_TARGET` can be set to pass a value to those supported compilers when compiling:

```
set(CMAKE_SYSTEM_NAME QNX)

set(arch gcc_ntoarmv7le)

set(CMAKE_C_COMPILER qcc)
set(CMAKE_C_COMPILER_TARGET ${arch})
set(CMAKE_CXX_COMPILER QCC)
set(CMAKE_CXX_COMPILER_TARGET ${arch})
```

Cross Compiling for Windows CE

Cross compiling for Windows CE requires the corresponding SDK being installed on your system. These SDKs are usually installed under `C:/Program Files (x86)/Windows CE Tools/SDKs`.

A toolchain file to configure a Visual Studio generator for Windows CE may look like this:

```
set(CMAKE_SYSTEM_NAME WindowsCE)

set(CMAKE_SYSTEM_VERSION 8.0)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(CMAKE_GENERATOR_TOOLSET CE800) # Can be omitted for 8.0
set(CMAKE_GENERATOR_PLATFORM SDK_AM335X_SK_WEC2013_V310)
```

The `CMAKE_GENERATOR_PLATFORM` tells the generator which SDK to use. Further `CMAKE_SYSTEM_VERSION` tells the generator what version of Windows CE to use. Currently version 8.0 (Windows Embedded Compact 2013) is supported out of the box. Other versions may require one to set `CMAKE_GENERATOR_TOOLSET` to the correct value.

Cross Compiling for Windows 10 Universal Applications

A toolchain file to configure a Visual Studio generator for a Windows 10 Universal Application may look like this:

```
set(CMAKE_SYSTEM_NAME WindowsStore)
set(CMAKE_SYSTEM_VERSION 10.0)
```

A Windows 10 Universal Application targets both Windows Store and Windows Phone. Specify the `CMAKE_SYSTEM_VERSION` variable to be `10.0` to build with the latest available Windows 10 SDK. Specify a more specific version (e.g. `10.0.10240.0` for RTM) to build with the corresponding SDK.

Cross Compiling for Windows Phone

A toolchain file to configure a Visual Studio generator for Windows Phone may look like this:

```
set(CMAKE_SYSTEM_NAME WindowsPhone)
set(CMAKE_SYSTEM_VERSION 8.1)
```

Cross Compiling for Windows Store

A toolchain file to configure a Visual Studio generator for Windows Store may look like this:

```
set(CMAKE_SYSTEM_NAME WindowsStore)
set(CMAKE_SYSTEM_VERSION 8.1)
```

Cross Compiling for Android

A toolchain file may configure cross-compiling for Android by setting the `CMAKE_SYSTEM_NAME` variable to `Android`. Further configuration is specific to the Android development environment to be used.

For Visual Studio Generators, CMake expects NVIDIA Nsight Tegra Visual Studio Edition to be installed. See that section for further configuration details.

For Makefile Generators and the `Ninja` generator, CMake expects one of these environments:

- NDK
- Standalone Toolchain

CMake uses the following steps to select one of the environments:

- If the `CMAKE_ANDROID_NDK` variable is set, the NDK at the specified location will be used.
- Else, if the `CMAKE_ANDROID_STANDALONE_TOOLCHAIN` variable is set, the Standalone Toolchain at the specified location will be used.
- Else, if the `CMAKE_SYSROOT` variable is set to a directory of the form `<ndk>/platforms/android-<api>/arch-<arch>`, the `<ndk>` part will be used as the value of `CMAKE_ANDROID_NDK` and the NDK will be used.
- Else, if the `CMAKE_SYSROOT` variable is set to a directory of the form `<standalone-toolchain>/sysroot`, the `<standalone-toolchain>` part will be used as the value of `CMAKE_ANDROID_STANDALONE_TOOLCHAIN` and the Standalone Toolchain will be used.

- Else, if a cmake variable `ANDROID_NDK` is set it will be used as the value of `CMAKE_ANDROID_NDK`, and the NDK will be used.
- Else, if a cmake variable `ANDROID_STANDALONE_TOOLCHAIN` is set, it will be used as the value of `CMAKE_ANDROID_STANDALONE_TOOLCHAIN`, and the Standalone Toolchain will be used.
- Else, if an environment variable `ANDROID_NDK_ROOT` or `ANDROID_NDK` is set, it will be used as the value of `CMAKE_ANDROID_NDK`, and the NDK will be used.
- Else, if an environment variable `ANDROID_STANDALONE_TOOLCHAIN` is set then it will be used as the value of `CMAKE_ANDROID_STANDALONE_TOOLCHAIN`, and the Standalone Toolchain will be used.
- Else, an error diagnostic will be issued that neither the NDK or Standalone Toolchain can be found.

Cross Compiling for Android with the NDK

A toolchain file may configure Makefile Generators or the `Ninja` generator to target Android for cross-compiling.

Configure use of an Android NDK with the following variables:

- `CMAKE_SYSTEM_NAME`

  Set to `Android`. Must be specified to enable cross compiling for Android.

- `CMAKE_SYSTEM_VERSION`

  Set to the Android API level. If not specified, the value is determined as follows:If the `CMAKE_ANDROID_API` variable is set, its value is used as the API level.If the `CMAKE_SYSROOT` variable is set, the API level is detected from the NDK directory structure containing the sysroot.Otherwise, the latest API level available in the NDK is used.

- `CMAKE_ANDROID_ARCH_ABI`

  Set to the Android ABI (architecture). If not specified, this variable will default to `armeabi`. The `CMAKE_ANDROID_ARCH` variable will be computed from `CMAKE_ANDROID_ARCH_ABI` automatically. Also see the `CMAKE_ANDROID_ARM_MODE` and `CMAKE_ANDROID_ARM_NEON` variables.

- `CMAKE_ANDROID_NDK`

  Set to the absolute path to the Android NDK root directory. A `${CMAKE_ANDROID_NDK}/platforms` directory must exist. If not specified, a default for this variable will be chosen as specified above.

- `CMAKE_ANDROID_NDK_DEPRECATED_HEADERS`

  Set to a true value to use the deprecated per-api-level headers instead of the unified headers. If not specified, the default will be false unless using a NDK that does not provide unified headers.

- `CMAKE_ANDROID_NDK_TOOLCHAIN_VERSION`

  Set to the version of the NDK toolchain to be selected as the compiler. If not specified, the default will be the latest available GCC toolchain.

- `CMAKE_ANDROID_STL_TYPE`

  Set to specify which C++ standard library to use. If not specified, a default will be selected as described in the variable documentation.

The following variables will be computed and provided automatically:

- `CMAKE_ANDROID_TOOLCHAIN_PREFIX`

The absolute path prefix to the binutils in the NDK toolchain.

- `CMAKE_ANDROID_TOOLCHAIN_SUFFIX`

  The host platform suffix of the binutils in the NDK toolchain.

For example, a toolchain file might contain:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_SYSTEM_VERSION 21) # API level
set(CMAKE_ANDROID_ARCH_ABI arm64-v8a)
set(CMAKE_ANDROID_NDK /path/to/android-ndk)
set(CMAKE_ANDROID_STL_TYPE gnustl_static)
```

Alternatively one may specify the values without a toolchain file:

```
$ cmake ../src \
  -DCMAKE_SYSTEM_NAME=Android \
  -DCMAKE_SYSTEM_VERSION=21 \
  -DCMAKE_ANDROID_ARCH_ABI=arm64-v8a \
  -DCMAKE_ANDROID_NDK=/path/to/android-ndk \
  -DCMAKE_ANDROID_STL_TYPE=gnustl_static
```

Cross Compiling for Android with a Standalone Toolchain

A toolchain file may configure Makefile Generators or the `Ninja` generator to target Android for cross-compiling using a standalone toolchain.

Configure use of an Android standalone toolchain with the following variables:

- `CMAKE_SYSTEM_NAME`

  Set to `Android`. Must be specified to enable cross compiling for Android.

- `CMAKE_ANDROID_STANDALONE_TOOLCHAIN`

  Set to the absolute path to the standalone toolchain root directory. A `${CMAKE_ANDROID_STANDALONE_TOOLCHAIN}/sysroot` directory must exist. If not specified, a default for this variable will be chosen as specified above.

- `CMAKE_ANDROID_ARM_MODE`

  When the standalone toolchain targets ARM, optionally set this to `ON` to target 32-bit ARM instead of 16-bit Thumb. See variable documentation for details.

- `CMAKE_ANDROID_ARM_NEON`

  When the standalone toolchain targets ARM v7, optionally set thisto `ON` to target ARM NEON devices. See variable documentation for details.

The following variables will be computed and provided automatically:

- `CMAKE_SYSTEM_VERSION`

  The Android API level detected from the standalone toolchain.

- `CMAKE_ANDROID_ARCH_ABI`

  The Android ABI detected from the standalone toolchain.

- `CMAKE_ANDROID_TOOLCHAIN_PREFIX`

  The absolute path prefix to the binutils in the standalone toolchain.

- `CMAKE_ANDROID_TOOLCHAIN_SUFFIX`

  The host platform suffix of the binutils in the standalone toolchain.

For example, a toolchain file might contain:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_ANDROID_STANDALONE_TOOLCHAIN /path/to/android-toolchain)
```

Alternatively one may specify the values without a toolchain file:

```
$ cmake ../src \
  -DCMAKE_SYSTEM_NAME=Android \
  -DCMAKE_ANDROID_STANDALONE_TOOLCHAIN=/path/to/android-toolchain
```

Cross Compiling for Android with NVIDIA Nsight Tegra Visual Studio Edition¶

A toolchain file to configure one of the Visual Studio Generators to build using NVIDIA Nsight Tegra targeting Android may look like this:

```
set(CMAKE_SYSTEM_NAME Android)
```

The `CMAKE_GENERATOR_TOOLSET` may be set to select the Nsight Tegra “Toolchain Version” value.

See also target properties:

- `ANDROID_ANT_ADDITIONAL_OPTIONS`
- `ANDROID_API_MIN`
- `ANDROID_API`
- `ANDROID_ARCH`
- `ANDROID_ASSETS_DIRECTORIES`
- `ANDROID_GUI`
- `ANDROID_JAR_DEPENDENCIES`
- `ANDROID_JAR_DIRECTORIES`
- `ANDROID_JAVA_SOURCE_DIR`
- `ANDROID_NATIVE_LIB_DEPENDENCIES`
- `ANDROID_NATIVE_LIB_DIRECTORIES`
- `ANDROID_PROCESS_MAX`
- `ANDROID_PROGUARD_CONFIG_PATH`
- `ANDROID_PROGUARD`
- `ANDROID_SECURE_PROPS_PATH`
- `ANDROID_SKIP_ANT_STEP`

- `ANDROID_STL_TYPE`

==========