cmake-commands(7)¶

v3.13

Contents

Scripting Commands

These commands are always available.

- list
- macro
- mark_as_advanced
- math
- message
- option
- return
- separate_arguments
- set_directory_properties
- set_property
- set
- site_name
- string
- unset
- variable_watch
- while

Project Commands

These commands are available only in CMake projects.

- add_compile_definitions
- add_compile_options
- add_custom_command
- add_custom_target
- add_definitions
- add_dependencies
- add_executable
- add_library
- add_link_options
- add_subdirectory
- add_test
- aux_source_directory
- build_command
- create_test_sourcelist
- define_property
- enable_language
- enable_testing
- export
- fltk_wrap_ui
- get_source_file_property
- get_target_property
- get_test_property
- include_directories
- include_external_msproject
- include_regular_expression
- install
- link_directories

- link_libraries
- load_cache
- project
- qt_wrap_cpp
- qt_wrap_ui
- remove_definitions
- set_source_files_properties
- set_target_properties
- set_tests_properties
- source_group
- target_compile_definitions
- target_compile_features
- target_compile_options
- target_include_directories
- target_link_directories
- target_link_libraries
- target_link_options
- target_sources
- try_compile
- try_run

CTest Commands

These commands are available only in CTest scripts.

- ctest_build
- ctest_configure
- ctest_coverage
- ctest_empty_binary_directory
- ctest_memcheck
- ctest_read_custom_files
- ctest_run_script
- ctest_sleep
- ctest_start
- ctest_submit
- ctest_test
- ctest_update
- ctest_upload

Deprecated Commands

These commands are available only for compatibility with older versions of CMake. Do not use them in new code.

- build_name
- exec_program
- export_library_dependencies

- install_files
- install_programs
- install_targets
- load_command
- make_directory
- output_required_files
- remove
- subdir_depends
- subdirs
- use_mangled_mesa
- utility_source
- variable_requires
- write_file

==========

Scripting Commands

break

Break from an enclosing foreach or while loop.

```
break()
```

Breaks from an enclosing foreach loop or while loop

See also the `continue()` command.

cmake_host_system_information

Query host system specific information.

```
cmake_host_system_information(RESULT <variable> QUERY <key> ...)
```

Queries system information of the host system on which cmake runs. One or more `<key>` can be provided to select the information to be queried. The list of queried values is stored in `<variable>`.

`<key>` can be one of the following values:

Key	Description
`NUMBER_OF_LOGICAL_CORES`	Number of logical cores
`NUMBER_OF_PHYSICAL_CORES`	Number of physical cores
`HOSTNAME`	Hostname
`FQDN`	Fully qualified domain name
`TOTAL_VIRTUAL_MEMORY`	Total virtual memory in MiB [1]
`AVAILABLE_VIRTUAL_MEMORY`	Available virtual memory in MiB [1]
`TOTAL_PHYSICAL_MEMORY`	Total physical memory in MiB [1]
`AVAILABLE_PHYSICAL_MEMORY`	Available physical memory in MiB [1]
`IS_64BIT`	One if processor is 64Bit
`HAS_FPU`	One if processor has floating point unit
`HAS_MMX`	One if processor supports MMX instructions
`HAS_MMX_PLUS`	One if processor supports Ext. MMX instructions
`HAS_SSE`	One if processor supports SSE instructions
`HAS_SSE2`	One if processor supports SSE2 instructions
`HAS_SSE_FP`	One if processor supports SSE FP instructions
`HAS_SSE_MMX`	One if processor supports SSE MMX instructions
`HAS_AMD_3DNOW`	One if processor supports 3DNow instructions
`HAS_AMD_3DNOW_PLUS`	One if processor supports 3DNow+ instructions
`HAS_IA64`	One if IA64 processor emulating x86
`HAS_SERIAL_NUMBER`	One if processor has serial number
`PROCESSOR_SERIAL_NUMBER`	Processor serial number
`PROCESSOR_NAME`	Human readable processor name
`PROCESSOR_DESCRIPTION`	Human readable full processor description
`OS_NAME`	See `CMAKE_HOST_SYSTEM_NAME`
`OS_RELEASE`	The OS sub-type e.g. on Windows `Professional`
`OS_VERSION`	The OS build ID
`OS_PLATFORM`	See `CMAKE_HOST_SYSTEM_PROCESSOR`

Footnotes

[1]	***(1, 2, 3, 4)* One MiB (mebibyte) is equal to 1024x1024 bytes.**

cmake_minimum_required

Set the minimum required version of cmake for a project and update Policy Settings to match the version given:

```
cmake_minimum_required(VERSION <min>[...<max>] [FATAL_ERROR])
```

`<min>` and the optional `<max>` are each CMake versions of the form `major.minor[.patch[.tweak]]`, and the `...` is literal.

If the running version of CMake is lower than the `<min>` required version it will stop processing the project and report an error. The optional `<max>` version, if specified, must be at least the `<min>` version and affects policy settings as described below. If the running version of CMake is older than 3.12, the extra `...` dots will be seen as version component separators, resulting in the `...<max>` part being ignored and preserving the pre-3.12 behavior of basing policies on `<min>`.

The `FATAL_ERROR` option is accepted but ignored by CMake 2.6 and higher. It should be specified so CMake versions 2.4 and lower fail with an error instead of just a warning.

Note

Call the `cmake_minimum_required()` command at the beginning of the top-level `CMakeLists.txt` file even before calling the `project()` command. It is important to establish version and policy settings before invoking other commands whose behavior they may affect. See also policy `CMP0000`.

Calling `cmake_minimum_required()` inside a `function()` limits some effects to the function scope when invoked. Such calls should not be made with the intention of having global effects.

Policy Settings

The `cmake_minimum_required(VERSION)` command implicitly invokes the `cmake_policy(VERSION)` command to specify that the current project code is written for the given range of CMake versions. All policies known to the running version of CMake and introduced in the `<min>` (or `<max>`, if specified) version or earlier will be set to use `NEW` behavior. All policies introduced in later versions will be unset. This effectively requests behavior preferred as of a given CMake version and tells newer CMake versions to warn about their new policies.

When a `<min>` version higher than 2.4 is specified the command implicitly invokes:

```
cmake_policy(VERSION <min>[...<max>])
```

which sets CMake policies based on the range of versions specified. When a `<min>` version 2.4 or lower is given the command implicitly invokes:

```
cmake_policy(VERSION 2.4[...<max>])
```

which enables compatibility features for CMake 2.4 and lower.

cmake_parse_arguments

`cmake_parse_arguments` is intended to be used in macros or functions for parsing the arguments given to that macro or function. It processes the arguments and defines a set of variables which hold the values of the respective options.

```
cmake_parse_arguments(<prefix> <options> <one_value_keywords>
                      <multi_value_keywords> args...)

cmake_parse_arguments(PARSE_ARGV N <prefix> <options> <one_value_keywords>
                      <multi_value_keywords>)
```

The first signature reads processes arguments passed in the `args...`. This may be used in either a `macro()` or a `function()`.

The `PARSE_ARGV` signature is only for use in a `function()` body. In this case the arguments that are parsed come from the `ARGV#` variables of the calling function. The parsing starts with the Nth argument, where `N` is an unsigned integer. This allows for the values to have special characters like `;` in them.

The `<options>` argument contains all options for the respective macro, i.e. keywords which can be used when calling the macro without any value following, like e.g. the `OPTIONAL` keyword of the `install()` command.

The `<one_value_keywords>` argument contains all keywords for this macro which are followed by one value, like e.g. `DESTINATION` keyword of the `install()` command.

The `<multi_value_keywords>` argument contains all keywords for this macro which can be followed by more than one value, like e.g. the `TARGETS` or `FILES` keywords of the `install()` command.

Note

All keywords shall be unique. I.e. every keyword shall only be specified once in either `<options>`, `<one_value_keywords>` or `<multi_value_keywords>`. A warning will be emitted if uniqueness is violated.

When done, `cmake_parse_arguments` will consider for each of the keywords listed in `<options>`, `<one_value_keywords>` and `<multi_value_keywords>` a variable composed of the given `<prefix>` followed by `"_"` and the name of the respective keyword. These variables will then hold the respective value from the argument list or be undefined if the associated option could not be found. For the `<options>` keywords, these will always be defined, to `TRUE` or `FALSE`, whether the option is in the argument list or not.

All remaining arguments are collected in a variable `<prefix>_UNPARSED_ARGUMENTS` that will be undefined if all argument where recognized. This can be checked afterwards to see whether your macro was called with unrecognized parameters.

As an example here a `my_install()` macro, which takes similar arguments as the real `install()` command:

```
macro(my_install)
    set(options OPTIONAL FAST)
    set(oneValueArgs DESTINATION RENAME)
    set(multiValueArgs TARGETS CONFIGURATIONS)
    cmake_parse_arguments(MY_INSTALL "${options}" "${oneValueArgs}"
                          "${multiValueArgs}" ${ARGN} )

    # ...
```

Assume `my_install()` has been called like this:

```
my_install(TARGETS foo bar DESTINATION bin OPTIONAL blub)
```

After the `cmake_parse_arguments` call the macro will have set or undefined the following variables:

```
MY_INSTALL_OPTIONAL = TRUE
MY_INSTALL_FAST = FALSE # was not used in call to my_install
MY_INSTALL_DESTINATION = "bin"
MY_INSTALL_RENAME <UNDEFINED> # was not used
MY_INSTALL_TARGETS = "foo;bar"
MY_INSTALL_CONFIGURATIONS <UNDEFINED> # was not used
MY_INSTALL_UNPARSED_ARGUMENTS = "blub" # nothing expected after "OPTIONAL"
```

You can then continue and process these variables.

Keywords terminate lists of values, e.g. if directly after a one_value_keyword another recognized keyword follows, this is interpreted as the beginning of the new option. E.g. `my_install(TARGETS foo DESTINATION OPTIONAL)` would result in `MY_INSTALL_DESTINATION` set to `"OPTIONAL"`, but as `OPTIONAL` is a keyword itself `MY_INSTALL_DESTINATION` will be empty and `MY_INSTALL_OPTIONAL` will therefore be set to `TRUE`.

cmake_policy

Manage CMake Policy settings. See the `cmake-policies(7)` manual for defined policies.

As CMake evolves it is sometimes necessary to change existing behavior in order to fix bugs or improve implementations of existing features. The CMake Policy mechanism is designed to help keep existing projects building as new versions of CMake introduce changes in behavior. Each new policy (behavioral change) is given an identifier of the form `CMP<NNNN>` where `<NNNN>` is an integer index. Documentation associated with each policy describes the `OLD` and `NEW` behavior and the reason the policy was introduced. Projects may set each policy to select the desired behavior. When CMake needs to know which behavior to use it checks for a setting specified by the project. If no setting is available the `OLD` behavior is assumed and a warning is produced requesting that the policy be set.

Setting Policies by CMake Version

The `cmake_policy` command is used to set policies to `OLD` or `NEW` behavior. While setting policies individually is supported, we encourage projects to set policies based on CMake versions:

```
cmake_policy(VERSION <min>[...<max>])
```

`<min>` and the optional `<max>` are each CMake versions of the form `major.minor[.patch[.tweak]]`, and the `...` is literal. The `<min>` version must be at least `2.4` and at most the running version of CMake. The `<max>` version, if specified, must be at least the `<min>` version but may exceed the running version of CMake. If the running version of CMake is older than 3.12, the extra `...` dots will be seen as version component separators, resulting in the `...<max>` part being ignored and preserving the pre-3.12 behavior of basing policies on `<min>`.

This specifies that the current CMake code is written for the given range of CMake versions. All policies known to the running version of CMake and introduced in the `<min>` (or `<max>`, if specified) version or earlier will be set to use `NEW` behavior. All policies introduced in later versions will be unset (unless the `CMAKE_POLICY_DEFAULT_CMP` variable sets a default). This effectively requests behavior preferred as of a given CMake version and tells newer CMake versions to warn about their new policies.

Note that the `cmake_minimum_required(VERSION)` command implicitly calls `cmake_policy(VERSION)` too.

Setting Policies Explicitly

```
cmake_policy(SET CMP<NNNN> NEW)
cmake_policy(SET CMP<NNNN> OLD)
```

Tell CMake to use the `OLD` or `NEW` behavior for a given policy. Projects depending on the old behavior of a given policy may silence a policy warning by setting the policy state to `OLD`. Alternatively one may fix the project to work with the new behavior and set the policy state to `NEW`.

Note

The `OLD` behavior of a policy is `deprecated by definition` and may be removed in a future version of CMake.

Checking Policy Settings

```
cmake_policy(GET CMP<NNNN> <variable>)
```

Check whether a given policy is set to `OLD` or `NEW` behavior. The output `<variable>` value will be `OLD` or `NEW` if the policy is set, and empty otherwise.

CMake Policy Stack

CMake keeps policy settings on a stack, so changes made by the cmake_policy command affect only the top of the stack. A new entry on the policy stack is managed automatically for each subdirectory to protect its parents and siblings. CMake also manages a new entry for scripts loaded by `include()` and `find_package()` commands except when invoked with the `NO_POLICY_SCOPE` option (see also policy `CMP0011`). The `cmake_policy` command provides an interface to manage custom entries on the policy stack:

```
cmake_policy(PUSH)
cmake_policy(POP)
```

Each `PUSH` must have a matching `POP` to erase any changes. This is useful to make temporary changes to policy settings. Calls to the `cmake_minimum_required(VERSION)`, `cmake_policy(VERSION)`, or `cmake_policy(SET)` commands influence only the current top of the policy stack.

Commands created by the `function()` and `macro()` commands record policy settings when they are created and use the pre-record policies when they are invoked. If the function or macro implementation sets policies, the changes automatically propagate up through callers until they reach the closest nested policy stack entry.

configure_file

Copy a file to another location and modify its contents.

```
configure_file(<input> <output>
               [COPYONLY] [ESCAPE_QUOTES] [@ONLY]
               [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])
```

Copies an `<input>` file to an `<output>` file and substitutes variable values referenced as `@VAR@` or `${VAR}` in the input file content. Each variable reference will be replaced with the current value of the variable, or the empty string if the variable is not defined. Furthermore, input lines of the form:

```
#cmakedefine VAR ...
```

will be replaced with either:

```
#define VAR ...
```

or:

```
/* #undef VAR */
```

depending on whether `VAR` is set in CMake to any value not considered a false constant by the `if()` command. The "..." content on the line after the variable name, if any, is processed as above. Input file lines of the form `#cmakedefine01 VAR` will be replaced with either `#define VAR 1` or `#define VAR 0` similarly. The result lines (with the exception of the `#undef` comments) can be indented using spaces and/or tabs between the `#` character and the `cmakedefine` or `cmakedefine01` words. This whitespace indentation will be preserved in the output lines:

```
#  cmakedefine VAR
#  cmakedefine01 VAR
```

will be replaced, if `VAR` is defined, with:

```
#  define VAR
#  define VAR 1
```

If the input file is modified the build system will re-run CMake to re-configure the file and generate the build system again. The generated file is modified and its timestamp updated on subsequent cmake runs only if its content is changed.

The arguments are:

- `<input>`

  Path to the input file. A relative path is treated with respect to the value of `CMAKE_CURRENT_SOURCE_DIR`. The input path must be a file, not a directory.

- `<output>`

  Path to the output file or directory. A relative path is treated with respect to the value of `CMAKE_CURRENT_BINARY_DIR`. If the path names an existing directory the output file is placed in that directory with the same file name as the input file.

- `COPYONLY`

  Copy the file without replacing any variable references or other content. This option may not be used with `NEWLINE_STYLE`.

- `ESCAPE_QUOTES`

  Escape any substituted quotes with backslashes (C-style).

- `@ONLY`

  Restrict variable replacement to references of the form `@VAR@`. This is useful for configuring scripts that use `${VAR}` syntax.

- `NEWLINE_STYLE <style>`

  Specify the newline style for the output file. Specify `UNIX` or `LF` for `\n` newlines, or specify `DOS`, `WIN32`, or `CRLF` for `\r\n` newlines. This option may not be used with `COPYONLY`.

Example

Consider a source tree containing a `foo.h.in` file:

```
#cmakedefine FOO_ENABLE
#cmakedefine FOO_STRING "@FOO_STRING@"
```

An adjacent `CMakeLists.txt` may use `configure_file` to configure the header:

```
option(FOO_ENABLE "Enable Foo" ON)
if(FOO_ENABLE)
  set(FOO_STRING "foo")
endif()
configure_file(foo.h.in foo.h @ONLY)
```

This creates a `foo.h` in the build directory corresponding to this source directory. If the `FOO_ENABLE` option is on, the configured file will contain:

```
#define FOO_ENABLE
#define FOO_STRING "foo"
```

Otherwise it will contain:

```
/* #undef FOO_ENABLE */
/* #undef FOO_STRING */
```

One may then use the `include_directories()` command to specify the output directory as an include directory:

```
include_directories(${CMAKE_CURRENT_BINARY_DIR})
```

so that sources may include the header as `#include <foo.h>`.

continue

Continue to the top of enclosing foreach or while loop.

```
continue()
```

The `continue` command allows a cmake script to abort the rest of a block in a `foreach()` or `while()` loop, and start at the top of the next iteration. See also the `break()` command.

elseif

Starts the elseif portion of an if block.

```
elseif(expression)
```

See the `if()` command.

else

Starts the else portion of an if block.

```
else(expression)
```

See the `if()` command.

endforeach

Ends a list of commands in a foreach block.

```
endforeach(expression)
```

See the `foreach()` command.

endfunction

Ends a list of commands in a function block.

```
endfunction(expression)
```

See the `function()` command.

endif

Ends a list of commands in an if block.

```
endif(expression)
```

See the `if()` command.

endmacro

Ends a list of commands in a macro block.

```
endmacro(expression)
```

See the `macro()` command.

endwhile

Ends a list of commands in a while block.

```
endwhile(expression)
```

See the `while()` command.

execute_process

Execute one or more child processes.

```
execute_process(COMMAND <cmd1> [args1...]]
                [COMMAND <cmd2> [args2...] [...]]
```

```
                  [WORKING_DIRECTORY <directory>]
                  [TIMEOUT <seconds>]
                  [RESULT_VARIABLE <variable>]
                  [RESULTS_VARIABLE <variable>]
                  [OUTPUT_VARIABLE <variable>]
                  [ERROR_VARIABLE <variable>]
                  [INPUT_FILE <file>]
                  [OUTPUT_FILE <file>]
                  [ERROR_FILE <file>]
                  [OUTPUT_QUIET]
                  [ERROR_QUIET]
                  [OUTPUT_STRIP_TRAILING_WHITESPACE]
                  [ERROR_STRIP_TRAILING_WHITESPACE]
                  [ENCODING <name>])
```

Runs the given sequence of one or more commands in parallel with the standard output of each process piped to the standard input of the next. A single standard error pipe is used for all processes.

Options:

- `COMMAND`

  A child process command line.CMake executes the child process using operating system APIs directly. All arguments are passed VERBATIM to the child process. No intermediate shell is used, so shell operators such as `>` are treated as normal arguments. (Use the `INPUT_*`, `OUTPUT_*`, and `ERROR_*` options to redirect stdin, stdout, and stderr.)If a sequential execution of multiple commands is required, use multiple `execute_process()` calls with a single `COMMAND` argument.

- `WORKING_DIRECTORY`

  The named directory will be set as the current working directory of the child processes.

- `TIMEOUT`

  The child processes will be terminated if they do not finish in the specified number of seconds (fractions are allowed).

- `RESULT_VARIABLE`

  The variable will be set to contain the result of last child process. This will be an integer return code from the last child or a string describing an error condition.

- `RESULTS_VARIABLE <variable>`

  The variable will be set to contain the result of all processes as a ;-list, in order of the given `COMMAND` arguments. Each entry will be an integer return code from the corresponding child or a string describing an error condition.

- `OUTPUT_VARIABLE`, `ERROR_VARIABLE`

  The variable named will be set with the contents of the standard output and standard error pipes, respectively. If the same variable is named for both pipes their output will be merged in the order produced.

- `INPUT_FILE, OUTPUT_FILE`, `ERROR_FILE`

  The file named will be attached to the standard input of the first process, standard output of the last process, or standard error of all processes, respectively. If the same file is named for both output and error then it will be used for both.

- `OUTPUT_QUIET`, `ERROR_QUIET`

  The standard output or standard error results will be quietly ignored.

- `ENCODING <name>`

  On Windows, the encoding that is used to decode output from the process. Ignored on other platforms. Valid encoding names are: `NONE` Perform no decoding. This assumes that the process output is encoded in the same way as CMake's internal encoding (UTF-8). This is the default. `AUTO` Use the current active console's codepage or if that isn't available then use ANSI. `ANSI` Use the ANSI codepage. `OEM` Use the original equipment manufacturer (OEM) code page. `UTF8` or `UTF-8` Use the UTF-8 codepage. Prior to CMake 3.11.0, only `UTF8` was accepted for this encoding. In CMake 3.11.0, `UTF-8` was added for consistency with the UTF-8 RFC naming convention.

If more than one `OUTPUT_*` or `ERROR_*` option is given for the same pipe the precedence is not specified. If no `OUTPUT_*` or `ERROR_*` options are given the output will be shared with the corresponding pipes of the CMake process itself.

The `execute_process()` command is a newer more powerful version of `exec_program()`, but the old command has been kept for compatibility. Both commands run while CMake is processing the project prior to build system generation. Use `add_custom_target()` and `add_custom_command()` to create custom commands that run at build time.

file

File manipulation command.

Synopsis

```
Reading
  file(READ <filename> <out-var> [...])
  file(STRINGS <filename> <out-var> [...])
  file(<HASH> <filename> <out-var>)
  file(TIMESTAMP <filename> <out-var> [...])

Writing
  file({WRITE | APPEND} <filename> <content>...)
  file({TOUCH | TOUCH_NOCREATE} [<file>...])
  file(GENERATE OUTPUT <output-file> [...])

Filesystem
  file({GLOB | GLOB_RECURSE} <out-var> [...] [<globbing-expr>...])
  file(RENAME <oldname> <newname>)
  file({REMOVE | REMOVE_RECURSE } [<files>...])
  file(MAKE_DIRECTORY [<dir>...])
  file({COPY | INSTALL} <file>... DESTINATION <dir> [...])

Path Conversion
  file(RELATIVE_PATH <out-var> <directory> <file>)
  file({TO_CMAKE_PATH | TO_NATIVE_PATH} <path> <out-var>)

Transfer
```

```
  file(DOWNLOAD <url> <file> [...])
  file(UPLOAD <file> <url> [...])

Locking
  file(LOCK <path> [...])
```

Reading

```
file(READ <filename> <variable>
     [OFFSET <offset>] [LIMIT <max-in>] [HEX])
```

Read content from a file called `<filename>` and store it in a `<variable>`. Optionally start from the given `<offset>` and read at most `<max-in>` bytes. The `HEX` option causes data to be converted to a hexadecimal representation (useful for binary data).

```
file(STRINGS <filename> <variable> [<options>...])
```

Parse a list of ASCII strings from `<filename>` and store it in `<variable>`. Binary data in the file are ignored. Carriage return (`\r`, CR) characters are ignored. The options are:

- `LENGTH_MAXIMUM <max-len>`

  Consider only strings of at most a given length.

- `LENGTH_MINIMUM <min-len>`

  Consider only strings of at least a given length.

- `LIMIT_COUNT <max-num>`

  Limit the number of distinct strings to be extracted.

- `LIMIT_INPUT <max-in>`

  Limit the number of input bytes to read from the file.

- `LIMIT_OUTPUT <max-out>`

  Limit the number of total bytes to store in the `<variable>`.

- `NEWLINE_CONSUME`

  Treat newline characters (`\n`, LF) as part of string content instead of terminating at them.

- `NO_HEX_CONVERSION`

  Intel Hex and Motorola S-record files are automatically converted to binary while reading unless this option is given.

- `REGEX <regex>`

  Consider only strings that match the given regular expression.

- `ENCODING <encoding-type>`

  Consider strings of a given encoding. Currently supported encodings are: UTF-8, UTF-16LE, UTF-16BE, UTF-32LE, UTF-32BE. If the ENCODING option is not provided and the file has a Byte Order Mark, the ENCODING option will be defaulted to respect the Byte Order Mark.

For example, the code

```
file(STRINGS myfile.txt myfile)
```

stores a list in the variable `myfile` in which each item is a line from the input file.

```
file(<HASH> <filename> <variable>)
```

Compute a cryptographic hash of the content of `<filename>` and store it in a `<variable>`. The supported `<HASH>` algorithm names are those listed by the string() command.

```
file(TIMESTAMP <filename> <variable> [<format>] [UTC])
```

Compute a string representation of the modification time of `<filename>` and store it in `<variable>`. Should the command be unable to obtain a timestamp variable will be set to the empty string (“”).

See the `string(TIMESTAMP)` command for documentation of the `<format>` and `UTC` options.

Writing

```
file(WRITE <filename> <content>...)
file(APPEND <filename> <content>...)
```

Write `<content>` into a file called `<filename>`. If the file does not exist, it will be created. If the file already exists, `WRITE` mode will overwrite it and `APPEND` mode will append to the end. Any directories in the path specified by `<filename>` that do not exist will be created.

If the file is a build input, use the `configure_file()` command to update the file only when its content changes.

```
file(TOUCH [<files>...])
file(TOUCH_NOCREATE [<files>...])
```

Create a file with no content if it does not yet exist. If the file already exists, its access and/or modification will be updated to the time when the function call is executed.

Use TOUCH_NOCREATE to touch a file if it exists but not create it. If a file does not exist it will be silently ignored.

With TOUCH and TOUCH_NOCREATE the contents of an existing file will not be modified.

```
file(GENERATE OUTPUT output-file
     <INPUT input-file|CONTENT content>
     [CONDITION expression])
```

Generate an output file for each build configuration supported by the current `CMake Generator`. Evaluate `generatorexpressions` from the input content to produce the output content. The options are:

- `CONDITION <condition>`

  Generate the output file for a particular configuration only if the condition is true. The condition must be either `0` or `1` after evaluating generator expressions.

- `CONTENT <content>`

  Use the content given explicitly as input.

- `INPUT <input-file>`

  Use the content from a given file as input. A relative path is treated with respect to the value of `CMAKE_CURRENT_SOURCE_DIR`. See policy `CMP0070`.

- `OUTPUT <output-file>`

  Specify the output file name to generate. Use generator expressions such as `$<CONFIG>` to specify a configuration-specific output file name. Multiple configurations may generate the same output file only if the generated content is identical. Otherwise, the `<output-file>` must evaluate to an unique name for each configuration. A relative path (after evaluating generator expressions) is treated with respect to the value of `CMAKE_CURRENT_BINARY_DIR`. See policy `CMP0070`.

Exactly one `CONTENT` or `INPUT` option must be given. A specific `OUTPUT` file may be named by at most one invocation of `file(GENERATE)`. Generated files are modified and their timestamp updated on subsequent cmake runs only if their content is changed.

Note also that `file(GENERATE)` does not create the output file until the generation phase. The output file will not yet have been written when the `file(GENERATE)` command returns, it is written only after processing all of a project's `CMakeLists.txt` files.

Filesystem

```
file(GLOB <variable>
     [LIST_DIRECTORIES true|false] [RELATIVE <path>] [CONFIGURE_DEPENDS]
     [<globbing-expressions>...])
file(GLOB_RECURSE <variable> [FOLLOW_SYMLINKS]
     [LIST_DIRECTORIES true|false] [RELATIVE <path>] [CONFIGURE_DEPENDS]
     [<globbing-expressions>...])
```

Generate a list of files that match the `<globbing-expressions>` and store it into the `<variable>`. Globbing expressions are similar to regular expressions, but much simpler. If `RELATIVE` flag is specified, the results will be returned as relative paths to the given path. The results will be ordered lexicographically.

If the `CONFIGURE_DEPENDS` flag is specified, CMake will add logic to the main build system check target to rerun the flagged `GLOB` commands at build time. If any of the outputs change, CMake will regenerate the build system.

By default `GLOB` lists directories - directories are omitted in result if `LIST_DIRECTORIES` is set to false.

Note

We do not recommend using GLOB to collect a list of source files from your source tree. If no CMakeLists.txt file changes when a source is added or removed then the generated build system cannot know when to ask CMake to regenerate. The `CONFIGURE_DEPENDS` flag may not work reliably on all generators, or if a new generator is added in the future that cannot support it, projects using it will be stuck. Even if `CONFIGURE_DEPENDS` works reliably, there is still a cost to perform the check on every rebuild.

Examples of globbing expressions include:

```
*.cxx      - match all files with extension cxx
*.vt?      - match all files with extension vta,...,vtz
f[3-5].txt - match files f3.txt, f4.txt, f5.txt
```

The `GLOB_RECURSE` mode will traverse all the subdirectories of the matched directory and match the files. Subdirectories that are symlinks are only traversed if `FOLLOW_SYMLINKS` is given or policy `CMP0009` is not set to `NEW`.

By default `GLOB_RECURSE` omits directories from result list - setting `LIST_DIRECTORIES` to true adds directories to result list. If `FOLLOW_SYMLINKS` is given or policy `CMP0009` is not set to `OLD` then `LIST_DIRECTORIES` treats symlinks as directories.

Examples of recursive globbing include:

```
/dir/*.py  - match all python files in /dir and subdirectories
file(RENAME <oldname> <newname>)
```

Move a file or directory within a filesystem from `<oldname>` to `<newname>`, replacing the destination atomically.

```
file(REMOVE [<files>...])
file(REMOVE_RECURSE [<files>...])
```

Remove the given files. The `REMOVE_RECURSE` mode will remove the given files and directories, also non-empty directories. No error is emitted if a given file does not exist.

```
file(MAKE_DIRECTORY [<directories>...])
```

Create the given directories and their parents as needed.

```
file(<COPY|INSTALL> <files>... DESTINATION <dir>
     [FILE_PERMISSIONS <permissions>...]
     [DIRECTORY_PERMISSIONS <permissions>...]
     [NO_SOURCE_PERMISSIONS] [USE_SOURCE_PERMISSIONS]
     [FILES_MATCHING]
     [[PATTERN <pattern> | REGEX <regex>]
      [EXCLUDE] [PERMISSIONS <permissions>...]] [...])
```

The `COPY` signature copies files, directories, and symlinks to a destination folder. Relative input paths are evaluated with respect to the current source directory, and a relative destination is evaluated with respect to the current build directory. Copying preserves input file timestamps, and optimizes out a file if it exists at the destination with the same timestamp. Copying preserves input permissions unless explicit permissions or `NO_SOURCE_PERMISSIONS` are given (default is `USE_SOURCE_PERMISSIONS`).

See the `install(DIRECTORY)` command for documentation of permissions, `FILES_MATCHING`, `PATTERN`, `REGEX`, and `EXCLUDE` options. Copying directories preserves the structure of their content even if options are used to select a subset of files.

The `INSTALL` signature differs slightly from `COPY`: it prints status messages (subject to the `CMAKE_INSTALL_MESSAGE` variable), and `NO_SOURCE_PERMISSIONS` is default. Installation scripts generated by the `install()` command use this signature (with some undocumented options for internal use).

Path Conversion

```
file(RELATIVE_PATH <variable> <directory> <file>)
```

Compute the relative path from a `<directory>` to a `<file>` and store it in the `<variable>`.

```
file(TO_CMAKE_PATH "<path>" <variable>)
file(TO_NATIVE_PATH "<path>" <variable>)
```

The `TO_CMAKE_PATH` mode converts a native `<path>` into a cmake-style path with forward-slashes (`/`). The input can be a single path or a system search path like `$ENV{PATH}`. A search path will be converted to a cmake-style list separated by `;` characters.

The `TO_NATIVE_PATH` mode converts a cmake-style `<path>` into a native path with platform-specific slashes (`\` on Windows and `/` elsewhere).

Always use double quotes around the `<path>` to be sure it is treated as a single argument to this command.

Transfer

```
file(DOWNLOAD <url> <file> [<options>...])
file(UPLOAD   <file> <url> [<options>...])
```

The `DOWNLOAD` mode downloads the given `<url>` to a local `<file>`. The `UPLOAD` mode uploads a local `<file>` to a given `<url>`.

Options to both `DOWNLOAD` and `UPLOAD` are:

- `INACTIVITY_TIMEOUT <seconds>`

  Terminate the operation after a period of inactivity.

- `LOG <variable>`

Store a human-readable log of the operation in a variable.

- `SHOW_PROGRESS`

  Print progress information as status messages until the operation is complete.

- `STATUS <variable>`

  Store the resulting status of the operation in a variable. The status is a `;` separated list of length 2. The first element is the numeric return value for the operation, and the second element is a string value for the error. A `0` numeric error means no error in the operation.

- `TIMEOUT <seconds>`

  Terminate the operation after a given total time has elapsed.

- `USERPWD <username>:<password>`

  Set username and password for operation.

- `HTTPHEADER <HTTP-header>`

  HTTP header for operation. Suboption can be repeated several times.

- `NETRC <level>`

  Specify whether the .netrc file is to be used for operation. If this option is not specified, the value of the `CMAKE_NETRC` variable will be used instead. Valid levels are: `IGNORED` The .netrc file is ignored. This is the default. `OPTIONAL` The .netrc file is optional, and information in the URL is preferred. The file will be scanned to find which ever information is not specified in the URL. `REQUIRED` The .netrc file is required, and information in the URL is ignored.

- `NETRC_FILE <file>`

  Specify an alternative .netrc file to the one in your home directory, if the `NETRC` level is `OPTIONAL` or `REQUIRED`. If this option is not specified, the value of the `CMAKE_NETRC_FILE` variable will be used instead.

If neither `NETRC` option is given CMake will check variables `CMAKE_NETRC` and `CMAKE_NETRC_FILE`, respectively.

Additional options to `DOWNLOAD` are:

```
EXPECTED_HASH ALGO=<value>
```

> Verify that the downloaded content hash matches the expected value, where `ALGO` is one of the algorithms supported by `file(<HASH>)`. If it does not match, the operation fails with an error.

- `EXPECTED_MD5 <value>`

  Historical short-hand for `EXPECTED_HASH MD5=<value>`.

- `TLS_VERIFY <ON|OFF>`

  Specify whether to verify the server certificate for `https://` URLs. The default is to *not* verify.

- `TLS_CAINFO <file>`

  Specify a custom Certificate Authority file for `https://` URLs.

For `https://` URLs CMake must be built with OpenSSL support. `TLS/SSL` certificates are not checked by default. Set `TLS_VERIFY` to `ON` to check certificates and/or use `EXPECTED_HASH` to verify downloaded content. If neither `TLS` option is given CMake will check variables `CMAKE_TLS_VERIFY` and `CMAKE_TLS_CAINFO`, respectively.

Locking

```
file(LOCK <path> [DIRECTORY] [RELEASE]
     [GUARD <FUNCTION|FILE|PROCESS>]
     [RESULT_VARIABLE <variable>]
     [TIMEOUT <seconds>])
```

Lock a file specified by `<path>` if no `DIRECTORY` option present and file `<path>/cmake.lock` otherwise. File will be locked for scope defined by `GUARD` option (default value is `PROCESS`). `RELEASE` option can be used to unlock file explicitly. If option `TIMEOUT` is not specified CMake will wait until lock succeed or until fatal error occurs. If `TIMEOUT` is set to `0` lock will be tried once and result will be reported immediately. If `TIMEOUT` is not `0` CMake will try to lock file for the period specified by `<seconds>` value. Any errors will be interpreted as fatal if there is no `RESULT_VARIABLE` option. Otherwise result will be stored in `<variable>` and will be `0` on success or error message on failure.

Note that lock is advisory - there is no guarantee that other processes will respect this lock, i.e. lock synchronize two or more CMake instances sharing some modifiable resources. Similar logic applied to `DIRECTORY` option - locking parent directory doesn't prevent other `LOCK` commands to lock any child directory or file.

Trying to lock file twice is not allowed. Any intermediate directories and file itself will be created if they not exist. `GUARD` and `TIMEOUT` options ignored on `RELEASE` operation.

find_file

A short-hand signature is:

```
find_file (<VAR> name1 [path1 path2 ...])
```

The general signature is:

```
find_file (
          <VAR>
          name | NAMES name1 [name2 ...]
          [HINTS path1 [path2 ... ENV var]]
          [PATHS path1 [path2 ... ENV var]]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [DOC "cache documentation string"]
          [NO_DEFAULT_PATH]
          [NO_PACKAGE_ROOT_PATH]
          [NO_CMAKE_PATH]
          [NO_CMAKE_ENVIRONMENT_PATH]
          [NO_SYSTEM_ENVIRONMENT_PATH]
          [NO_CMAKE_SYSTEM_PATH]
```

```
            [CMAKE_FIND_ROOT_PATH_BOTH |
             ONLY_CMAKE_FIND_ROOT_PATH |
             NO_CMAKE_FIND_ROOT_PATH]
           )
```

This command is used to find a full path to named file. A cache entry named by `<VAR>` is created to store the result of this command. If the full path to a file is found the result is stored in the variable and the search will not be repeated unless the variable is cleared. If nothing is found, the result will be `<VAR>-NOTFOUND`, and the search will be attempted again the next time find_file is invoked with the same variable.

Options include:

- `NAMES`

  Specify one or more possible names for the full path to a file.When using this to specify names with and without a version suffix, we recommend specifying the unversioned name first so that locally-built packages can be found before those provided by distributions.

- `HINTS`, `PATHS`

  Specify directories to search in addition to the default locations. The `ENV var` sub-option reads paths from a system environment variable.

- `PATH_SUFFIXES`

  Specify additional subdirectories to check below each directory location otherwise considered.

- `DOC`

  Specify the documentation string for the `<VAR>` cache entry.

If `NO_DEFAULT_PATH` is specified, then no additional paths are added to the search. If `NO_DEFAULT_PATH` is not specified, the search process is as follows:

1. If called from within a find module loaded by

   `find_package(<PackageName>)`

   , search prefixes unique to the current package being found. Specifically look in the

   `<PackageName>_ROOT`

   CMake variable and the

   `<PackageName>_ROOT`

   environment variable. The package root variables are maintained as a stack so if called from nested find modules, root paths from the parent's find module will be searched after paths from the current module, i.e.

   ```
   <CurrentPackage>_ROOT
   ```

   ,

```
ENV{<CurrentPackage>_ROOT}
```

,

```
<ParentPackage>_ROOT
```

,

```
ENV{<ParentPackage>_ROOT}
```

, etc. This can be skipped if

```
NO_PACKAGE_ROOT_PATH
```

is passed. See policy

`CMP0074`

.

- `<prefix>/include/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/include` for each `<prefix>` in the`_ROOT` CMake variable and the `_ROOT` environment variable if called from within a find module loaded by `find_package()`

2. Search paths specified in cmake-specific cache variables. These are intended to be used on the command line with a

```
-DVAR=value
```

. The values are interpreted as

;-lists

. This can be skipped if

```
NO_CMAKE_PATH
```

is passed.

- `<prefix>/include/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/include` for each `<prefix>` in `CMAKE_PREFIX_PATH`
- `CMAKE_INCLUDE_PATH`
- `CMAKE_FRAMEWORK_PATH`

3. Search paths specified in cmake-specific environment variables. These are intended to be set in the user's shell configuration, and therefore use the host's native path separator (

```
;
```

on Windows and

```
:
```

on UNIX). This can be skipped if

```
NO_CMAKE_ENVIRONMENT_PATH
```

is passed.

- `<prefix>/include/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/include` for each `<prefix>` in `CMAKE_PREFIX_PATH`
- `CMAKE_INCLUDE_PATH`
- `CMAKE_FRAMEWORK_PATH`

4. Search the paths specified by the `HINTS` option. These should be paths computed by system introspection, such as a hint provided by the location of another item already found. Hard-coded guesses should be specified with the `PATHS` option.
5. Search the standard system environment variables. This can be skipped if

```
NO_SYSTEM_ENVIRONMENT_PATH
```

is an argument.

- Directories in `INCLUDE`. On Windows hosts: `<prefix>/include/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/include` for each `<prefix>/[s]bin` in `PATH`, and `<entry>/include` for other entries in `PATH`, and the directories in `PATH` itself.

6. Search cmake variables defined in the Platform files for the current system. This can be skipped if

```
NO_CMAKE_SYSTEM_PATH
```

is passed.

- `<prefix>/include/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/include` for each `<prefix>` in `CMAKE_SYSTEM_PREFIX_PATH`
- `CMAKE_SYSTEM_INCLUDE_PATH`
- `CMAKE_SYSTEM_FRAMEWORK_PATH`

7. Search the paths specified by the PATHS option or in the short-hand version of the command. These are typically hard-coded guesses.

On macOS the `CMAKE_FIND_FRAMEWORK` and `CMAKE_FIND_APPBUNDLE` variables determine the order of preference between Apple-style and unix-style package components.

The CMake variable `CMAKE_FIND_ROOT_PATH` specifies one or more directories to be prepended to all other search directories. This effectively “re-roots” the entire search under given locations. Paths which are descendants of the `CMAKE_STAGING_PREFIX` are excluded from this re-rooting, because that variable is always a path on the host system. By default the `CMAKE_FIND_ROOT_PATH` is empty.

The `CMAKE_SYSROOT` variable can also be used to specify exactly one directory to use as a prefix. Setting `CMAKE_SYSROOT` also has other effects. See the documentation for that variable for more.

These variables are especially useful when cross-compiling to point to the root directory of the target environment and CMake will search there too. By default at first the directories listed in `CMAKE_FIND_ROOT_PATH` are searched, then the `CMAKE_SYSROOT` directory is searched, and then the non-rooted directories will be searched. The default behavior can be adjusted by setting `CMAKE_FIND_ROOT_PATH_MODE_INCLUDE`. This behavior can be manually overridden on a per-call basis using options:

- `CMAKE_FIND_ROOT_PATH_BOTH`

  Search in the order described above.

- `NO_CMAKE_FIND_ROOT_PATH`

  Do not use the `CMAKE_FIND_ROOT_PATH` variable.

- `ONLY_CMAKE_FIND_ROOT_PATH`

  Search only the re-rooted directories and directories below `CMAKE_STAGING_PREFIX`.

The default search order is designed to be most-specific to least-specific for common use cases. Projects may override the order by simply calling the command multiple times and using the `NO_*` options:

```
find_file (<VAR> NAMES name PATHS paths... NO_DEFAULT_PATH)
find_file (<VAR> NAMES name)
```

Once one of the calls succeeds the result variable will be set and stored in the cache so that no call will search again.

find_library

A short-hand signature is:

```
find_library (<VAR> name1 [path1 path2 ...])
```

The general signature is:

```
find_library (
          <VAR>
          name | NAMES name1 [name2 ...] [NAMES_PER_DIR]
          [HINTS path1 [path2 ... ENV var]]
          [PATHS path1 [path2 ... ENV var]]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [DOC "cache documentation string"]
          [NO_DEFAULT_PATH]
          [NO_PACKAGE_ROOT_PATH]
          [NO_CMAKE_PATH]
          [NO_CMAKE_ENVIRONMENT_PATH]
          [NO_SYSTEM_ENVIRONMENT_PATH]
          [NO_CMAKE_SYSTEM_PATH]
          [CMAKE_FIND_ROOT_PATH_BOTH |
           ONLY_CMAKE_FIND_ROOT_PATH |
           NO_CMAKE_FIND_ROOT_PATH]
         )
```

This command is used to find a library. A cache entry named by `<VAR>` is created to store the result of this command. If the library is found the result is stored in the variable and the search will not be repeated unless the variable is cleared. If nothing is found, the result will be `<VAR>-NOTFOUND`, and the search will be attempted again the next time find_library is invoked with the same variable.

Options include:

- `NAMES`

  Specify one or more possible names for the library.When using this to specify names with and without a version suffix, we recommend specifying the unversioned name first so that locally-built packages can be found before those provided by distributions.

- `HINTS`, `PATHS`

  Specify directories to search in addition to the default locations. The `ENV var` sub-option reads paths from a system environment variable.

- `PATH_SUFFIXES`

  Specify additional subdirectories to check below each directory location otherwise considered.

- `DOC`

  Specify the documentation string for the `<VAR>` cache entry.

If `NO_DEFAULT_PATH` is specified, then no additional paths are added to the search. If `NO_DEFAULT_PATH` is not specified, the search process is as follows:

1. If called from within a find module loaded by

   `find_package(<PackageName>)`

   , search prefixes unique to the current package being found. Specifically look in the

   `<PackageName>_ROOT`

   CMake variable and the

   `<PackageName>_ROOT`

   environment variable. The package root variables are maintained as a stack so if called from nested find modules, root paths from the parent's find module will be searched after paths from the current module, i.e.

   ```
   <CurrentPackage>_ROOT
   ```

   ,

   ```
   ENV{<CurrentPackage>_ROOT}
   ```

   ,

```
<ParentPackage>_ROOT
```

,

```
ENV{<ParentPackage>_ROOT}
```

, etc. This can be skipped if

```
NO_PACKAGE_ROOT_PATH
```

is passed. See policy

`CMP0074`

.

- `<prefix>/lib/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/lib` for each `<prefix>` in the `_ROOT` CMake variable and the `_ROOT` environment variable if called from within a find module loaded by `find_package()`

2. Search paths specified in cmake-specific cache variables. These are intended to be used on the command line with a

```
-DVAR=value
```

. The values are interpreted as

;-lists

. This can be skipped if

```
NO_CMAKE_PATH
```

is passed.

- `<prefix>/lib/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/lib` for each `<prefix>` in `CMAKE_PREFIX_PATH`
- `CMAKE_LIBRARY_PATH`
- `CMAKE_FRAMEWORK_PATH`

3. Search paths specified in cmake-specific environment variables. These are intended to be set in the user's shell configuration, and therefore use the host's native path separator (

```
;
```

on Windows and

```
:
```

on UNIX). This can be skipped if

```
NO_CMAKE_ENVIRONMENT_PATH
```

is passed.

- `<prefix>/lib/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/lib` for each `<prefix>` in `CMAKE_PREFIX_PATH`
- `CMAKE_LIBRARY_PATH`
- `CMAKE_FRAMEWORK_PATH`

4. Search the paths specified by the `HINTS` option. These should be paths computed by system introspection, such as a hint provided by the location of another item already found. Hard-coded guesses should be specified with the `PATHS` option.
5. Search the standard system environment variables. This can be skipped if

```
NO_SYSTEM_ENVIRONMENT_PATH
```

is an argument.

- Directories in `LIB`. On Windows hosts: `<prefix>/lib/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/lib` for each `<prefix>/[s]bin` in `PATH`, and `<entry>/lib` for other entries in `PATH`, and the directories in `PATH` itself.

6. Search cmake variables defined in the Platform files for the current system. This can be skipped if

```
NO_CMAKE_SYSTEM_PATH
```

is passed.

- `<prefix>/lib/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/lib` for each `<prefix>` in `CMAKE_SYSTEM_PREFIX_PATH`
- `CMAKE_SYSTEM_LIBRARY_PATH`
- `CMAKE_SYSTEM_FRAMEWORK_PATH`

7. Search the paths specified by the PATHS option or in the short-hand version of the command. These are typically hard-coded guesses.

On macOS the `CMAKE_FIND_FRAMEWORK` and `CMAKE_FIND_APPBUNDLE` variables determine the order of preference between Apple-style and unix-style package components.

The CMake variable `CMAKE_FIND_ROOT_PATH` specifies one or more directories to be prepended to all other search directories. This effectively "re-roots" the entire search under given locations. Paths which are descendants of the `CMAKE_STAGING_PREFIX` are excluded from this re-rooting, because that variable is always a path on the host system. By default the `CMAKE_FIND_ROOT_PATH` is empty.

The `CMAKE_SYSROOT` variable can also be used to specify exactly one directory to use as a prefix. Setting `CMAKE_SYSROOT` also has other effects. See the documentation for that variable for more.

These variables are especially useful when cross-compiling to point to the root directory of the target environment and CMake will search there too. By default at first the directories listed in `CMAKE_FIND_ROOT_PATH` are searched, then the `CMAKE_SYSROOT` directory is searched, and then the non-rooted directories will be searched. The default behavior can be adjusted by setting `CMAKE_FIND_ROOT_PATH_MODE_LIBRARY`. This behavior can be manually overridden on a per-call basis using options:

- `CMAKE_FIND_ROOT_PATH_BOTH`

  Search in the order described above.

- `NO_CMAKE_FIND_ROOT_PATH`

  Do not use the `CMAKE_FIND_ROOT_PATH` variable.

- `ONLY_CMAKE_FIND_ROOT_PATH`

  Search only the re-rooted directories and directories below `CMAKE_STAGING_PREFIX`.

The default search order is designed to be most-specific to least-specific for common use cases. Projects may override the order by simply calling the command multiple times and using the `NO_*` options:

```
find_library (<VAR> NAMES name PATHS paths... NO_DEFAULT_PATH)
find_library (<VAR> NAMES name)
```

Once one of the calls succeeds the result variable will be set and stored in the cache so that no call will search again.

When more than one value is given to the `NAMES` option this command by default will consider one name at a time and search every directory for it. The `NAMES_PER_DIR` option tells this command to consider one directory at a time and search for all names in it.

Each library name given to the `NAMES` option is first considered as a library file name and then considered with platform-specific prefixes (e.g. `lib`) and suffixes (e.g. `.so`). Therefore one may specify library file names such as `libfoo.a` directly. This can be used to locate static libraries on UNIX-like systems.

If the library found is a framework, then `<VAR>` will be set to the full path to the framework `<fullPath>/A.framework`. When a full path to a framework is used as a library, CMake will use a `-framework A`, and a `-F<fullPath>` to link the framework to the target.

If the `CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX` variable is set all search paths will be tested as normal, with the suffix appended, and with all matches of `lib/` replaced with `lib${CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX}/`. This variable overrides the `FIND_LIBRARY_USE_LIB32_PATHS`, `FIND_LIBRARY_USE_LIBX32_PATHS`, and `FIND_LIBRARY_USE_LIB64_PATHS` global properties.

If the `FIND_LIBRARY_USE_LIB32_PATHS` global property is set all search paths will be tested as normal, with `32/` appended, and with all matches of `lib/` replaced with `lib32/`. This property is automatically set for the platforms that are known to need it if at least one of the languages supported by the `project()` command is enabled.

If the `FIND_LIBRARY_USE_LIBX32_PATHS` global property is set all search paths will be tested as normal, with `x32/` appended, and with all matches of `lib/` replaced with `libx32/`. This property is automatically set for the platforms that are known to need it if at least one of the languages supported by the `project()` command is enabled.

If the `FIND_LIBRARY_USE_LIB64_PATHS` global property is set all search paths will be tested as normal, with `64/` appended, and with all matches of `lib/` replaced with `lib64/`. This property is automatically set for the platforms that are known to need it if at least one of the languages supported by the `project()` command is enabled.

find_package

Contents

Find an external project, and load its settings.

Basic Signature and Module Mode

```
find_package(<PackageName> [version] [EXACT] [QUIET] [MODULE]
             [REQUIRED] [[COMPONENTS] [components...]]
             [OPTIONAL_COMPONENTS components...]
             [NO_POLICY_SCOPE])
```

Finds and loads settings from an external project. `<PackageName>_FOUND` will be set to indicate whether the package was found. When the package is found package-specific information is provided through variables and Imported Targets documented by the package itself. The `QUIET` option disables messages if the package cannot be found. The `REQUIRED` option stops processing with an error message if the package cannot be found.

A package-specific list of required components may be listed after the `COMPONENTS` option (or after the `REQUIRED` option if present). Additional optional components may be listed after `OPTIONAL_COMPONENTS`. Available components and their influence on whether a package is considered to be found are defined by the target package.

The `[version]` argument requests a version with which the package found should be compatible (format is `major[.minor[.patch[.tweak]]]`). The `EXACT` option requests that the version be matched exactly. If no `[version]` and/or component list is given to a recursive invocation inside a find-module, the corresponding arguments are forwarded automatically from the outer call (including the `EXACT` flag for `[version]`). Version support is currently provided only on a package-by-package basis (see the Version Selection section below).

See the `cmake_policy()` command documentation for discussion of the `NO_POLICY_SCOPE` option.

The command has two modes by which it searches for packages: “Module” mode and “Config” mode. The above signature selects Module mode. If no module is found the command falls back to Config mode, described below. This fall back is disabled if the `MODULE` option is given.

In Module mode, CMake searches for a file called `Find<PackageName>.cmake` in the `CMAKE_MODULE_PATH` followed by the CMake installation. If the file is found, it is read and processed by CMake. It is responsible for finding the package, checking the version, and producing any needed messages. Some find-modules provide limited or no support for versioning; check the module documentation.

Full Signature and Config Mode

User code should generally look for packages using the above basic signature. The remainder of this command documentation specifies the full command signature and details of the search process. Project maintainers wishing to provide a package to be found by this command are encouraged to read on.

The complete Config mode command signature is:

```
find_package(<PackageName> [version] [EXACT] [QUIET]
             [REQUIRED] [[COMPONENTS] [components...]]
             [CONFIG|NO_MODULE]
             [NO_POLICY_SCOPE]
             [NAMES name1 [name2 ...]]
             [CONFIGS config1 [config2 ...]]
             [HINTS path1 [path2 ... ]]
             [PATHS path1 [path2 ... ]]
             [PATH_SUFFIXES suffix1 [suffix2 ...]]
             [NO_DEFAULT_PATH]
             [NO_PACKAGE_ROOT_PATH]
             [NO_CMAKE_PATH]
             [NO_CMAKE_ENVIRONMENT_PATH]
             [NO_SYSTEM_ENVIRONMENT_PATH]
             [NO_CMAKE_PACKAGE_REGISTRY]
             [NO_CMAKE_BUILDS_PATH] # Deprecated; does nothing.
             [NO_CMAKE_SYSTEM_PATH]
             [NO_CMAKE_SYSTEM_PACKAGE_REGISTRY]
             [CMAKE_FIND_ROOT_PATH_BOTH |
              ONLY_CMAKE_FIND_ROOT_PATH |
              NO_CMAKE_FIND_ROOT_PATH])
```

The `CONFIG` option, the synonymous `NO_MODULE` option, or the use of options not specified in the basic signatureall enforce pure Config mode. In pure Config mode, the command skips Module mode search and proceeds at once with Config mode search.

Config mode search attempts to locate a configuration file provided by the package to be found. A cache entry called `<PackageName>_DIR` is created to hold the directory containing the file. By default the command searches for a package with the name `<PackageName>`. If the `NAMES` option is given the names following it are used instead of `<PackageName>`. The command searches for a file called `<PackageName>Config.cmake` or `<lower-case-package-name>-config.cmake` for each name specified. A replacement set of possible configuration file names may be given using the `CONFIGS` option. The search procedure is specified below. Once found, the configuration file is read and processed by CMake. Since the file is provided by the package it already knows the location of package contents. The full path to the configuration file is stored in the cmake variable `<PackageName>_CONFIG`.

All configuration files which have been considered by CMake while searching for an installation of the package with an appropriate version are stored in the cmake variable `<PackageName>_CONSIDERED_CONFIGS`, the associated versions in `<PackageName>_CONSIDERED_VERSIONS`.

If the package configuration file cannot be found CMake will generate an error describing the problem unless the `QUIET` argument is specified. If `REQUIRED` is specified and the package is not found a fatal error is generated and the configure step stops executing. If `<PackageName>_DIR` has been set to a directory not containing a configuration file CMake will ignore it and search from scratch.

Package maintainers providing CMake package configuration files are encouraged to name and install them such that the Search Procedure outlined below will find them without requiring use of additional options.

Version Selection

When the `[version]` argument is given Config mode will only find a version of the package that claims compatibility with the requested version (format is `major[.minor[.patch[.tweak]]]`). If the `EXACT` option is given only a version of the package claiming an exact match of the requested version may be found. CMake does not establish any convention for the meaning of version numbers. Package version numbers are checked by "version" files provided by the packages themselves. For a candidate package configuration file `<config-file>.cmake` the corresponding version file is located next to it and named either `<config-file>-version.cmake` or `<config-file>Version.cmake`. If no such version file is available then the configuration file is assumed to not be compatible with any requested version. A basic version file containing generic version matching code can be created using the `CMakePackageConfigHelpers` module. When a version file is found it is loaded to check the requested version number. The version file is loaded in a nested scope in which the following variables have been defined:

- `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 checks whether it satisfies the requested version and sets these variables:

- `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

These variables are checked by the `find_package` command to determine whether the configuration file provides an acceptable version. They are not available after the find_package call returns. If the version is acceptable the following variables are set:

- `<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

and the corresponding package configuration file is loaded. When multiple package configuration files are available whose version files claim compatibility with the version requested it is unspecified which one is chosen: unless the variable `CMAKE_FIND_PACKAGE_SORT_ORDER` is set no attempt is made to choose a highest or closest version number.

To control the order in which `find_package` checks for compatibility use the two variables `CMAKE_FIND_PACKAGE_SORT_ORDER` and `CMAKE_FIND_PACKAGE_SORT_DIRECTION`. For instance in order to select the highest version one can set:

```
SET(CMAKE_FIND_PACKAGE_SORT_ORDER NATURAL)
SET(CMAKE_FIND_PACKAGE_SORT_DIRECTION DEC)
```

before calling `find_package`.

Search Procedure

CMake constructs a set of possible installation prefixes for the package. Under each prefix several directories are searched for a configuration file. The tables below show the directories searched. Each entry is meant for installation trees following Windows (W), UNIX (U), or Apple (A) conventions:

```
<prefix>/                                                (W)
<prefix>/(cmake|CMake)/                                  (W)
<prefix>/<name>*/                                        (W)
<prefix>/<name>*/(cmake|CMake)/                          (W)
<prefix>/(lib/<arch>|lib*|share)/cmake/<name>*/          (U)
<prefix>/(lib/<arch>|lib*|share)/<name>*/                (U)
<prefix>/(lib/<arch>|lib*|share)/<name>*/(cmake|CMake)/  (U)
<prefix>/<name>*/(lib/<arch>|lib*|share)/cmake/<name>*/  (W/U)
<prefix>/<name>*/(lib/<arch>|lib*|share)/<name>*/        (W/U)
<prefix>/<name>*/(lib/<arch>|lib*|share)/<name>*/(cmake|CMake)/ (W/U)
```

On systems supporting macOS Frameworks and Application Bundles the following directories are searched for frameworks or bundles containing a configuration file:

```
<prefix>/<name>.framework/Resources/                   (A)
<prefix>/<name>.framework/Resources/CMake/             (A)
<prefix>/<name>.framework/Versions/*/Resources/        (A)
<prefix>/<name>.framework/Versions/*/Resources/CMake/  (A)
<prefix>/<name>.app/Contents/Resources/                (A)
<prefix>/<name>.app/Contents/Resources/CMake/          (A)
```

In all cases the `<name>` is treated as case-insensitive and corresponds to any of the names specified (`<PackageName>` or names given by `NAMES`).

Paths with `lib/<arch>` are enabled if the `CMAKE_LIBRARY_ARCHITECTURE` variable is set. `lib*` includes one or more of the values `lib64`, `lib32`, `libx32` or `lib` (searched in that order).

- Paths with `lib64` are searched on 64 bit platforms if the `FIND_LIBRARY_USE_LIB64_PATHS` property is set to `TRUE`.
- Paths with `lib32` are searched on 32 bit platforms if the `FIND_LIBRARY_USE_LIB32_PATHS` property is set to `TRUE`.
- Paths with `libx32` are searched on platforms using the x32 ABI if the `FIND_LIBRARY_USE_LIBX32_PATHS` property is set to `TRUE`.
- The `lib` path is always searched.

If `PATH_SUFFIXES` is specified, the suffixes are appended to each (W) or (U) directory entry one-by-one.

This set of directories is intended to work in cooperation with projects that provide configuration files in their installation trees. Directories above marked with (W) are intended for installations on Windows where the prefix may point at the top of an application's installation directory. Those marked with (U) are intended for installations on UNIX platforms where the prefix is shared by multiple packages. This is merely a convention, so all (W) and (U) directories are still searched on all platforms. Directories marked with (A) are intended for installations on Apple platforms. The `CMAKE_FIND_FRAMEWORK` and `CMAKE_FIND_APPBUNDLE` variables determine the order of preference.

The set of installation prefixes is constructed using the following steps. If `NO_DEFAULT_PATH` is specified all `NO_*` options are enabled.

1. Search paths specified in the `<PackageName>_ROOT` CMake variable and the `<PackageName>_ROOT` environment variable, where `<PackageName>` is the package to be found. The package root variables are maintained as a stack so if called from within a find module, root paths from the parent's find module will also be searched after paths for the current package. This can be skipped if `NO_PACKAGE_ROOT_PATH` is passed. See policy `CMP0074`.
2. Search paths specified in cmake-specific cache variables. These are intended to be used on the command line with a `-DVAR=value`. The values are interpreted as ;-lists. This can be skipped if `NO_CMAKE_PATH` is passed:

   ```
   CMAKE_PREFIX_PATH
   CMAKE_FRAMEWORK_PATH
   CMAKE_APPBUNDLE_PATH
   ```

3. Search paths specified in cmake-specific environment variables. These are intended to be set in the user's shell configuration, and therefore use the host's native path separator (`;` on Windows and `:` on UNIX). This can be skipped if `NO_CMAKE_ENVIRONMENT_PATH` is passed:

   ```
   <PackageName>_DIR
   CMAKE_PREFIX_PATH
   CMAKE_FRAMEWORK_PATH
   CMAKE_APPBUNDLE_PATH
   ```

4. Search paths specified by the `HINTS` option. These should be paths computed by system introspection, such as a hint provided by the location of another item already found. Hard-coded guesses should be specified with the `PATHS` option.
5. Search the standard system environment variables. This can be skipped if `NO_SYSTEM_ENVIRONMENT_PATH` is passed. Path entries ending in `/bin` or `/sbin` are automatically converted to their parent directories:

   ```
   PATH
   ```

6. Search paths stored in the CMake User Package Registry. This can be skipped if `NO_CMAKE_PACKAGE_REGISTRY` is passed or by setting the `CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY` to `TRUE`. See the `cmake-packages(7)` manual for details on the user package registry.
7. Search cmake variables defined in the Platform files for the current system. This can be skipped if `NO_CMAKE_SYSTEM_PATH` is passed:

   ```
   CMAKE_SYSTEM_PREFIX_PATH
   CMAKE_SYSTEM_FRAMEWORK_PATH
   CMAKE_SYSTEM_APPBUNDLE_PATH
   ```

8. Search paths stored in the CMake System Package Registry. This can be skipped if `NO_CMAKE_SYSTEM_PACKAGE_REGISTRY` is passed or by setting the `CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY` to `TRUE`. See the `cmake-packages(7)` manual for details on the system package registry.
9. Search paths specified by the `PATHS` option. These are typically hard-coded guesses.

The CMake variable `CMAKE_FIND_ROOT_PATH` specifies one or more directories to be prepended to all other search directories. This effectively "re-roots" the entire search under given locations. Paths which are descendants of the `CMAKE_STAGING_PREFIX` are excluded from this re-rooting, because that variable is always a path on the host system. By default the `CMAKE_FIND_ROOT_PATH` is empty.

The `CMAKE_SYSROOT` variable can also be used to specify exactly one directory to use as a prefix. Setting `CMAKE_SYSROOT` also has other effects. See the documentation for that variable for more.

These variables are especially useful when cross-compiling to point to the root directory of the target environment and CMake will search there too. By default at first the directories listed in `CMAKE_FIND_ROOT_PATH` are searched, then the `CMAKE_SYSROOT` directory is searched, and then the non-rooted directories will be searched. The default behavior can be adjusted by setting `CMAKE_FIND_ROOT_PATH_MODE_PACKAGE`. This behavior can be manually overridden on a per-call basis using options:

- `CMAKE_FIND_ROOT_PATH_BOTH`

  Search in the order described above.

- `NO_CMAKE_FIND_ROOT_PATH`

  Do not use the `CMAKE_FIND_ROOT_PATH` variable.

- `ONLY_CMAKE_FIND_ROOT_PATH`

  Search only the re-rooted directories and directories below `CMAKE_STAGING_PREFIX`.

The default search order is designed to be most-specific to least-specific for common use cases. Projects may override the order by simply calling the command multiple times and using the `NO_*` options:

```
find_package (<PackageName> PATHS paths... NO_DEFAULT_PATH)
find_package (<PackageName>)
```

Once one of the calls succeeds the result variable will be set and stored in the cache so that no call will search again.

Every non-REQUIRED `find_package` call can be disabled by setting the `CMAKE_DISABLE_FIND_PACKAGE_` variable to `TRUE`.

Package File Interface Variables

When loading a find module or package configuration file `find_package` defines variables to provide information about the call arguments (and restores their original state before returning):

- `CMAKE_FIND_PACKAGE_NAME`

  the `<PackageName>` which is searched for

- `<PackageName>_FIND_REQUIRED`

  true if `REQUIRED` option was given

- `<PackageName>_FIND_QUIETLY`

  true if `QUIET` option was given

- `<PackageName>_FIND_VERSION`

full requested version string

- `<PackageName>_FIND_VERSION_MAJOR`
  major version if requested, else 0
- `<PackageName>_FIND_VERSION_MINOR`
  minor version if requested, else 0
- `<PackageName>_FIND_VERSION_PATCH`
  patch version if requested, else 0
- `<PackageName>_FIND_VERSION_TWEAK`
  tweak version if requested, else 0
- `<PackageName>_FIND_VERSION_COUNT`
  number of version components, 0 to 4
- `<PackageName>_FIND_VERSION_EXACT`
  true if `EXACT` option was given
- `<PackageName>_FIND_COMPONENTS`
  list of requested components
- `<PackageName>_FIND_REQUIRED_<c>`
  true if component `<c>` is required, false if component `<c>` is optional

In Module mode the loaded find module is responsible to honor the request detailed by these variables; see the find module for details. In Config mode `find_package` handles `REQUIRED`, `QUIET`, and `[version]` options automatically but leaves it to the package configuration file to handle components in a way that makes sense for the package. The package configuration file may set `<PackageName>_FOUND` to false to tell `find_package` that component requirements are not satisfied.

find_path

A short-hand signature is:

```
find_path (<VAR> name1 [path1 path2 ...])
```

The general signature is:

```
find_path (
          <VAR>
          name | NAMES name1 [name2 ...]
          [HINTS path1 [path2 ... ENV var]]
          [PATHS path1 [path2 ... ENV var]]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [DOC "cache documentation string"]
          [NO_DEFAULT_PATH]
          [NO_PACKAGE_ROOT_PATH]
          [NO_CMAKE_PATH]
          [NO_CMAKE_ENVIRONMENT_PATH]
```

```
            [NO_SYSTEM_ENVIRONMENT_PATH]
            [NO_CMAKE_SYSTEM_PATH]
            [CMAKE_FIND_ROOT_PATH_BOTH |
             ONLY_CMAKE_FIND_ROOT_PATH |
             NO_CMAKE_FIND_ROOT_PATH]
           )
```

This command is used to find a directory containing the named file. A cache entry named by `<VAR>` is created to store the result of this command. If the file in a directory is found the result is stored in the variable and the search will not be repeated unless the variable is cleared. If nothing is found, the result will be `<VAR>-NOTFOUND`, and the search will be attempted again the next time find_path is invoked with the same variable.

Options include:

- `NAMES`

  Specify one or more possible names for the file in a directory.When using this to specify names with and without a version suffix, we recommend specifying the unversioned name first so that locally-built packages can be found before those provided by distributions.

- `HINTS`, `PATHS`

  Specify directories to search in addition to the default locations. The `ENV var` sub-option reads paths from a system environment variable.

- `PATH_SUFFIXES`

  Specify additional subdirectories to check below each directory location otherwise considered.

- `DOC`

  Specify the documentation string for the `<VAR>` cache entry.

If `NO_DEFAULT_PATH` is specified, then no additional paths are added to the search. If `NO_DEFAULT_PATH` is not specified, the search process is as follows:

1. If called from within a find module loaded by

   `find_package(<PackageName>)`

   , search prefixes unique to the current package being found. Specifically look in the

   `<PackageName>_ROOT`

   CMake variable and the

   `<PackageName>_ROOT`

   environment variable. The package root variables are maintained as a stack so if called from nested find modules, root paths from the parent's find module will be searched after paths from the current module, i.e.

   ```
     <CurrentPackage>_ROOT
   ```

,

```
ENV{<CurrentPackage>_ROOT}
```

,

```
<ParentPackage>_ROOT
```

,

```
ENV{<ParentPackage>_ROOT}
```

, etc. This can be skipped if

```
NO_PACKAGE_ROOT_PATH
```

is passed. See policy

`CMP0074`

.

- `<prefix>/include/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/include` for each `<prefix>` in the`_ROOT` CMake variable and the `_ROOT` environment variable if called from within a find module loaded by `find_package()`

2. Search paths specified in cmake-specific cache variables. These are intended to be used on the command line with a

```
-DVAR=value
```

. The values are interpreted as

;-lists

. This can be skipped if

```
NO_CMAKE_PATH
```

is passed.

- `<prefix>/include/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/include` for each `<prefix>` in `CMAKE_PREFIX_PATH`
- `CMAKE_INCLUDE_PATH`
- `CMAKE_FRAMEWORK_PATH`

3. Search paths specified in cmake-specific environment variables. These are intended to be set in the user's shell configuration, and therefore use the host's native path separator (

```
;
```

on Windows and

```
:
```

on UNIX). This can be skipped if

```
NO_CMAKE_ENVIRONMENT_PATH
```

is passed.

- `<prefix>/include/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/include` for each `<prefix>` in `CMAKE_PREFIX_PATH`
- `CMAKE_INCLUDE_PATH`
- `CMAKE_FRAMEWORK_PATH`

4. Search the paths specified by the `HINTS` option. These should be paths computed by system introspection, such as a hint provided by the location of another item already found. Hard-coded guesses should be specified with the `PATHS` option.
5. Search the standard system environment variables. This can be skipped if

```
NO_SYSTEM_ENVIRONMENT_PATH
```

is an argument.

- Directories in `INCLUDE`. On Windows hosts: `<prefix>/include/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/include` for each `<prefix>/[s]bin` in `PATH`, and `<entry>/include` for other entries in `PATH`, and the directories in `PATH` itself.

6. Search cmake variables defined in the Platform files for the current system. This can be skipped if

```
NO_CMAKE_SYSTEM_PATH
```

is passed.

- `<prefix>/include/<arch>` if `CMAKE_LIBRARY_ARCHITECTURE` is set, and `<prefix>/include` for each `<prefix>` in `CMAKE_SYSTEM_PREFIX_PATH`
- `CMAKE_SYSTEM_INCLUDE_PATH`
- `CMAKE_SYSTEM_FRAMEWORK_PATH`

7. Search the paths specified by the PATHS option or in the short-hand version of the command. These are typically hard-coded guesses.

On macOS the `CMAKE_FIND_FRAMEWORK` and `CMAKE_FIND_APPBUNDLE` variables determine the order of preference between Apple-style and unix-style package components.

The CMake variable `CMAKE_FIND_ROOT_PATH` specifies one or more directories to be prepended to all other search directories. This effectively "re-roots" the entire search under given locations. Paths which are descendants of the `CMAKE_STAGING_PREFIX` are excluded from this re-rooting, because that variable is always a path on the host system. By default the `CMAKE_FIND_ROOT_PATH` is empty.

The `CMAKE_SYSROOT` variable can also be used to specify exactly one directory to use as a prefix. Setting `CMAKE_SYSROOT` also has other effects. See the documentation for that variable for more.

These variables are especially useful when cross-compiling to point to the root directory of the target environment and CMake will search there too. By default at first the directories listed in `CMAKE_FIND_ROOT_PATH` are searched, then the `CMAKE_SYSROOT` directory is searched, and then the non-rooted directories will be searched. The default behavior can be adjusted by setting `CMAKE_FIND_ROOT_PATH_MODE_INCLUDE`. This behavior can be manually overridden on a per-call basis using options:

- `CMAKE_FIND_ROOT_PATH_BOTH`

  Search in the order described above.

- `NO_CMAKE_FIND_ROOT_PATH`

  Do not use the `CMAKE_FIND_ROOT_PATH` variable.

- `ONLY_CMAKE_FIND_ROOT_PATH`

  Search only the re-rooted directories and directories below `CMAKE_STAGING_PREFIX`.

The default search order is designed to be most-specific to least-specific for common use cases. Projects may override the order by simply calling the command multiple times and using the `NO_*` options:

```
find_path (<VAR> NAMES name PATHS paths... NO_DEFAULT_PATH)
find_path (<VAR> NAMES name)
```

Once one of the calls succeeds the result variable will be set and stored in the cache so that no call will search again.

When searching for frameworks, if the file is specified as `A/b.h`, then the framework search will look for `A.framework/Headers/b.h`. If that is found the path will be set to the path to the framework. CMake will convert this to the correct `-F` option to include the file.

find_program

A short-hand signature is:

```
find_program (<VAR> name1 [path1 path2 ...])
```

The general signature is:

```
find_program (
          <VAR>
          name | NAMES name1 [name2 ...] [NAMES_PER_DIR]
          [HINTS path1 [path2 ... ENV var]]
          [PATHS path1 [path2 ... ENV var]]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [DOC "cache documentation string"]
          [NO_DEFAULT_PATH]
          [NO_PACKAGE_ROOT_PATH]
          [NO_CMAKE_PATH]
          [NO_CMAKE_ENVIRONMENT_PATH]
          [NO_SYSTEM_ENVIRONMENT_PATH]
          [NO_CMAKE_SYSTEM_PATH]
```

```
            [CMAKE_FIND_ROOT_PATH_BOTH |
             ONLY_CMAKE_FIND_ROOT_PATH |
             NO_CMAKE_FIND_ROOT_PATH]
           )
```

This command is used to find a program. A cache entry named by `<VAR>` is created to store the result of this command. If the program is found the result is stored in the variable and the search will not be repeated unless the variable is cleared. If nothing is found, the result will be `<VAR>-NOTFOUND`, and the search will be attempted again the next time find_program is invoked with the same variable.

Options include:

- `NAMES`

  Specify one or more possible names for the program.When using this to specify names with and without a version suffix, we recommend specifying the unversioned name first so that locally-built packages can be found before those provided by distributions.

- `HINTS`, `PATHS`

  Specify directories to search in addition to the default locations. The `ENV var` sub-option reads paths from a system environment variable.

- `PATH_SUFFIXES`

  Specify additional subdirectories to check below each directory location otherwise considered.

- `DOC`

  Specify the documentation string for the `<VAR>` cache entry.

If `NO_DEFAULT_PATH` is specified, then no additional paths are added to the search. If `NO_DEFAULT_PATH` is not specified, the search process is as follows:

1. If called from within a find module loaded by

   `find_package(<PackageName>)`

   , search prefixes unique to the current package being found. Specifically look in the

   `<PackageName>_ROOT`

   CMake variable and the

   `<PackageName>_ROOT`

   environment variable. The package root variables are maintained as a stack so if called from nested find modules, root paths from the parent's find module will be searched after paths from the current module, i.e.

   ```
   <CurrentPackage>_ROOT
   ```

   ,

```
ENV{<CurrentPackage>_ROOT}
```

,

```
<ParentPackage>_ROOT
```

,

```
ENV{<ParentPackage>_ROOT}
```

, etc. This can be skipped if

```
NO_PACKAGE_ROOT_PATH
```

is passed. See policy

`CMP0074`

.

- `<prefix>/[s]bin` for each `<prefix>` in the `_ROOT` CMake variable and the `_ROOT` environment variable if called from within a find module loaded by `find_package()`

2. Search paths specified in cmake-specific cache variables. These are intended to be used on the command line with a

```
-DVAR=value
```

. The values are interpreted as

;-lists

. This can be skipped if

```
NO_CMAKE_PATH
```

is passed.

- `<prefix>/[s]bin` for each `<prefix>` in `CMAKE_PREFIX_PATH`
- `CMAKE_PROGRAM_PATH`
- `CMAKE_APPBUNDLE_PATH`

3. Search paths specified in cmake-specific environment variables. These are intended to be set in the user's shell configuration, and therefore use the host's native path separator (

```
;
```

on Windows and

```
:
```

on UNIX). This can be skipped if

```
NO_CMAKE_ENVIRONMENT_PATH
```

is passed.

- `<prefix>/[s]bin` for each `<prefix>` in `CMAKE_PREFIX_PATH`
- `CMAKE_PROGRAM_PATH`
- `CMAKE_APPBUNDLE_PATH`

4. Search the paths specified by the `HINTS` option. These should be paths computed by system introspection, such as a hint provided by the location of another item already found. Hard-coded guesses should be specified with the `PATHS` option.
5. Search the standard system environment variables. This can be skipped if

```
NO_SYSTEM_ENVIRONMENT_PATH
```

is an argument.

- `PATH`

6. Search cmake variables defined in the Platform files for the current system. This can be skipped if

```
NO_CMAKE_SYSTEM_PATH
```

is passed.

- `<prefix>/[s]bin` for each `<prefix>` in `CMAKE_SYSTEM_PREFIX_PATH`
- `CMAKE_SYSTEM_PROGRAM_PATH`
- `CMAKE_SYSTEM_APPBUNDLE_PATH`

7. Search the paths specified by the PATHS option or in the short-hand version of the command. These are typically hard-coded guesses.

On macOS the `CMAKE_FIND_FRAMEWORK` and `CMAKE_FIND_APPBUNDLE` variables determine the order of preference between Apple-style and unix-style package components.

The CMake variable `CMAKE_FIND_ROOT_PATH` specifies one or more directories to be prepended to all other search directories. This effectively "re-roots" the entire search under given locations. Paths which are descendants of the `CMAKE_STAGING_PREFIX` are excluded from this re-rooting, because that variable is always a path on the host system. By default the `CMAKE_FIND_ROOT_PATH` is empty.

The `CMAKE_SYSROOT` variable can also be used to specify exactly one directory to use as a prefix. Setting `CMAKE_SYSROOT` also has other effects. See the documentation for that variable for more.

These variables are especially useful when cross-compiling to point to the root directory of the target environment and CMake will search there too. By default at first the directories listed in `CMAKE_FIND_ROOT_PATH` are searched, then the `CMAKE_SYSROOT` directory is searched, and then the non-rooted directories will be searched. The default behavior can be adjusted by setting `CMAKE_FIND_ROOT_PATH_MODE_PROGRAM`. This behavior can be manually overridden on a per-call basis using options:

- `CMAKE_FIND_ROOT_PATH_BOTH`

Search in the order described above.

- `NO_CMAKE_FIND_ROOT_PATH`

  Do not use the `CMAKE_FIND_ROOT_PATH` variable.

- `ONLY_CMAKE_FIND_ROOT_PATH`

  Search only the re-rooted directories and directories below `CMAKE_STAGING_PREFIX`.

The default search order is designed to be most-specific to least-specific for common use cases. Projects may override the order by simply calling the command multiple times and using the `NO_*` options:

```
find_program (<VAR> NAMES name PATHS paths... NO_DEFAULT_PATH)
find_program (<VAR> NAMES name)
```

Once one of the calls succeeds the result variable will be set and stored in the cache so that no call will search again.

When more than one value is given to the `NAMES` option this command by default will consider one name at a time and search every directory for it. The `NAMES_PER_DIR` option tells this command to consider one directory at a time and search for all names in it.

foreach

Evaluate a group of commands for each value in a list.

```
foreach(loop_var arg1 arg2 ...)
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endforeach(loop_var)
```

All commands between foreach and the matching endforeach are recorded without being invoked. Once the endforeach is evaluated, the recorded list of commands is invoked once for each argument listed in the original foreach command. Before each iteration of the loop `${loop_var}` will be set as a variable with the current value in the list.

```
foreach(loop_var RANGE total)
foreach(loop_var RANGE start stop [step])
```

Foreach can also iterate over a generated range of numbers. There are three types of this iteration:

- When specifying single number, the range will have elements [0, ... to “total”] (inclusive).
- When specifying two numbers, the range will have elements from the first number to the second number (inclusive).
- The third optional number is the increment used to iterate from the first number to the second number (inclusive).

```
foreach(loop_var IN [LISTS [list1 [...]]]
                    [ITEMS [item1 [...]]])
```

Iterates over a precise list of items. The `LISTS` option names list-valued variables to be traversed, including empty elements (an empty string is a zero-length list). (Note macro arguments are not variables.) The `ITEMS` option ends argument parsing and includes all arguments following it in the iteration.

function

Start recording a function for later invocation as a command:

```
function(<name> [arg1 [arg2 [arg3 ...]]])
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endfunction(<name>)
```

Define a function named `<name>` that takes arguments named `arg1`, `arg2`, `arg3`, (...). Commands listed after function, but before the matching `endfunction()`, are not invoked until the function is invoked. When it is invoked, the commands recorded in the function are first modified by replacing formal parameters (`${arg1}`) with the arguments passed, and then invoked as normal commands. In addition to referencing the formal parameters you can reference the `ARGC` variable which will be set to the number of arguments passed into the function as well as `ARGV0`, `ARGV1`, `ARGV2`, ... which will have the actual values of the arguments passed in. This facilitates creating functions with optional arguments. Additionally `ARGV` holds the list of all arguments given to the function and `ARGN` holds the list of arguments past the last expected argument. Referencing to `ARGV#` arguments beyond `ARGC` have undefined behavior. Checking that `ARGC` is greater than `#` is the only way to ensure that `ARGV#` was passed to the function as an extra argument.

A function opens a new scope: see `set(var PARENT_SCOPE)` for details.

See the `cmake_policy()` command documentation for the behavior of policies inside functions.

get_cmake_property

Get a global property of the CMake instance.

```
get_cmake_property(VAR property)
```

Get a global property from the CMake instance. The value of the property is stored in the variable `VAR`. If the property is not found, `VAR` will be set to "NOTFOUND". See the `cmake-properties(7)` manual for available properties.

See also the `get_property()` command `GLOBAL` option.

In addition to global properties, this command (for historical reasons) also supports the `VARIABLES` and `MACROS` directory properties. It also supports a special `COMPONENTS` global property that lists the components given to the `install()` command.

get_directory_property¶

Get a property of `DIRECTORY` scope.

```
get_directory_property(<variable> [DIRECTORY <dir>] <prop-name>)
```

Store a property of directory scope in the named `<variable>`. The `DIRECTORY` argument specifies another directory from which to retrieve the property value instead of the current directory. The specified directory must have already been traversed by CMake.

If the property is not defined for the nominated directory scope, an empty string is returned. In the case of `INHERITED` properties, if the property is not found for the nominated directory scope, the search will chain to a parent scope as described for the `define_property()` command.

```
get_directory_property(<variable> [DIRECTORY <dir>]
                       DEFINITION <var-name>)
```

Get a variable definition from a directory. This form is useful to get a variable definition from another directory.

See also the more general `get_property()` command.

get_filename_component

Get a specific component of a full filename.

```
get_filename_component(<VAR> <FileName> <COMP> [CACHE])
```

Set `<VAR>` to a component of `<FileName>`, where `<COMP>` is one of:

```
DIRECTORY = Directory without file name
NAME      = File name without directory
EXT       = File name longest extension (.b.c from d/a.b.c)
NAME_WE   = File name without directory or longest extension
PATH      = Legacy alias for DIRECTORY (use for CMake <= 2.8.11)
```

Paths are returned with forward slashes and have no trailing slashes. The longest file extension is always considered. If the optional `CACHE` argument is specified, the result variable is added to the cache.

```
get_filename_component(<VAR> <FileName>
                       <COMP> [BASE_DIR <BASE_DIR>]
                       [CACHE])
```

Set `<VAR>` to the absolute path of `<FileName>`, where `<COMP>` is one of:

```
ABSOLUTE  = Full path to file
REALPATH  = Full path to existing file with symlinks resolved
```

If the provided `<FileName>` is a relative path, it is evaluated relative to the given base directory `<BASE_DIR>`. If no base directory is provided, the default base directory will be `CMAKE_CURRENT_SOURCE_DIR`.

Paths are returned with forward slashes and have no trailing slashes. If the optional `CACHE` argument is specified, the result variable is added to the cache.

```
get_filename_component(<VAR> <FileName>
                       PROGRAM [PROGRAM_ARGS <ARG_VAR>]
                       [CACHE])
```

The program in `<FileName>` will be found in the system search path or left as a full path. If `PROGRAM_ARGS` is present with `PROGRAM`, then any command-line arguments present in the `<FileName>` string are split from the program name and stored in `<ARG_VAR>`. This is used to separate a program name from its arguments in a command line string.

get_property

Get a property.

```
get_property(<variable>
             <GLOBAL             |
              DIRECTORY [dir]    |
              TARGET    <target> |
              SOURCE    <source> |
              INSTALL   <file>   |
              TEST      <test>   |
              CACHE     <entry>  |
              VARIABLE>
             PROPERTY <name>
             [SET | DEFINED | BRIEF_DOCS | FULL_DOCS])
```

Get one property from one object in a scope. The first argument specifies the variable in which to store the result. The second argument determines the scope from which to get the property. It must be one of the following:

- `GLOBAL`

  Scope is unique and does not accept a name.

- `DIRECTORY`

  Scope defaults to the current directory but another directory (already processed by CMake) may be named by full or relative path.

- `TARGET`

  Scope must name one existing target.

- `SOURCE`

  Scope must name one source file.

- `INSTALL`

Scope must name one installed file path.

- `TEST`

  Scope must name one existing test.

- `CACHE`

  Scope must name one cache entry.

- `VARIABLE`

  Scope is unique and does not accept a name.

The required `PROPERTY` option is immediately followed by the name of the property to get. If the property is not set an empty value is returned, although some properties support inheriting from a parent scope if defined to behave that way (see `define_property()`).

If the `SET` option is given the variable is set to a boolean value indicating whether the property has been set. If the `DEFINED` option is given the variable is set to a boolean value indicating whether the property has been defined such as with the `define_property()` command. If `BRIEF_DOCS` or `FULL_DOCS` is given then the variable is set to a string containing documentation for the requested property. If documentation is requested for a property that has not been defined `NOTFOUND` is returned.

if

Conditionally execute a group of commands.

```
if(expression)
  # then section.
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  #...
elseif(expression2)
  # elseif section.
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  #...
else(expression)
  # else section.
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  #...
endif(expression)
```

Evaluates the given expression. If the result is true, the commands in the THEN section are invoked. Otherwise, the commands in the else section are invoked. The elseif and else sections are optional. You may have multiple elseif clauses. Note that the expression in the else and endif clause is optional. Long expressions can be used and there is a traditional order of precedence. Parenthetical expressions are evaluated first followed by unary tests such as `EXISTS`, `COMMAND`, and `DEFINED`. Then any binary tests such as `EQUAL`, `LESS`, `LESS_EQUAL`, `GREATER`, `GREATER_EQUAL`, `STREQUAL`, `STRLESS`, `STRLESS_EQUAL`, `STRGREATER`, `STRGREATER_EQUAL`, `VERSION_EQUAL`, `VERSION_LESS`, `VERSION_LESS_EQUAL`, `VERSION_GREATER`,

`VERSION_GREATER_EQUAL`, and `MATCHES` will be evaluated. Then boolean `NOT` operators and finally boolean `AND` and then `OR` operators will be evaluated.

Possible expressions are:

- `if(<constant>)`

  True if the constant is `1`, `ON`, `YES`, `TRUE`, `Y`, or a non-zero number. False if the constant is `0`, `OFF`, `NO`, `FALSE`, `N`, `IGNORE`, `NOTFOUND`, the empty string, or ends in the suffix `-NOTFOUND`. Named boolean constants are case-insensitive. If the argument is not one of these specific constants, it is treated as a variable or string and the following signature is used.

- `if(<variable|string>)`

  True if given a variable that is defined to a value that is not a false constant. False otherwise. (Note macro arguments are not variables.)

- `if(NOT <expression>)`

  True if the expression is not true.

- `if(<expr1> AND <expr2>)`

  True if both expressions would be considered true individually.

- `if(<expr1> OR <expr2>)`

  True if either expression would be considered true individually.

- `if(COMMAND command-name)`

  True if the given name is a command, macro or function that can be invoked.

- `if(POLICY policy-id)`

  True if the given name is an existing policy (of the form `CMP<NNNN>`).

- `if(TARGET target-name)`

  True if the given name is an existing logical target name created by a call to the `add_executable()`, `add_library()`, or `add_custom_target()` command that has already been invoked (in any directory).

- `if(TEST test-name)`

  True if the given name is an existing test name created by the `add_test()` command.

- `if(EXISTS path-to-file-or-directory)`

  True if the named file or directory exists. Behavior is well-defined only for full paths.

- `if(file1 IS_NEWER_THAN file2)`

  True if `file1` is newer than `file2` or if one of the two files doesn't exist. Behavior is well-defined only for full paths. If the file time stamps are exactly the same, an `IS_NEWER_THAN` comparison returns true, so that any dependent build operations will occur in the event of a tie. This includes the case of passing the same file name for both file1 and file2.

- `if(IS_DIRECTORY path-to-directory)`

  True if the given name is a directory. Behavior is well-defined only for full paths.

- `if(IS_SYMLINK file-name)`

  True if the given name is a symbolic link. Behavior is well-defined only for full paths.

- `if(IS_ABSOLUTE path)`

  True if the given path is an absolute path.

- `if(<variable|string> MATCHES regex)`

  True if the given string or variable's value matches the given regular expression. See Regex Specification for regex format. `()` groups are captured in `CMAKE_MATCH_` variables.

- `if(<variable|string> LESS <variable|string>)`

  True if the given string or variable's value is a valid number and less than that on the right.

- `if(<variable|string> GREATER <variable|string>)`

  True if the given string or variable's value is a valid number and greater than that on the right.

- `if(<variable|string> EQUAL <variable|string>)`

  True if the given string or variable's value is a valid number and equal to that on the right.

- `if(<variable|string> LESS_EQUAL <variable|string>)`

  True if the given string or variable's value is a valid number and less than or equal to that on the right.

- `if(<variable|string> GREATER_EQUAL <variable|string>)`

  True if the given string or variable's value is a valid number and greater than or equal to that on the right.

- `if(<variable|string> STRLESS <variable|string>)`

  True if the given string or variable's value is lexicographically less than the string or variable on the right.

- `if(<variable|string> STRGREATER <variable|string>)`

  True if the given string or variable's value is lexicographically greater than the string or variable on the right.

- `if(<variable|string> STREQUAL <variable|string>)`

  True if the given string or variable's value is lexicographically equal to the string or variable on the right.

- `if(<variable|string> STRLESS_EQUAL <variable|string>)`

  True if the given string or variable's value is lexicographically less than or equal to the string or variable on the right.

- `if(<variable|string> STRGREATER_EQUAL <variable|string>)`

  True if the given string or variable's value is lexicographically greater than or equal to the string or variable on the right.

- `if(<variable|string> VERSION_LESS <variable|string>)`

  Component-wise integer version number comparison (version format is `major[.minor[.patch[.tweak]]]`, omitted components are treated as zero). Any non-integer version component or non-integer trailing part of a version component effectively truncates the string at that point.

- `if(<variable|string> VERSION_GREATER <variable|string>)`

  Component-wise integer version number comparison (version format is `major[.minor[.patch[.tweak]]]`, omitted components are treated as zero). Any non-integer version component or non-integer trailing part of a version component effectively truncates the string at that point.

- `if(<variable|string> VERSION_EQUAL <variable|string>)`

  Component-wise integer version number comparison (version format is `major[.minor[.patch[.tweak]]]`, omitted components are treated as zero). Any non-integer version component or non-integer trailing part of a version component effectively truncates the string at that point.

- `if(<variable|string> VERSION_LESS_EQUAL <variable|string>)`

  Component-wise integer version number comparison (version format is `major[.minor[.patch[.tweak]]]`, omitted components are treated as zero). Any non-integer version component or non-integer trailing part of a version component effectively truncates the string at that point.

- `if(<variable|string> VERSION_GREATER_EQUAL <variable|string>)`

  Component-wise integer version number comparison (version format is `major[.minor[.patch[.tweak]]]`, omitted components are treated as zero). Any non-integer version component or non-integer trailing part of a version component effectively truncates the string at that point.

- `if(<variable|string> IN_LIST <variable>)`

  True if the given element is contained in the named list variable.

- `if(DEFINED <variable>)`

  True if the given variable is defined. It does not matter if the variable is true or false just if it has been set. (Note macro arguments are not variables.)

- `if((expression) AND (expression OR (expression)))`

  The expressions inside the parenthesis are evaluated first and then the remaining expression is evaluated as in the previous examples. Where there are nested parenthesis the innermost are evaluated as part of evaluating the expression that contains them.

The if command was written very early in CMake's history, predating the `${}` variable evaluation syntax, and for convenience evaluates variables named by its arguments as shown in the above signatures. Note that normal variable evaluation with `${}` applies before the if command even receives the arguments. Therefore code like:

```
set(var1 OFF)
set(var2 "var1")
if(${var2})
```

appears to the if command as:

```
if(var1)
```

and is evaluated according to the `if(<variable>)` case documented above. The result is `OFF` which is false. However, if we remove the `${}` from the example then the command sees:

```
if(var2)
```

which is true because `var2` is defined to "var1" which is not a false constant.

Automatic evaluation applies in the other cases whenever the above-documented signature accepts `<variable|string>`:

- The left hand argument to `MATCHES` is first checked to see if it is a defined variable, if so the variable's value is used, otherwise the original value is used.
- If the left hand argument to `MATCHES` is missing it returns false without error
- Both left and right hand arguments to `LESS`, `GREATER`, `EQUAL`, `LESS_EQUAL`, and `GREATER_EQUAL`, are independently tested to see if they are defined variables, if so their defined values are used otherwise the original value is used.
- Both left and right hand arguments to `STRLESS`, `STRGREATER`, `STREQUAL`, `STRLESS_EQUAL`, and `STRGREATER_EQUAL` are independently tested to see if they are defined variables, if so their defined values are used otherwise the original value is used.
- Both left and right hand arguments to `VERSION_LESS`, `VERSION_GREATER`, `VERSION_EQUAL`, `VERSION_LESS_EQUAL`, and `VERSION_GREATER_EQUAL` are independently tested to see if they are defined variables, if so their defined values are used otherwise the original value is used.
- The right hand argument to `NOT` is tested to see if it is a boolean constant, if so the value is used, otherwise it is assumed to be a variable and it is dereferenced.
- The left and right hand arguments to `AND` and `OR` are independently tested to see if they are boolean constants, if so they are used as such, otherwise they are assumed to be variables and are dereferenced.

To prevent ambiguity, potential variable or keyword names can be specified in a Quoted Argument or a Bracket Argument. A quoted or bracketed variable or keyword will be interpreted as a string and not dereferenced or interpreted. See policy `CMP0054`.

include

Load and run CMake code from a file or module.

```
include(<file|module> [OPTIONAL] [RESULT_VARIABLE <VAR>]
                      [NO_POLICY_SCOPE])
```

Load and run CMake code from the file given. Variable reads and writes access the scope of the caller (dynamic scoping). If `OPTIONAL` is present, then no error is raised if the file does not exist. If `RESULT_VARIABLE` is given the variable will be set to the full filename which has been included or NOTFOUND if it failed.

If a module is specified instead of a file, the file with name `<modulename>.cmake` is searched first in `CMAKE_MODULE_PATH`, then in the CMake module directory. There is one exception to this: if the file which calls `include()` is located itself in the CMake builtin module directory, then first the CMake builtin module directory is searched and `CMAKE_MODULE_PATH` afterwards. See also policy `CMP0017`.

See the `cmake_policy()` command documentation for discussion of the `NO_POLICY_SCOPE` option.

include_guard

Provides an include guard for the file currently being processed by CMake.

```
include_guard([DIRECTORY|GLOBAL])
```

Sets up an include guard for the current CMake file (see the `CMAKE_CURRENT_LIST_FILE` variable documentation).

CMake will end its processing of the current file at the location of the `include_guard()` command if the current file has already been processed for the applicable scope (see below). This provides functionality similar to the include guards commonly used in source headers or to the `#pragma once` directive. If the current file has been processed previously for the applicable scope, the effect is as though `return()` had been called. Do not call this command from inside a function being defined within the current file.

An optional argument specifying the scope of the guard may be provided. Possible values for the option are:

- `DIRECTORY`

  The include guard applies within the current directory and below. The file will only be included once within this directory scope, but may be included again by other files outside of this directory (i.e. a parent directory or another directory not pulled in by `add_subdirectory()` or `include()` from the current file or its children).

- `GLOBAL`

  The include guard applies globally to the whole build. The current file will only be included once regardless of the scope.

If no arguments given, `include_guard` has the same scope as a variable, meaning that the include guard effect is isolated by the most recent function scope or current directory if no inner function scopes exist. In this case the command behavior is the same as:

```
if(__CURRENT_FILE_VAR__)
  return()
endif()
set(__CURRENT_FILE_VAR__ TRUE)
```

list

List operations.

Synopsis

```
Reading
  list(LENGTH <list> <out-var>)
  list(GET <list> <element index> [<index> ...] <out-var>)
  list(JOIN <list> <glue> <out-var>)
  list(SUBLIST <list> <begin> <length> <out-var>)

Search
  list(FIND <list> <value> <out-var>)

Modification
  list(APPEND <list> [<element>...])
```

```
    list(FILTER <list> {INCLUDE | EXCLUDE} REGEX <regex>)
    list(INSERT <list> <index> [<element>...])
    list(REMOVE_ITEM <list> <value>...)
    list(REMOVE_AT <list> <index>...)
    list(REMOVE_DUPLICATES <list>)
    list(TRANSFORM <list> <ACTION> [...])

  Ordering
    list(REVERSE <list>)
    list(SORT <list> [...])
```

Introduction

The list subcommands `APPEND`, `INSERT`, `FILTER`, `REMOVE_AT`, `REMOVE_ITEM`, `REMOVE_DUPLICATES`, `REVERSE` and `SORT` may create new values for the list within the current CMake variable scope. Similar to the `set()` command, the LIST command creates new variable values in the current scope, even if the list itself is actually defined in a parent scope. To propagate the results of these operations upwards, use `set()` with `PARENT_SCOPE`, `set()` with `CACHEINTERNAL`, or some other means of value propagation.

Note

A list in cmake is a `;` separated group of strings. To create a list the set command can be used. For example, `set(var a b c d e)` creates a list with `a;b;c;d;e`, and `set(var "a b c d e")` creates a string or a list with one item in it. (Note macro arguments are not variables, and therefore cannot be used in LIST commands.)

Note

When specifying index values, if `<element index>` is 0 or greater, it is indexed from the beginning of the list, with 0 representing the first list element. If `<element index>` is -1 or lesser, it is indexed from the end of the list, with -1 representing the last list element. Be careful when counting with negative indices: they do not start from 0. -0 is equivalent to 0, the first list element.

Reading

```
  list(LENGTH <list> <output variable>)
```

Returns the list's length.

```
  list(GET <list> <element index> [<element index> ...] <output variable>)
```

Returns the list of elements specified by indices from the list.

```
  list(JOIN <list> <glue> <output variable>)
```

Returns a string joining all list's elements using the glue string. To join multiple strings, which are not part of a list, use `JOIN` operator from `string()` command.

```
  list(SUBLIST <list> <begin> <length> <output variable>)
```

Returns a sublist of the given list. If `<length>` is 0, an empty list will be returned. If `<length>` is -1 or the list is smaller than `<begin>+<length>` then the remaining elements of the list starting at `<begin>` will be returned.

Search

```
list(FIND <list> <value> <output variable>)
```

Returns the index of the element specified in the list or -1 if it wasn't found.

Modification

```
list(APPEND <list> [<element> ...])
```

Appends elements to the list.

```
list(FILTER <list> <INCLUDE|EXCLUDE> REGEX <regular_expression>)
```

Includes or removes items from the list that match the mode's pattern. In `REGEX` mode, items will be matched against the given regular expression.

For more information on regular expressions see also the `string()` command.

```
list(INSERT <list> <element_index> <element> [<element> ...])
```

Inserts elements to the list to the specified location.

```
list(REMOVE_ITEM <list> <value> [<value> ...])
```

Removes the given items from the list.

```
list(REMOVE_AT <list> <index> [<index> ...])
```

Removes items at given indices from the list.

```
list(REMOVE_DUPLICATES <list>)
```

Removes duplicated items in the list.

```
list(TRANSFORM <list> <ACTION> [<SELECTOR>]
                      [OUTPUT_VARIABLE <output variable>])
```

Transforms the list by applying an action to all or, by specifying a `<SELECTOR>`, to the selected elements of the list, storing result in-place or in the specified output variable.

Note

`TRANSFORM` sub-command does not change the number of elements of the list. If a `<SELECTOR>` is specified, only some elements will be changed, the other ones will remain same as before the transformation.

`<ACTION>` specify the action to apply to the elements of list. The actions have exactly the same semantics as sub-commands of `string()` command.

The `<ACTION>` may be one of:

`APPEND`, `PREPEND`: Append, prepend specified value to each element of the list.

```
  list(TRANSFORM <list> <APPEND|PREPEND> <value> ...)
```

`TOUPPER`, `TOLOWER`: Convert each element of the list to upper, lower characters.

```
  list(TRANSFORM <list> <TOLOWER|TOUPPER> ...)
```

`STRIP`: Remove leading and trailing spaces from each element of the list.

```
  list(TRANSFORM <list> STRIP ...)
```

`GENEX_STRIP`: Strip any `generator expressions` from each element of the list.

```
  list(TRANSFORM <list> GENEX_STRIP ...)
```

`REPLACE`: Match the regular expression as many times as possible and substitute the replacement expression for the match for each element of the list (Same semantic as `REGEX REPLACE` from `string()` command).

```
  list(TRANSFORM <list> REPLACE <regular_expression>
                                <replace_expression> ...)
```

`<SELECTOR>` select which elements of the list will be transformed. Only one type of selector can be specified at a time.

The `<SELECTOR>` may be one of:

`AT`: Specify a list of indexes.

```
  list(TRANSFORM <list> <ACTION> AT <index> [<index> ...] ...)
```

`FOR`: Specify a range with, optionally, an increment used to iterate over the range.

```
  list(TRANSFORM <list> <ACTION> FOR <start> <stop> [<step>] ...)
```

`REGEX`: Specify a regular expression. Only elements matching the regular expression will be transformed.

```
  list(TRANSFORM <list> <ACTION> REGEX <regular_expression> ...)
```

Ordering

```
list(REVERSE <list>)
```

Reverses the contents of the list in-place.

```
list(SORT <list> [COMPARE <compare>] [CASE <case>] [ORDER <order>])
```

Sorts the list in-place alphabetically. Use the `COMPARE` keyword to select the comparison method for sorting. The `<compare>` option should be one of:

- `STRING`: Sorts a list of strings alphabetically. This is the default behavior if the `COMPARE` option is not given.
- `FILE_BASENAME`: Sorts a list of pathnames of files by their basenames.

Use the `CASE` keyword to select a case sensitive or case insensitive sort mode. The `<case>` option should be one of:

- `SENSITIVE`: List items are sorted in a case-sensitive manner. This is the default behavior if the `CASE` option is not given.
- `INSENSITIVE`: List items are sorted case insensitively. The order of items which differ only by upper/lowercase is not specified.

To control the sort order, the `ORDER` keyword can be given. The `<order>` option should be one of:

- `ASCENDING`: Sorts the list in ascending order. This is the default behavior when the `ORDER` option is not given.
- `DESCENDING`: Sorts the list in descending order.

macro

Start recording a macro for later invocation as a command:

```
macro(<name> [arg1 [arg2 [arg3 ...]]])
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endmacro(<name>)
```

Define a macro named `<name>` that takes arguments named `arg1`, `arg2`, `arg3`, (...). Commands listed after macro, but before the matching `endmacro()`, are not invoked until the macro is invoked. When it is invoked, the commands recorded in the macro are first modified by replacing formal parameters (`${arg1}`) with the arguments passed, and then invoked as normal commands. In addition to referencing the formal parameters you can reference the values `${ARGC}` which will be set to the number of arguments passed into the function as well as `${ARGV0}`, `${ARGV1}`, `${ARGV2}`, ... which will have the actual values of the arguments passed in. This facilitates creating macros with optional arguments. Additionally `${ARGV}` holds the list of all arguments given to the macro and `${ARGN}` holds the list of arguments past the last expected argument. Referencing to

`${ARGV#}` arguments beyond `${ARGC}` have undefined behavior. Checking that `${ARGC}` is greater than `#` is the only way to ensure that `${ARGV#}` was passed to the function as an extra argument.

See the `cmake_policy()` command documentation for the behavior of policies inside macros.

Macro Argument Caveats

Note that the parameters to a macro and values such as `ARGN` are not variables in the usual CMake sense. They are string replacements much like the C preprocessor would do with a macro. Therefore you will NOT be able to use commands like:

```
if(ARGV1) # ARGV1 is not a variable
if(DEFINED ARGV2) # ARGV2 is not a variable
if(ARGC GREATER 2) # ARGC is not a variable
foreach(loop_var IN LISTS ARGN) # ARGN is not a variable
```

In the first case, you can use `if(${ARGV1})`. In the second and third case, the proper way to check if an optional variable was passed to the macro is to use `if(${ARGC} GREATER 2)`. In the last case, you can use `foreach(loop_var${ARGN})` but this will skip empty arguments. If you need to include them, you can use:

```
set(list_var "${ARGN}")
foreach(loop_var IN LISTS list_var)
```

Note that if you have a variable with the same name in the scope from which the macro is called, using unreferenced names will use the existing variable instead of the arguments. For example:

```
macro(_BAR)
  foreach(arg IN LISTS ARGN)
    [...]
  endforeach()
endmacro()

function(_FOO)
  _bar(x y z)
endfunction()

_foo(a b c)
```

Will loop over `a;b;c` and not over `x;y;z` as one might be expecting. If you want true CMake variables and/or better CMake scope control you should look at the function command.

mark_as_advanced

Mark cmake cached variables as advanced.

```
mark_as_advanced([CLEAR|FORCE] VAR [VAR2 ...])
```

Mark the named cached variables as advanced. An advanced variable will not be displayed in any of the cmake GUIs unless the show advanced option is on. If `CLEAR` is the first argument advanced variables are changed back to unadvanced. If `FORCE` is the first argument, then the variable is made advanced. If neither `FORCE` nor `CLEAR` is specified, new values will be marked as advanced, but if the variable already has an advanced/non-advanced state, it will not be changed.

It does nothing in script mode.

math

Mathematical expressions.

```
math(EXPR <output-variable> <math-expression> [OUTPUT_FORMAT <format>])
```

`EXPR` evaluates mathematical expression and returns result in the output variable. Example mathematical expression is `5 * (10 + 13)`. Supported operators are `+`, `-`, `*`, `/`, `%`, `|`, `&`, `^`, `~`, `<<`, `>>`, and `(...)`. They have the same meaning as they do in C code.

Numeric constants are evaluated in decimal or hexadecimal representation.

The result is formatted according to the option “OUTPUT_FORMAT” , where `<format>` is one of:

```
HEXADECIMAL = Result in output variable will be formatted in C code
Hexadecimal notation.
DECIMAL = Result in output variable will be formatted in decimal notation.
```

For example:

```
math(EXPR value "100 * 0xA" DECIMAL)  results in value is set to "1000"
math(EXPR value "100 * 0xA" HEXADECIMAL)  results in value is set to "0x3e8"
```

message

Display a message to the user.

```
message([<mode>] "message to display" ...)
```

The optional `<mode>` keyword determines the type of message:

```
(none)         = Important information
STATUS         = Incidental information
WARNING        = CMake Warning, continue processing
AUTHOR_WARNING = CMake Warning (dev), continue processing
SEND_ERROR     = CMake Error, continue processing,
                           but skip generation
FATAL_ERROR    = CMake Error, stop processing and generation
DEPRECATION    = CMake Deprecation Error or Warning if variable
                 CMAKE_ERROR_DEPRECATED or CMAKE_WARN_DEPRECATED
                 is enabled, respectively, else no message.
```

The CMake command-line tool displays STATUS messages on stdout and all other message types on stderr. The CMake GUI displays all messages in its log area. The interactive dialogs (ccmake and CMakeSetup) show STATUS messages one at a time on a status line and other messages in interactive pop-up boxes.

CMake Warning and Error message text displays using a simple markup language. Non-indented text is formatted in line-wrapped paragraphs delimited by newlines. Indented text is considered pre-formatted.

option

Provides an option that the user can optionally select.

```
option(<option_variable> "help string describing option"
       [initial value])
```

Provide an option for the user to select as `ON` or `OFF`. If no initial value is provided, `OFF` is used. If the option is already set as a normal variable then the command does nothing (see policy `CMP0077`).

If you have options that depend on the values of other options, see the module help for `CMakeDependentOption`.

return

Return from a file, directory or function.

```
return()
```

Returns from a file, directory or function. When this command is encountered in an included file (via `include()` or `find_package()`), it causes processing of the current file to stop and control is returned to the including file. If it is encountered in a file which is not included by another file, e.g. a `CMakeLists.txt`, control is returned to the parent directory if there is one. If return is called in a function, control is returned to the caller of the function. Note that a macro is not a function and does not handle return like a function does.

separate_arguments

Parse space-separated arguments into a semicolon-separated list.

```
separate_arguments(<var> <NATIVE|UNIX|WINDOWS>_COMMAND "<args>")
```

Parses a UNIX- or Windows-style command-line string “” and stores a semicolon-separated list of the arguments in `<var>`. The entire command line must be given in one “” argument.

The `UNIX_COMMAND` mode separates arguments by unquoted whitespace. It recognizes both single-quote and double-quote pairs. A backslash escapes the next literal character (`\"` is `"`); there are no special escapes (`\n` is just `n`).

The `WINDOWS_COMMAND` mode parses a Windows command-line using the same syntax the runtime library uses to construct argv at startup. It separates arguments by whitespace that is not double-quoted. Backslashes are literal unless they precede double-quotes. See the MSDN article Parsing C Command-Line Arguments for details.

The `NATIVE_COMMAND` mode parses a Windows command-line if the host system is Windows, and a UNIX command-line otherwise.

```
separate_arguments(<var>)
```

Convert the value of `<var>` to a semi-colon separated list. All spaces are replaced with ‘;’. This helps with generating command lines.

set_directory_properties

Set properties of the current directory and subdirectories in key-value pairs.

```
set_directory_properties(PROPERTIES prop1 value1 prop2 value2)
```

See Properties on Directories for the list of properties known to CMake and their individual documentation for the behavior of each property.

set_property

Set a named property in a given scope.

```
set_property(<GLOBAL                            |
              DIRECTORY [dir]                   |
              TARGET    [target1 [target2 ...]] |
              SOURCE    [src1 [src2 ...]]       |
              INSTALL   [file1 [file2 ...]]     |
              TEST      [test1 [test2 ...]]     |
              CACHE     [entry1 [entry2 ...]]>
             [APPEND] [APPEND_STRING]
             PROPERTY <name> [value1 [value2 ...]])
```

Set one property on zero or more objects of a scope. The first argument determines the scope in which the property is set. It must be one of the following:

- `GLOBAL`

  Scope is unique and does not accept a name.

- `DIRECTORY`

  Scope defaults to the current directory but another directory (already processed by CMake) may be named by full or relative path.

- `TARGET`

  Scope may name zero or more existing targets.

- `SOURCE`

  Scope may name zero or more source files. Note that source file properties are visible only to targets added in the same directory (CMakeLists.txt).

- `INSTALL`

  Scope may name zero or more installed file paths. These are made available to CPack to influence deployment.Both the property key and value may use generator expressions. Specific properties may apply to installed files and/or directories.Path components have to be separated by forward slashes, must be normalized and are case sensitive.To reference the installation prefix itself with a relative path use ".".Currently installed file properties are only defined for the WIX generator where the given paths are relative to the installation prefix.

- `TEST`

  Scope may name zero or more existing tests.

- `CACHE`

  Scope must name zero or more cache existing entries.

The required `PROPERTY` option is immediately followed by the name of the property to set. Remaining arguments are used to compose the property value in the form of a semicolon-separated list.

If the `APPEND` option is given the list is appended to any existing property value. If the `APPEND_STRING` option is given the string is appended to any existing property value as string, i.e. it results in a longer string and not a list of strings. When using `APPEND` or `APPEND_STRING` with a property defined to support `INHERITED` behavior (see `define_property()`), no inheriting occurs when finding the initial value to append to. If the property is not already directly set in the nominated scope, the command will behave as though `APPEND` or `APPEND_STRING` had not been given.

See the `cmake-properties(7)` manual for a list of properties in each scope.

set

Set a normal, cache, or environment variable to a given value. See the cmake-language(7) variablesdocumentation for the scopes and interaction of normal variables and cache entries.

Signatures of this command that specify a `<value>...` placeholder expect zero or more arguments. Multiple arguments will be joined as a ;-list to form the actual variable value to be set. Zero arguments will cause normal variables to be unset. See the `unset()` command to unset variables explicitly.

Set Normal Variable

```
set(<variable> <value>... [PARENT_SCOPE])
```

Set the given `<variable>` in the current function or directory scope.

If the `PARENT_SCOPE` option is given the variable will be set in the scope above the current scope. Each new directory or function creates a new scope. This command will set the value of a variable into the parent directory or calling function (whichever is applicable to the case at hand). The previous state of the variable's value stays the same in the current scope (e.g., if it was undefined before, it is still undefined and if it had a value, it is still that value).

Set Cache Entry

```
set(<variable> <value>... CACHE <type> <docstring> [FORCE])
```

Set the given cache `<variable>` (cache entry). Since cache entries are meant to provide user-settable values this does not overwrite existing cache entries by default. Use the `FORCE` option to overwrite existing entries.

The `<type>` must be specified as one of:

- `BOOL`

  Boolean `ON/OFF` value. `cmake-gui(1)` offers a checkbox.

- `FILEPATH`

  Path to a file on disk. `cmake-gui(1)` offers a file dialog.

- `PATH`

  Path to a directory on disk. `cmake-gui(1)` offers a file dialog.

- `STRING`

  A line of text. `cmake-gui(1)` offers a text field or a drop-down selection if the `STRINGS` cache entry property is set.

- `INTERNAL`

  A line of text. `cmake-gui(1)` does not show internal entries. They may be used to store variables persistently across runs. Use of this type implies `FORCE`.

The `<docstring>` must be specified as a line of text providing a quick summary of the option for presentation to `cmake-gui(1)` users.

If the cache entry does not exist prior to the call or the `FORCE` option is given then the cache entry will be set to the given value. Furthermore, any normal variable binding in the current scope will be removed to expose the newly cached value to any immediately following evaluation.

It is possible for the cache entry to exist prior to the call but have no type set if it was created on the `cmake(1)` command line by a user through the `-D<var>=<value>` option without specifying a type. In this case the `set` command will add the type. Furthermore, if the `<type>` is `PATH` or `FILEPATH` and the `<value>` provided on the command line is a relative path, then the `set` command will treat the path as relative to the current working directory and convert it to an absolute path.

Set Environment Variable

```
set(ENV{<variable>} <value>...)
```

Set the current process environment `<variable>` to the given value.

site_name

Set the given variable to the name of the computer.

```
site_name(variable)
```

string

String operations.

Synopsis

```
Search and Replace
  string(FIND <string> <substring> <out-var> [...])
  string(REPLACE <match-string> <replace-string> <out-var> <input>...)

Regular Expressions
  string(REGEX MATCH <match-regex> <out-var> <input>...)
  string(REGEX MATCHALL <match-regex> <out-var> <input>...)
  string(REGEX REPLACE <match-regex> <replace-expr> <out-var> <input>...)

Manipulation
  string(APPEND <string-var> [<input>...])
  string(PREPEND <string-var> [<input>...])
  string(CONCAT <out-var> [<input>...])
  string(JOIN <glue> <out-var> [<input>...])
  string(TOLOWER <string1> <out-var>)
  string(TOUPPER <string1> <out-var>)
  string(LENGTH <string> <out-var>)
  string(SUBSTRING <string> <begin> <length> <out-var>)
  string(STRIP <string> <out-var>)
  string(GENEX_STRIP <string> <out-var>)

Comparison
  string(COMPARE <op> <string1> <string2> <out-var>)

Hashing
  string(<HASH> <out-var> <input>)

Generation
  string(ASCII <number>... <out-var>)
  string(CONFIGURE <string1> <out-var> [...])
```

```
  string(MAKE_C_IDENTIFIER <string> <out-var>)
  string(RANDOM [<option>...] <out-var>)
  string(TIMESTAMP <out-var> [<format string>] [UTC])
  string(UUID <out-var> ...)
```

Search and Replace

```
string(FIND <string> <substring> <output variable> [REVERSE])
```

Return the position where the given substring was found in the supplied string. If the `REVERSE` flag was used, the command will search for the position of the last occurrence of the specified substring. If the substring is not found, a position of -1 is returned.

```
string(REPLACE <match_string>
       <replace_string> <output variable>
       <input> [<input>...])
```

Replace all occurrences of `match_string` in the input with `replace_string` and store the result in the output.

Regular Expressions

```
string(REGEX MATCH <regular_expression>
       <output variable> <input> [<input>...])
```

Match the regular expression once and store the match in the output variable. All `<input>` arguments are concatenated before matching.

```
string(REGEX MATCHALL <regular_expression>
       <output variable> <input> [<input>...])
```

Match the regular expression as many times as possible and store the matches in the output variable as a list. All `<input>` arguments are concatenated before matching.

```
string(REGEX REPLACE <regular_expression>
       <replace_expression> <output variable>
       <input> [<input>...])
```

Match the regular expression as many times as possible and substitute the replacement expression for the match in the output. All `<input>` arguments are concatenated before matching.

The replace expression may refer to paren-delimited subexpressions of the match using `\1`, `\2`, ..., `\9`. Note that two backslashes (`\\1`) are required in CMake code to get a backslash through argument parsing.

Regex Specification

The following characters have special meaning in regular expressions:

- `^`

  Matches at beginning of input

- `$`

  Matches at end of input

- `.`

  Matches any single character

- `\<char>`

  Matches the single character specified by `<char>`. Use this to match special regex characters, e.g. `\.` for a literal `.` or `\\` for a literal backslash `\`. Escaping a non-special character is unnecessary but allowed, e.g. `\a` matches `a`.

- `[ ]`

  Matches any character(s) inside the brackets

- `[^ ]`

  Matches any character(s) not inside the brackets

- `-`

  Inside brackets, specifies an inclusive range between characters on either side e.g. `[a-f]` is `[abcdef]` To match a literal `-` using brackets, make it the first or the last character e.g. `[+*/-]` matches basic mathematical operators.

- `*`

  Matches preceding pattern zero or more times

- `+`

  Matches preceding pattern one or more times

- `?`

  Matches preceding pattern zero or once only

- `|`

  Matches a pattern on either side of the `|`

- `()`

  Saves a matched subexpression, which can be referenced in the `REGEX REPLACE` operation. Additionally it is saved by all regular expression-related commands, including e.g. `if(MATCHES)`, in the variables `CMAKE_MATCH_` for `<n>` 0..9.

`*`, `+` and `?` have higher precedence than concatenation. `|` has lower precedence than concatenation. This means that the regular expression `^ab+d$` matches `abbd` but not `ababd`, and the regular expression `^(ab|cd)$` matches `ab` but not `abd`.

CMake language Escape Sequences such as `\t`, `\r`, `\n`, and `\\` may be used to construct literal tabs, carriage returns, newlines, and backslashes (respectively) to pass in a regex. For example:

- The quoted argument `"[ \t\r\n]"` specifies a regex that matches any single whitespace character.

- The quoted argument `"[/\\]"` specifies a regex that matches a single forward slash `/` or backslash `\`.
- The quoted argument `"[A-Za-z0-9_]"` specifies a regex that matches any single "word" character in the C locale.
- The quoted argument `"\\(\\a\\+b\\)"` specifies a regex that matches the exact string `(a+b)`. Each `\\` is parsed in a quoted argument as just `\`, so the regex itself is actually `\(\a\+\b\)`. This can alternatively be specified in a Bracket Argument without having to escape the backslashes, e.g. `[[\(\a\+\b\)]]`.

Manipulation

```
string(APPEND <string variable> [<input>...])
```

Append all the input arguments to the string.

```
string(PREPEND <string variable> [<input>...])
```

Prepend all the input arguments to the string.

```
string(CONCAT <output variable> [<input>...])
```

Concatenate all the input arguments together and store the result in the named output variable.

```
string(JOIN <glue> <output variable> [<input>...])
```

Join all the input arguments together using the glue string and store the result in the named output variable.

To join list's elements, use preferably the `JOIN` operator from `list()` command. This allows for the elements to have special characters like `;` in them.

```
string(TOLOWER <string1> <output variable>)
```

Convert string to lower characters.

```
string(TOUPPER <string1> <output variable>)
```

Convert string to upper characters.

```
string(LENGTH <string> <output variable>)
```

Store in an output variable a given string's length.

```
string(SUBSTRING <string> <begin> <length> <output variable>)
```

Store in an output variable a substring of a given string. If length is `-1` the remainder of the string starting at begin will be returned. If string is shorter than length then end of string is used instead.

Note

CMake 3.1 and below reported an error if length pointed past the end of string.

```
string(STRIP <string> <output variable>)
```

Store in an output variable a substring of a given string with leading and trailing spaces removed.

```
string(GENEX_STRIP <input string> <output variable>)
```

Strip any `generator expressions` from the `input string` and store the result in the `output variable`.

Comparison

```
string(COMPARE LESS <string1> <string2> <output variable>)
string(COMPARE GREATER <string1> <string2> <output variable>)
string(COMPARE EQUAL <string1> <string2> <output variable>)
string(COMPARE NOTEQUAL <string1> <string2> <output variable>)
string(COMPARE LESS_EQUAL <string1> <string2> <output variable>)
string(COMPARE GREATER_EQUAL <string1> <string2> <output variable>)
```

Compare the strings and store true or false in the output variable.

Hashing

```
string(<HASH> <output variable> <input>)
```

Compute a cryptographic hash of the input string. The supported `<HASH>` algorithm names are:

- `MD5`

  Message-Digest Algorithm 5, RFC 1321.
- `SHA1`

  US Secure Hash Algorithm 1, RFC 3174.
- `SHA224`

  US Secure Hash Algorithms, RFC 4634.
- `SHA256`

  US Secure Hash Algorithms, RFC 4634.
- `SHA384`

  US Secure Hash Algorithms, RFC 4634.
- `SHA512`

  US Secure Hash Algorithms, RFC 4634.
- `SHA3_224`

  Keccak SHA-3.

- `SHA3_256`

  Keccak SHA-3.

- `SHA3_384`

  Keccak SHA-3.

- `SHA3_512`

  Keccak SHA-3.

Generation

```
string(ASCII <number> [<number> ...] <output variable>)
```

Convert all numbers into corresponding ASCII characters.

```
string(CONFIGURE <string1> <output variable>
       [@ONLY] [ESCAPE_QUOTES])
```

Transform a string like `configure_file()` transforms a file.

```
string(MAKE_C_IDENTIFIER <input string> <output variable>)
```

Convert each non-alphanumeric character in the `<input string>` to an underscore and store the result in the `<outputvariable>`. If the first character of the string is a digit, an underscore will also be prepended to the result.

```
string(RANDOM [LENGTH <length>] [ALPHABET <alphabet>]
       [RANDOM_SEED <seed>] <output variable>)
```

Return a random string of given length consisting of characters from the given alphabet. Default length is 5 characters and default alphabet is all numbers and upper and lower case letters. If an integer `RANDOM_SEED` is given, its value will be used to seed the random number generator.

```
string(TIMESTAMP <output variable> [<format string>] [UTC])
```

Write a string representation of the current date and/or time to the output variable.

Should the command be unable to obtain a timestamp the output variable will be set to the empty string “”.

The optional `UTC` flag requests the current date/time representation to be in Coordinated Universal Time (UTC) rather than local time.

The optional `<format string>` may contain the following format specifiers:

```
%%        A literal percent sign (%).
%d        The day of the current month (01-31).
%H        The hour on a 24-hour clock (00-23).
%I        The hour on a 12-hour clock (01-12).
%j        The day of the current year (001-366).
```

```
%m        The month of the current year (01-12).
%b        Abbreviated month name (e.g. Oct).
%B        Full month name (e.g. October).
%M        The minute of the current hour (00-59).
%s        Seconds since midnight (UTC) 1-Jan-1970 (UNIX time).
%S        The second of the current minute.
          60 represents a leap second. (00-60)
%U        The week number of the current year (00-53).
%w        The day of the current week. 0 is Sunday. (0-6)
%a        Abbreviated weekday name (e.g. Fri).
%A        Full weekday name (e.g. Friday).
%y        The last two digits of the current year (00-99)
%Y        The current year.
```

Unknown format specifiers will be ignored and copied to the output as-is.

If no explicit `<format string>` is given it will default to:

```
%Y-%m-%dT%H:%M:%S    for local time.
%Y-%m-%dT%H:%M:%SZ   for UTC.
```

Note

If the `SOURCE_DATE_EPOCH` environment variable is set, its value will be used instead of the current time. See https://reproducible-builds.org/specs/source-date-epoch/ for details.

```
string(UUID <output variable> NAMESPACE <namespace> NAME <name>
       TYPE <MD5|SHA1> [UPPER])
```

Create a universally unique identifier (aka GUID) as per RFC4122 based on the hash of the combined values of `<namespace>` (which itself has to be a valid UUID) and `<name>`. The hash algorithm can be either `MD5` (Version 3 UUID) or `SHA1` (Version 5 UUID). A UUID has the format `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx` where each x represents a lower case hexadecimal character. Where required an uppercase representation can be requested with the optional `UPPER` flag.

unset

Unset a variable, cache variable, or environment variable.

```
unset(<variable> [CACHE | PARENT_SCOPE])
```

Removes a normal variable from the current scope, causing it to become undefined. If `CACHE` is present, then a cache variable is removed instead of a normal variable. Note that when evaluating Variable References of the form `${VAR}`, CMake first searches for a normal variable with that name. If no such normal variable exists, CMake will then search for a cache entry with that name. Because of this unsetting a normal variable can expose a cache variable that was previously hidden. To force a variable reference of the form `${VAR}` to return an empty string, use `set(<variable> "")`, which clears the normal variable but leaves it defined.

If `PARENT_SCOPE` is present then the variable is removed from the scope above the current scope. See the same option in the `set()` command for further details.

`<variable>` can be an environment variable such as:

```
unset(ENV{LD_LIBRARY_PATH})
```

in which case the variable will be removed from the current environment.

variable_watch

Watch the CMake variable for change.

```
variable_watch(<variable name> [<command to execute>])
```

If the specified variable changes, the message will be printed about the variable being changed. If the command is specified, the command will be executed. The command will receive the following arguments: COMMAND( )

while

Evaluate a group of commands while a condition is true

```
while(condition)
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endwhile(condition)
```

All commands between while and the matching `endwhile()` are recorded without being invoked. Once the `endwhile()` is evaluated, the recorded list of commands is invoked as long as the condition is true. The condition is evaluated using the same logic as the `if()` command.

==========

Project Commands

add_compile_definitions

Adds preprocessor definitions to the compilation of source files.

```
add_compile_definitions(<definition> ...)
```

Adds preprocessor definitions to the compiler command line for targets in the current directory and below (whether added before or after this command is invoked). See documentation of the `directory` and `target` `COMPILE_DEFINITIONS` properties.

Definitions are specified using the syntax `VAR` or `VAR=value`. Function-style definitions are not supported. CMake will automatically escape the value correctly for the native build system (note that CMake language syntax may require escapes to specify some values).

Arguments to `add_compile_definitions` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

add_compile_options

Adds options to the compilation of source files.

```
add_compile_options(<option> ...)
```

Adds options to the compiler command line for targets in the current directory and below that are added after this command is invoked. See documentation of the `directory` and `target` `COMPILE_OPTIONS` properties.

This command can be used to add any options, but alternative commands exist to add preprocessor definitions (`target_compile_definitions()` and `add_compile_definitions()`) or include directories (`target_include_directories()` and `include_directories()`).

Arguments to `add_compile_options` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

The final set of compile or link options used for a target is constructed by accumulating options from the current target and the usage requirements of it dependencies. The set of options is de-duplicated to avoid repetition. While beneficial for individual options, the de-duplication step can break up option groups. For example, `-D A -D B` becomes `-D A B`. One may specify a group of options using shell-like quoting along with a `SHELL:` prefix. The `SHELL:` prefix is dropped and the rest of the option string is parsed using the `separate_arguments()` `UNIX_COMMAND` mode. For example, `"SHELL:-D A" "SHELL:-D B"` becomes `-D A -D B`.

add_custom_command

Add a custom build rule to the generated build system.

There are two main signatures for `add_custom_command`.

Generating Files

The first signature is for adding a custom command to produce an output:

```
add_custom_command(OUTPUT output1 [output2 ...]
                   COMMAND command1 [ARGS] [args1...]
                   [COMMAND command2 [ARGS] [args2...] ...]
                   [MAIN_DEPENDENCY depend]
                   [DEPENDS [depends...]]
                   [BYPRODUCTS [files...]]
                   [IMPLICIT_DEPENDS <lang1> depend1
                                    [<lang2> depend2] ...]
                   [WORKING_DIRECTORY dir]
                   [COMMENT comment]
                   [DEPFILE depfile]
                   [VERBATIM] [APPEND] [USES_TERMINAL]
                   [COMMAND_EXPAND_LISTS])
```

This defines a command to generate specified `OUTPUT` file(s). A target created in the same directory (`CMakeLists.txt` file) that specifies any output of the custom command as a source file is given a rule to generate the file using the command at build time. Do not list the output in more than one independent target that may build in parallel or the two instances of the rule may conflict (instead use the `add_custom_target()` command to drive the command and make the other targets depend on that one). In makefile terms this creates a new target in the following form:

```
OUTPUT: MAIN_DEPENDENCY DEPENDS
        COMMAND
```

The options are:

- `APPEND`

  Append the `COMMAND` and `DEPENDS` option values to the custom command for the first output specified. There must have already been a previous call to this command with the same output. The `COMMENT`, `MAIN_DEPENDENCY`, and `WORKING_DIRECTORY` options are currently ignored when APPEND is given, but may be used in the future.

- `BYPRODUCTS`

  Specify the files the command is expected to produce but whose modification time may or may not be newer than the dependencies. If a byproduct name is a relative path it will be interpreted relative to the build tree directory corresponding to the current source directory. Each byproduct file will be marked with the `GENERATED` source file property automatically.Explicit specification of byproducts is supported by the `Ninja` generator to tell the `ninja` build tool how to regenerate byproducts when they are missing. It is also useful when other build rules (e.g. custom commands) depend on the byproducts. Ninja requires a build rule for any generated file on which another rule depends even if there are order-only dependencies to ensure the byproducts will be available before their dependents build.The `BYPRODUCTS` option is ignored on non-Ninja generators except to mark byproducts `GENERATED`.

- `COMMAND`

Specify the command-line(s) to execute at build time. If more than one `COMMAND` is specified they will be executed in order, but *not* necessarily composed into a stateful shell or batch script. (To run a full script, use the `configure_file()` command or the `file(GENERATE)` command to create it, and then specify a `COMMAND` to launch it.) The optional `ARGS` argument is for backward compatibility and will be ignored.If `COMMAND` specifies an executable target name (created by the `add_executable()` command) it will automatically be replaced by the location of the executable created at build time. If set, the`CROSSCOMPILING_EMULATOR` executable target property will also be prepended to the command to allow the executable to run on the host. (Use the `TARGET_FILE` `generator expression` to reference an executable later in the command line.) Additionally a target-level dependency will be added so that the executable target will be built before any target using this custom command. However this does NOT add a file-level dependency that would cause the custom command to re-run whenever the executable is recompiled.Arguments to `COMMAND` may use `generator expressions`. References to target names in generator expressions imply target-level dependencies, but NOT file-level dependencies. List target names with the `DEPENDS` option to add file-level dependencies.

- `COMMENT`

  Display the given message before the commands are executed at build time.

- `DEPENDS`

  Specify files on which the command depends. If any dependency is an `OUTPUT` of another custom command in the same directory (`CMakeLists.txt` file) CMake automatically brings the other custom command into the target in which this command is built. If `DEPENDS` is not specified the command will run whenever the `OUTPUT` is missing; if the command does not actually create the `OUTPUT` then the rule will always run. If `DEPENDS` specifies any target (created by the `add_custom_target()`, `add_executable()`, or `add_library()` command) a target-level dependency is created to make sure the target is built before any target using this custom command. Additionally, if the target is an executable or library a file-level dependency is created to cause the custom command to re-run whenever the target is recompiled.Arguments to `DEPENDS` may use `generator expressions`.

- `COMMAND_EXPAND_LISTS`

  Lists in `COMMAND` arguments will be expanded, including those created with `generator expressions`, allowing `COMMAND` arguments such as `${CC} "-I$<JOIN:$<TARGET_PROPERTY:foo,INCLUDE_DIRECTORIES>,;-I>" foo.cc` to be properly expanded.

- `IMPLICIT_DEPENDS`

  Request scanning of implicit dependencies of an input file. The language given specifies the programming language whose corresponding dependency scanner should be used. Currently only `C` and `CXX` language scanners are supported. The language has to be specified for every file in the `IMPLICIT_DEPENDS` list. Dependencies discovered from the scanning are added to those of the custom command at build time. Note that the `IMPLICIT_DEPENDS` option is currently supported only for Makefile generators and will be ignored by other generators.

- `MAIN_DEPENDENCY`

  Specify the primary input source file to the command. This is treated just like any value given to the `DEPENDS` option but also suggests to Visual Studio generators where to hang the custom command. Each source file may have at most one command specifying it as its main dependency. A compile command (i.e. for a library or an executable) counts as an implicit main dependency which gets silently overwritten by a custom command specification.

- `OUTPUT`

  Specify the output files the command is expected to produce. If an output name is a relative path it will be interpreted relative to the build tree directory corresponding to the current source directory. Each output file will be marked with the `GENERATED` source file property automatically. If the output of the custom command is not actually created as a file on disk it should be marked with the `SYMBOLIC` source file property.

- `USES_TERMINAL`

  The command will be given direct access to the terminal if possible. With the `Ninja` generator, this places the command in the `console` `pool`.

- `VERBATIM`

  All arguments to the commands will be escaped properly for the build tool so that the invoked command receives each argument unchanged. Note that one level of escapes is still used by the CMake language processor before add_custom_command even sees the arguments. Use of `VERBATIM` is recommended as it enables correct behavior. When `VERBATIM` is not given the behavior is platform specific because there is no protection of tool-specific special characters.

- `WORKING_DIRECTORY`

  Execute the command with the given current working directory. If it is a relative path it will be interpreted relative to the build tree directory corresponding to the current source directory.Arguments to `WORKING_DIRECTORY` may use `generator expressions`.

- `DEPFILE`

  Specify a `.d` depfile for the `Ninja` generator. A `.d` file holds dependencies usually emitted by the custom command itself. Using `DEPFILE` with other generators than Ninja is an error.

Build Events

The second signature adds a custom command to a target such as a library or executable. This is useful for performing an operation before or after building the target. The command becomes part of the target and will only execute when the target itself is built. If the target is already built, the command will not execute.

```
add_custom_command(TARGET <target>
                   PRE_BUILD | PRE_LINK | POST_BUILD
                   COMMAND command1 [ARGS] [args1...]
                   [COMMAND command2 [ARGS] [args2...] ...]
                   [BYPRODUCTS [files...]]
                   [WORKING_DIRECTORY dir]
                   [COMMENT comment]
                   [VERBATIM] [USES_TERMINAL])
```

This defines a new command that will be associated with building the specified `<target>`. The `<target>` must be defined in the current directory; targets defined in other directories may not be specified.

When the command will happen is determined by which of the following is specified:

- `PRE_BUILD`

On Visual Studio Generators, run before any other rules are executed within the target. On other generators, run just before `PRE_LINK` commands.

- `PRE_LINK`

  Run after sources have been compiled but before linking the binary or running the librarian or archiver tool of a static library. This is not defined for targets created by the `add_custom_target()` command.

- `POST_BUILD`

  Run after all other rules within the target have been executed.

Note

Because generator expressions can be used in custom commands, it is possible to define `COMMAND` lines or whole custom commands which evaluate to empty strings for certain configurations. For **Visual Studio 2010 (and newer)** generators these command lines or custom commands will be omitted for the specific configuration and no "empty-string-command" will be added.

This allows to add individual build events for every configuration.

add_custom_target

Add a target with no output so it will always be built.

```
add_custom_target(Name [ALL] [command1 [args1...]]
                  [COMMAND command2 [args2...] ...]
                  [DEPENDS depend depend depend ... ]
                  [BYPRODUCTS [files...]]
                  [WORKING_DIRECTORY dir]
                  [COMMENT comment]
                  [VERBATIM] [USES_TERMINAL]
                  [COMMAND_EXPAND_LISTS]
                  [SOURCES src1 [src2...]])
```

Adds a target with the given name that executes the given commands. The target has no output file and is *always considered out of date* even if the commands try to create a file with the name of the target. Use the `add_custom_command()` command to generate a file with dependencies. By default nothing depends on the custom target. Use the `add_dependencies()` command to add dependencies to or from other targets.

The options are:

- `ALL`

  Indicate that this target should be added to the default build target so that it will be run every time (the command cannot be called `ALL`).

- `BYPRODUCTS`

  Specify the files the command is expected to produce but whose modification time may or may not be updated on subsequent builds. If a byproduct name is a relative path it will be interpreted relative to the build tree directory corresponding to the current source directory. Each byproduct file will be marked with the `GENERATED` source file property automatically.Explicit specification of byproducts is supported

by the `Ninja` generator to tell the `ninja` build tool how to regenerate byproducts when they are missing. It is also useful when other build rules (e.g. custom commands) depend on the byproducts. Ninja requires a build rule for any generated file on which another rule depends even if there are order-only dependencies to ensure the byproducts will be available before their dependents build.The `BYPRODUCTS` option is ignored on non-Ninja generators except to mark byproducts `GENERATED`.

- `COMMAND`

  Specify the command-line(s) to execute at build time. If more than one `COMMAND` is specified they will be executed in order, but *not* necessarily composed into a stateful shell or batch script. (To run a full script, use the `configure_file()` command or the `file(GENERATE)` command to create it, and then specify a `COMMAND` to launch it.)If `COMMAND` specifies an executable target name (created by the `add_executable()` command) it will automatically be replaced by the location of the executable created at build time. If set, the `CROSSCOMPILING_EMULATOR` executable target property will also be prepended to the command to allow the executable to run on the host. Additionally a target-level dependency will be added so that the executable target will be built before this custom target.Arguments to `COMMAND` may use `generator expressions`. References to target names in generator expressions imply target-level dependencies.The command and arguments are optional and if not specified an empty target will be created.

- `COMMENT`

  Display the given message before the commands are executed at build time.

- `DEPENDS`

  Reference files and outputs of custom commands created with `add_custom_command()` command calls in the same directory (`CMakeLists.txt` file). They will be brought up to date when the target is built.Use the `add_dependencies()` command to add dependencies on other targets.

- `COMMAND_EXPAND_LISTS`

  Lists in `COMMAND` arguments will be expanded, including those created with `generator expressions`, allowing `COMMAND` arguments such as `${CC} "-I$<JOIN:$<TARGET_PROPERTY:foo,INCLUDE_DIRECTORIES>,;-I>" foo.cc` to be properly expanded.

- `SOURCES`

  Specify additional source files to be included in the custom target. Specified source files will be added to IDE project files for convenience in editing even if they have no build rules.

- `VERBATIM`

  All arguments to the commands will be escaped properly for the build tool so that the invoked command receives each argument unchanged. Note that one level of escapes is still used by the CMake language processor before `add_custom_target` even sees the arguments. Use of `VERBATIM` is recommended as it enables correct behavior. When `VERBATIM` is not given the behavior is platform specific because there is no protection of tool-specific special characters.

- `USES_TERMINAL`

  The command will be given direct access to the terminal if possible. With the `Ninja` generator, this places the command in the `console` `pool`.

- `WORKING_DIRECTORY`

Execute the command with the given current working directory. If it is a relative path it will be interpreted relative to the build tree directory corresponding to the current source directory.Arguments to `WORKING_DIRECTORY` may use `generator expressions`.

add_definitions

Adds -D define flags to the compilation of source files.

```
add_definitions(-DFOO -DBAR ...)
```

Adds definitions to the compiler command line for targets in the current directory and below (whether added before or after this command is invoked). This command can be used to add any flags, but it is intended to add preprocessor definitions.

Note

This command has been superseded by alternatives:

- Use `add_compile_definitions()` to add preprocessor definitions.
- Use `include_directories()` to add include directories.
- Use `add_compile_options()` to add other options.

Flags beginning in -D or /D that look like preprocessor definitions are automatically added to the `COMPILE_DEFINITIONS` directory property for the current directory. Definitions with non-trivial values may be left in the set of flags instead of being converted for reasons of backwards compatibility. See documentation of the `directory`, `target`, `source file` `COMPILE_DEFINITIONS` properties for details on adding preprocessor definitions to specific scopes and configurations.

See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

add_dependencies

Add a dependency between top-level targets.

```
add_dependencies(<target> [<target-dependency>]...)
```

Make a top-level `<target>` depend on other top-level targets to ensure that they build before `<target>` does. A top-level target is one created by one of the `add_executable()`, `add_library()`, or `add_custom_target()` commands (but not targets generated by CMake like `install`).

Dependencies added to an imported target or an interface library are followed transitively in its place since the target itself does not build.

See the `DEPENDS` option of `add_custom_target()` and `add_custom_command()` commands for adding file-level dependencies in custom rules. See the `OBJECT_DEPENDS` source file property to add file-level dependencies to object files.

add_executable

Add an executable to the project using the specified source files.

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
               [EXCLUDE_FROM_ALL]
               [source1] [source2 ...])
```

Adds an executable target called `<name>` to be built from the source files listed in the command invocation. (The source files can be omitted here if they are added later using `target_sources()`.) The `<name>` corresponds to the logical target name and must be globally unique within a project. The actual file name of the executable built is constructed based on conventions of the native platform (such as `<name>.exe` or just `<name>`).

By default the executable file will be created in the build tree directory corresponding to the source tree directory in which the command was invoked. See documentation of the `RUNTIME_OUTPUT_DIRECTORY` target property to change this location. See documentation of the `OUTPUT_NAME` target property to change the `<name>` part of the final file name.

If `WIN32` is given the property `WIN32_EXECUTABLE` will be set on the target created. See documentation of that target property for details.

If `MACOSX_BUNDLE` is given the corresponding property will be set on the created target. See documentation of the `MACOSX_BUNDLE` target property for details.

If `EXCLUDE_FROM_ALL` is given the corresponding property will be set on the created target. See documentation of the `EXCLUDE_FROM_ALL` target property for details.

Source arguments to `add_executable` may use “generator expressions” with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

See also `HEADER_FILE_ONLY` on what to do if some sources are pre-processed, and you want to have the original sources reachable from within IDE.

```
add_executable(<name> IMPORTED [GLOBAL])
```

An IMPORTED executable target references an executable file located outside the project. No rules are generated to build it, and the `IMPORTED` target property is `True`. The target name has scope in the directory in which it is created and below, but the `GLOBAL` option extends visibility. It may be referenced like any target built within the project. `IMPORTED` executables are useful for convenient reference from commands like `add_custom_command()`. Details about the imported executable are specified by setting properties whose names begin in `IMPORTED_`. The most important such property is `IMPORTED_LOCATION` (and its per-configuration version `IMPORTED_LOCATION_`) which specifies the location of the main executable file on disk. See documentation of the `IMPORTED_*` properties for more information.

```
add_executable(<name> ALIAS <target>)
```

Creates an Alias Target, such that `<name>` can be used to refer to `<target>` in subsequent commands. The `<name>` does not appear in the generated buildsystem as a make target. The `<target>` may not be a non-`GLOBAL` Imported Target or an `ALIAS`. `ALIAS` targets can be used as targets to read properties from, executables for custom commands and custom targets. They can also be tested for existence with the regular `if(TARGET)` subcommand. The `<name>` may not be used to modify properties of `<target>`, that is, it may not be used as the operand of `set_property()`, `set_target_properties()`, `target_link_libraries()` etc. An `ALIAS` target may not be installed or exported.

add_library

Contents

Add a library to the project using the specified source files.

Normal Libraries

```
add_library(<name> [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            [source1] [source2 ...])
```

Adds a library target called `<name>` to be built from the source files listed in the command invocation. (The source files can be omitted here if they are added later using `target_sources()`.) The `<name>` corresponds to the logical target name and must be globally unique within a project. The actual file name of the library built is constructed based on conventions of the native platform (such as `lib<name>.a` or `<name>.lib`).

`STATIC`, `SHARED`, or `MODULE` may be given to specify the type of library to be created. `STATIC` libraries are archives of object files for use when linking other targets. `SHARED` libraries are linked dynamically and loaded at runtime. `MODULE` libraries are plugins that are not linked into other targets but may be loaded dynamically at runtime using dlopen-like functionality. If no type is given explicitly the type is `STATIC` or `SHARED` based on whether the current value of the variable `BUILD_SHARED_LIBS` is `ON`. For `SHARED` and `MODULE` libraries the `POSITION_INDEPENDENT_CODE` target property is set to `ON` automatically. A `SHARED` or `STATIC` library may be marked with the `FRAMEWORK` target property to create an macOS Framework.

If a library does not export any symbols, it must not be declared as a `SHARED` library. For example, a Windows resource DLL or a managed C++/CLI DLL that exports no unmanaged symbols would need to be a `MODULE` library. This is because CMake expects a `SHARED` library to always have an associated import library on Windows.

By default the library file will be created in the build tree directory corresponding to the source tree directory in which the command was invoked. See documentation of the `ARCHIVE_OUTPUT_DIRECTORY`, `LIBRARY_OUTPUT_DIRECTORY`, and `RUNTIME_OUTPUT_DIRECTORY` target properties to change this location. See documentation of the `OUTPUT_NAME` target property to change the `<name>` part of the final file name.

If `EXCLUDE_FROM_ALL` is given the corresponding property will be set on the created target. See documentation of the `EXCLUDE_FROM_ALL` target property for details.

Source arguments to `add_library` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

See also `HEADER_FILE_ONLY` on what to do if some sources are pre-processed, and you want to have the original sources reachable from within IDE.

Imported Libraries

```
add_library(<name> <SHARED|STATIC|MODULE|OBJECT|UNKNOWN> IMPORTED
            [GLOBAL])
```

An IMPORTED library target references a library file located outside the project. No rules are generated to build it, and the `IMPORTED` target property is `True`. The target name has scope in the directory in which it is created and below, but the `GLOBAL` option extends visibility. It may be referenced like any target built within the project. `IMPORTED` libraries are useful for convenient reference from commands like `target_link_libraries()`. Details about the imported library are specified by setting properties whose names begin in `IMPORTED_` and `INTERFACE_`. The most important such property is `IMPORTED_LOCATION` (and its per-configuration variant `IMPORTED_LOCATION_`) which specifies the location of the main library file on disk. Or, for object libraries, `IMPORTED_OBJECTS` (and `IMPORTED_OBJECTS_`) specifies the locations of object files on disk. See documentation of the `IMPORTED_*` and `INTERFACE_*` properties for more information.

Object Libraries

```
add_library(<name> OBJECT <src>...)
```

Creates an Object Library. An object library compiles source files but does not archive or link their object files into a library. Instead other targets created by `add_library()` or `add_executable()` may reference the objects using an expression of the form `$<TARGET_OBJECTS:objlib>` as a source, where `objlib` is the object library name. For example:

```
add_library(... $<TARGET_OBJECTS:objlib> ...)
add_executable(... $<TARGET_OBJECTS:objlib> ...)
```

will include objlib's object files in a library and an executable along with those compiled from their own sources. Object libraries may contain only sources that compile, header files, and other files that would not affect linking of a normal library (e.g. `.txt`). They may contain custom commands generating such sources, but not `PRE_BUILD`, `PRE_LINK`, or `POST_BUILD` commands. Some native build systems (such as Xcode) may

not like targets that have only object files, so consider adding at least one real source file to any target that references `$<TARGET_OBJECTS:objlib>`.

Alias Libraries

```
add_library(<name> ALIAS <target>)
```

Creates an Alias Target, such that `<name>` can be used to refer to `<target>` in subsequent commands. The `<name>` does not appear in the generated buildsystem as a make target. The `<target>` may not be a non-`GLOBAL` Imported Target or an `ALIAS`. `ALIAS` targets can be used as linkable targets and as targets to read properties from. They can also be tested for existence with the regular `if(TARGET)` subcommand. The `<name>` may not be used to modify properties of `<target>`, that is, it may not be used as the operand of `set_property()`, `set_target_properties()`, `target_link_libraries()` etc. An `ALIAS` target may not be installed or exported.

Interface Libraries

```
add_library(<name> INTERFACE [IMPORTED [GLOBAL]])
```

Creates an Interface Library. An `INTERFACE` library target does not directly create build output, though it may have properties set on it and it may be installed, exported and imported. Typically the `INTERFACE_*` properties are populated on the interface target using the commands:

- `set_property()`,
- `target_link_libraries(INTERFACE)`,
- `target_link_options(INTERFACE)`,
- `target_include_directories(INTERFACE)`,
- `target_compile_options(INTERFACE)`,
- `target_compile_definitions(INTERFACE)`, and
- `target_sources(INTERFACE)`,

and then it is used as an argument to `target_link_libraries()` like any other target.

An `INTERFACE` Imported Target may also be created with this signature. An `IMPORTED` library target references a library defined outside the project. The target name has scope in the directory in which it is created and below, but the `GLOBAL` option extends visibility. It may be referenced like any target built within the project. `IMPORTED` libraries are useful for convenient reference from commands like `target_link_libraries()`.

add_link_options

Adds options to the link of shared library, module and executable targets.

```
add_link_options(<option> ...)
```

Adds options to the link step for targets in the current directory and below that are added after this command is invoked. See documentation of the `directory` and `target` `LINK_OPTIONS` properties.

This command can be used to add any options, but alternative commands exist to add libraries (`target_link_libraries()` or `link_libraries()`).

Arguments to `add_link_options` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

The final set of compile or link options used for a target is constructed by accumulating options from the current target and the usage requirements of it dependencies. The set of options is de-duplicated to avoid repetition. While beneficial for individual options, the de-duplication step can break up option groups. For example, `-D A -D B` becomes `-D A B`. One may specify a group of options using shell-like quoting along with a `SHELL:` prefix. The `SHELL:` prefix is dropped and the rest of the option string is parsed using the `separate_arguments()` `UNIX_COMMAND` mode. For example, `"SHELL:-D A" "SHELL:-D B"` becomes `-D A -D B`.

To pass options to the linker tool, each compiler driver has is own syntax. The `LINKER:` prefix can be used to specify, in a portable way, options to pass to the linker tool. The `LINKER:` prefix is replaced by the required driver option and the rest of the option string defines linker arguments using `,` as separator. These arguments will be formatted according to the `CMAKE_<LANG>_LINKER_WRAPPER_FLAG` and `CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP` variables.

For example, `"LINKER:-z,defs"` becomes `-Xlinker -z -Xlinker defs` for `Clang` and `-Wl,-z,defs` for `GNU GCC`.

The `LINKER:` prefix can be specified as part of a `SHELL:` prefix expression.

The `LINKER:` prefix supports, as alternate syntax, specification of arguments using `SHELL:` prefix and space as separator. Previous example becomes `"LINKER:SHELL:-z defs"`.

Note

Specifying `SHELL:` prefix elsewhere than at the beginning of the `LINKER:` prefix is not supported.

add_subdirectory

Add a subdirectory to the build.

```
add_subdirectory(source_dir [binary_dir]
                 [EXCLUDE_FROM_ALL])
```

Add a subdirectory to the build. The source_dir specifies the directory in which the source CMakeLists.txt and code files are located. If it is a relative path it will be evaluated with respect to the current directory (the typical usage), but it may also be an absolute path. The `binary_dir` specifies the directory in which to place the output files. If it is a relative path it will be evaluated with respect to the current output directory, but it may also be an absolute path. If `binary_dir` is not specified, the value of `source_dir`, before expanding any relative path, will be used (the typical usage). The CMakeLists.txt file in the specified source directory will be processed immediately by CMake before processing in the current input file continues beyond this command.

If the `EXCLUDE_FROM_ALL` argument is provided then targets in the subdirectory will not be included in the `ALL` target of the parent directory by default, and will be excluded from IDE project files. Users must explicitly build targets in the subdirectory. This is meant for use when the subdirectory contains a separate part of the project that is useful but not necessary, such as a set of examples. Typically the subdirectory should contain its own `project()` command invocation so that a full build system will be generated in the subdirectory (such as a VS IDE solution file). Note that inter-target dependencies supersede this exclusion. If a target built by the parent project depends on a target in the subdirectory, the dependee target will be included in the parent project build system to satisfy the dependency.

add_test

Add a test to the project to be run by `ctest(1)`.

```
add_test(NAME <name> COMMAND <command> [<arg>...]
         [CONFIGURATIONS <config>...]
         [WORKING_DIRECTORY <dir>])
```

Add a test called `<name>`. The test name may not contain spaces, quotes, or other characters special in CMake syntax. The options are:

- `COMMAND`

  Specify the test command-line. If `<command>` specifies an executable target (created by `add_executable()`) it will automatically be replaced by the location of the executable created at build time.

- `CONFIGURATIONS`

  Restrict execution of the test only to the named configurations.

- `WORKING_DIRECTORY`

  Set the `WORKING_DIRECTORY` test property to specify the working directory in which to execute the test. If not specified the test will be run with the current working directory set to the build directory corresponding to the current source directory.

The given test command is expected to exit with code `0` to pass and non-zero to fail, or vice-versa if the `WILL_FAIL` test property is set. Any output written to stdout or stderr will be captured by `ctest(1)` but does not affect the pass/fail status unless the `PASS_REGULAR_EXPRESSION` or `FAIL_REGULAR_EXPRESSION` test property is used.

The `COMMAND` and `WORKING_DIRECTORY` options may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions.

Example usage:

```
add_test(NAME mytest
         COMMAND testDriver --config $<CONFIGURATION>
                            --exe $<TARGET_FILE:myexe>)
```

This creates a test `mytest` whose command runs a `testDriver` tool passing the configuration name and the full path to the executable file produced by target `myexe`.

Note

CMake will generate tests only if the `enable_testing()` command has been invoked. The `CTest` module invokes the command automatically when the `BUILD_TESTING` option is `ON`.

```
add_test(<name> <command> [<arg>...])
```

Add a test called `<name>` with the given command-line. Unlike the above `NAME` signature no transformation is performed on the command-line to support target names or generator expressions.

aux_source_directory

Find all source files in a directory.

```
aux_source_directory(<dir> <variable>)
```

Collects the names of all the source files in the specified directory and stores the list in the `<variable>` provided. This command is intended to be used by projects that use explicit template instantiation. Template instantiation files can be stored in a "Templates" subdirectory and collected automatically using this command to avoid manually listing all instantiations.

It is tempting to use this command to avoid writing the list of source files for a library or executable target. While this seems to work, there is no way for CMake to generate a build system that knows when a new source file has been added. Normally the generated build system knows when it needs to rerun CMake because the CMakeLists.txt file is modified to add a new source. When the source is just added to the directory without modifying this file, one would have to manually rerun CMake to generate a build system incorporating the new file.

build_command

Get a command line to build the current project. This is mainly intended for internal use by the `CTest` module.

```
build_command(<variable>
              [CONFIGURATION <config>]
              [TARGET <target>]
              [PROJECT_NAME <projname>] # legacy, causes warning
             )
```

Sets the given `<variable>` to a command-line string of the form:

```
<cmake> --build . [--config <config>] [--target <target>] [-- -i]
```

where `<cmake>` is the location of the `cmake(1)` command-line tool, and `<config>` and `<target>` are the values provided to the `CONFIGURATION` and `TARGET` options, if any. The trailing `-- -i` option is added for Makefile Generators if policy `CMP0061` is not set to `NEW`.

When invoked, this `cmake --build` command line will launch the underlying build system tool.

```
build_command(<cachevariable> <makecommand>)
```

This second signature is deprecated, but still available for backwards compatibility. Use the first signature instead.

It sets the given `<cachevariable>` to a command-line string as above but without the `--target` option. The `<makecommand>` is ignored but should be the full path to devenv, nmake, make or one of the end user build tools for legacy invocations.

Note

In CMake versions prior to 3.0 this command returned a command line that directly invokes the native build tool for the current generator. Their implementation of the `PROJECT_NAME` option had no useful effects, so CMake now warns on use of the option.

create_test_sourcelist

Create a test driver and source list for building test programs.

```
create_test_sourcelist(sourceListName driverName
                       test1 test2 test3
                       EXTRA_INCLUDE include.h
                       FUNCTION function)
```

A test driver is a program that links together many small tests into a single executable. This is useful when building static executables with large libraries to shrink the total required size. The list of source files needed to build the test driver will be in `sourceListName`. `driverName` is the name of the test driver program. The rest of the arguments consist of a list of test source files, can be semicolon separated. Each test source file should have a function in it that is the same name as the file with no extension (foo.cxx should have int foo(int, char*[]);) `driverName` will be able to call each of the tests by name on the command line. If `EXTRA_INCLUDE` is specified, then the next argument is included into the generated file. If `FUNCTION` is specified, then the next argument is taken as a function name that is passed a pointer to ac and av. This can be used to add extra command line processing to each test. The `CMAKE_TESTDRIVER_BEFORE_TESTMAIN` cmake variable can be set to have code that will be placed directly before calling the test main function. `CMAKE_TESTDRIVER_AFTER_TESTMAIN` can be set to have code that will be placed directly after the call to the test main function.

define_property

Define and document custom properties.

```
define_property(<GLOBAL | DIRECTORY | TARGET | SOURCE |
                 TEST | VARIABLE | CACHED_VARIABLE>
                 PROPERTY <name> [INHERITED]
                 BRIEF_DOCS <brief-doc> [docs...]
                 FULL_DOCS <full-doc> [docs...])
```

Define one property in a scope for use with the `set_property()` and `get_property()` commands. This is primarily useful to associate documentation with property names that may be retrieved with the `get_property()` command. The first argument determines the kind of scope in which the property should be used. It must be one of the following:

```
GLOBAL    = associated with the global namespace
DIRECTORY = associated with one directory
TARGET    = associated with one target
SOURCE    = associated with one source file
TEST      = associated with a test named with add_test
VARIABLE  = documents a CMake language variable
CACHED_VARIABLE = documents a CMake cache variable
```

Note that unlike `set_property()` and `get_property()` no actual scope needs to be given; only the kind of scope is important.

The required `PROPERTY` option is immediately followed by the name of the property being defined.

If the `INHERITED` option is given, then the `get_property()` command will chain up to the next higher scope when the requested property is not set in the scope given to the command.

- `DIRECTORY` scope chains to its parent directory's scope, continuing the walk up parent directories until a directory has the property set or there are no more parents. If still not found at the top level directory, it chains to the `GLOBAL` scope.
- `TARGET`, `SOURCE` and `TEST` properties chain to `DIRECTORY` scope, including further chaining up the directories, etc. as needed.

Note that this scope chaining behavior only applies to calls to `get_property()`, `get_directory_property()`, `get_target_property()`, `get_source_file_property()` and `get_test_property()`. There is no inheriting behavior when *setting*properties, so using `APPEND` or `APPEND_STRING` with the `set_property()` command will not consider inherited values when working out the contents to append to.

The `BRIEF_DOCS` and `FULL_DOCS` options are followed by strings to be associated with the property as its brief and full documentation. Corresponding options to the `get_property()` command will retrieve the documentation.

enable_language

Enable a language (CXX/C/Fortran/etc)

```
enable_language(<lang> [OPTIONAL] )
```

This command enables support for the named language in CMake. This is the same as the project command but does not create any of the extra variables that are created by the project command. Example languages are `CXX`, `C`, `CUDA`, `Fortran`, and `ASM`.

If enabling `ASM`, enable it last so that CMake can check whether compilers for other languages like `C` work for assembly too.

This command must be called in file scope, not in a function call. Furthermore, it must be called in the highest directory common to all targets using the named language directly for compiling sources or indirectly through link dependencies. It is simplest to enable all needed languages in the top-level directory of a project.

The `OPTIONAL` keyword is a placeholder for future implementation and does not currently work. Instead you can use the `CheckLanguage` module to verify support before enabling.

enable_testing

Enable testing for current directory and below.

```
enable_testing()
```

Enables testing for this directory and below. See also the `add_test()` command. Note that ctest expects to find a test file in the build directory root. Therefore, this command should be in the source directory root.

export

Export targets from the build tree for use by outside projects.

```
export(EXPORT <export-name> [NAMESPACE <namespace>] [FILE <filename>])
```

Create a file `<filename>` that may be included by outside projects to import targets from the current project's build tree. This is useful during cross-compiling to build utility executables that can run on the host platform in one project and then import them into another project being compiled for the target platform. If the `NAMESPACE` option is given the `<namespace>` string will be prepended to all target names written to the file.

Target installations are associated with the export `<export-name>` using the `EXPORT` option of the `install(TARGETS)` command.

The file created by this command is specific to the build tree and should never be installed. See the `install(EXPORT)` command to export targets from an installation tree.

The properties set on the generated IMPORTED targets will have the same values as the final values of the input TARGETS.

```
export(TARGETS [target1 [target2 [...]]] [NAMESPACE <namespace>]
       [APPEND] FILE <filename> [EXPORT_LINK_INTERFACE_LIBRARIES])
```

This signature is similar to the `EXPORT` signature, but targets are listed explicitly rather than specified as an export-name. If the APPEND option is given the generated code will be appended to the file instead of overwriting it. The EXPORT_LINK_INTERFACE_LIBRARIES keyword, if present, causes the contents of the properties matching `(IMPORTED_)?LINK_INTERFACE_LIBRARIES(_<CONFIG>)?` to be exported, when policy CMP0022 is NEW. If a library target is included in the export but a target to which it links is not included the behavior is unspecified.

Note

Object Libraries under `Xcode` have special handling if multiple architectures are listed in `CMAKE_OSX_ARCHITECTURES`. In this case they will be exported as Interface Libraries with no object files available to clients. This is sufficient to satisfy transitive usage requirements of other targets that link to the object libraries in their implementation.

```
export(PACKAGE <PackageName>)
```

Store the current build directory in the CMake user package registry for package `<PackageName>`. The find_package command may consider the directory while searching for package `<PackageName>`. This helps dependent projects find and use a package from the current project's build tree without help from the user. Note that the entry in the package registry that this command creates works only in conjunction with a package configuration file (`<PackageName>Config.cmake`) that works with the build tree. In some cases, for example for packaging and for system wide installations, it is not desirable to write the user package registry. If the `CMAKE_EXPORT_NO_PACKAGE_REGISTRY` variable is enabled, the `export(PACKAGE)` command will do nothing.

```
export(TARGETS [target1 [target2 [...]]]  [ANDROID_MK <filename>])
```

This signature exports cmake built targets to the android ndk build system by creating an Android.mk file that references the prebuilt targets. The Android NDK supports the use of prebuilt libraries, both static and shared. This allows cmake to build the libraries of a project and make them available to an ndk build system complete with transitive dependencies, include flags and defines required to use the libraries. The signature takes a list of targets and puts them in the Android.mk file specified by the `<filename>` given. This signature can only be used if policy CMP0022 is NEW for all targets given. A error will be issued if that policy is set to OLD for one of the targets.

fltk_wrap_ui

Create FLTK user interfaces Wrappers.

```
fltk_wrap_ui(resultingLibraryName source1
             source2 ... sourceN )
```

Produce .h and .cxx files for all the .fl and .fld files listed. The resulting .h and .cxx files will be added to a variable named `resultingLibraryName_FLTK_UI_SRCS` which should be added to your library.

get_source_file_property

Get a property for a source file.

```
get_source_file_property(VAR file property)
```

Get a property from a source file. The value of the property is stored in the variable `VAR`. If the source property is not found, the behavior depends on whether it has been defined to be an `INHERITED` property or not (see `define_property()`). Non-inherited properties will set `VAR` to "NOTFOUND", whereas inherited properties will search the relevant parent scope as described for the `define_property()` command and if

still unable to find the property, `VAR` will be set to an empty string.

Use `set_source_files_properties()` to set property values. Source file properties usually control how the file is built. One property that is always there is `LOCATION`.

See also the more general `get_property()` command.

get_target_property

Get a property from a target.

```
get_target_property(VAR target property)
```

Get a property from a target. The value of the property is stored in the variable `VAR`. If the target property is not found, the behavior depends on whether it has been defined to be an `INHERITED` property or not (see `define_property()`). Non-inherited properties will set `VAR` to "NOTFOUND", whereas inherited properties will search the relevant parent scope as described for the `define_property()` command and if still unable to find the property, `VAR` will be set to an empty string.

Use `set_target_properties()` to set target property values. Properties are usually used to control how a target is built, but some query the target instead. This command can get properties for any target so far created. The targets do not need to be in the current `CMakeLists.txt` file.

See also the more general `get_property()` command.

get_test_property

Get a property of the test.

```
get_test_property(test property VAR)
```

Get a property from the test. The value of the property is stored in the variable `VAR`. If the test property is not found, the behavior depends on whether it has been defined to be an `INHERITED` property or not (see `define_property()`). Non-inherited properties will set `VAR` to "NOTFOUND", whereas inherited properties will search the relevant parent scope as described for the `define_property()` command and if still unable to find the property, `VAR` will be set to an empty string.

For a list of standard properties you can type `cmake --help-property-list`.

See also the more general `get_property()` command.

include_directories

Add include directories to the build.

```
include_directories([AFTER|BEFORE] [SYSTEM] dir1 [dir2 ...])
```

Add the given directories to those the compiler uses to search for include files. Relative paths are interpreted as relative to the current source directory.

The include directories are added to the `INCLUDE_DIRECTORIES` directory property for the current `CMakeLists` file. They are also added to the `INCLUDE_DIRECTORIES` target property for each target in the current `CMakeLists` file. The target property values are the ones used by the generators.

By default the directories specified are appended onto the current list of directories. This default behavior can be changed by setting `CMAKE_INCLUDE_DIRECTORIES_BEFORE` to `ON`. By using `AFTER` or `BEFORE` explicitly, you can select between appending and prepending, independent of the default.

If the `SYSTEM` option is given, the compiler will be told the directories are meant as system include directories on some platforms. Signalling this setting might achieve effects such as the compiler skipping warnings, or these fixed-install system files not being considered in dependency calculations - see compiler docs.

Arguments to `include_directories` may use "generator expressions" with the syntax "$<...>". See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

Note

Prefer the `target_include_directories()` command to add include directories to individual targets and optionally propagate/export them to dependents.

include_external_msproject

Include an external Microsoft project file in a workspace.

```
include_external_msproject(projectname location
                           [TYPE projectTypeGUID]
                           [GUID projectGUID]
                           [PLATFORM platformName]
                           dep1 dep2 ...)
```

Includes an external Microsoft project in the generated workspace file. Currently does nothing on UNIX. This will create a target named [projectname]. This can be used in the `add_dependencies()` command to make things depend on the external project.

`TYPE`, `GUID` and `PLATFORM` are optional parameters that allow one to specify the type of project, id (GUID) of the project and the name of the target platform. This is useful for projects requiring values other than the default (e.g. WIX projects).

If the imported project has different configuration names than the current project, set the `MAP_IMPORTED_CONFIG_` target property to specify the mapping.

include_regular_expression

Set the regular expression used for dependency checking.

```
include_regular_expression(regex_match [regex_complain])
```

Set the regular expressions used in dependency checking. Only files matching `regex_match` will be traced as dependencies. Only files matching `regex_complain` will generate warnings if they cannot be found (standard header paths are not searched). The defaults are:

```
regex_match    = "^.*$" (match everything)
regex_complain = "^$" (match empty string only)
```

install

Specify rules to run at install time.

Synopsis

```
install(TARGETS <target>... [...])
install({FILES | PROGRAMS} <file>... DESTINATION <dir> [...])
install(DIRECTORY <dir>... DESTINATION <dir> [...])
install(SCRIPT <file> [...])
install(CODE <code> [...])
install(EXPORT <export-name> DESTINATION <dir> [...])
```

Introduction

This command generates installation rules for a project. Rules specified by calls to this command within a source directory are executed in order during installation. The order across directories is not defined.

There are multiple signatures for this command. Some of them define installation options for files and targets. Options common to multiple signatures are covered here but they are valid only for signatures that specify them. The common options are:

- `DESTINATION`

  Specify the directory on disk to which a file will be installed. If a full path (with a leading slash or drive letter) is given it is used directly. If a relative path is given it is interpreted relative to the value of the `CMAKE_INSTALL_PREFIX` variable. The prefix can be relocated at install time using the `DESTDIR` mechanism explained in the `CMAKE_INSTALL_PREFIX` variable documentation.

- `PERMISSIONS`

  Specify permissions for installed files. Valid permissions are `OWNER_READ`, `OWNER_WRITE`, `OWNER_EXECUTE`, `GROUP_READ`, `GROUP_WRITE`, `GROUP_EXECUTE`, `WORLD_READ`, `WORLD_WRITE`, `WORLD_EXECUTE`, `SETUID`, and `SETGID`. Permissions that do not make sense on certain platforms are ignored on those platforms.

- `CONFIGURATIONS`

  Specify a list of build configurations for which the install rule applies (Debug, Release, etc.). Note that the values specified for this option only apply to options listed AFTER the `CONFIGURATIONS` option. For example, to set separate install paths for the Debug and Release configurations, do the following: `install(TARGETS target CONFIGURATIONS Debug RUNTIME DESTINATION Debug/bin)`

`install(TARGETS target CONFIGURATIONS Release RUNTIME DESTINATION Release/bin)` Note that `CONFIGURATIONS` appears BEFORE `RUNTIME DESTINATION`.

- `COMPONENT`

  Specify an installation component name with which the install rule is associated, such as “runtime” or “development”. During component-specific installation only install rules associated with the given component name will be executed. During a full installation all components are installed unless marked with `EXCLUDE_FROM_ALL`. If `COMPONENT` is not provided a default component “Unspecified” is created. The default component name may be controlled with the `CMAKE_INSTALL_DEFAULT_COMPONENT_NAME` variable.

- `EXCLUDE_FROM_ALL`

  Specify that the file is excluded from a full installation and only installed as part of a component-specific installation

- `RENAME`

  Specify a name for an installed file that may be different from the original file. Renaming is allowed only when a single file is installed by the command.

- `OPTIONAL`

  Specify that it is not an error if the file to be installed does not exist.

Command signatures that install files may print messages during installation. Use the `CMAKE_INSTALL_MESSAGE` variable to control which messages are printed.

Many of the `install()` variants implicitly create the directories containing the installed files.
If `CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS` is set, these directories will be created with the permissions specified. Otherwise, they will be created according to the uname rules on Unix-like platforms. Windows platforms are unaffected.

Installing Targets

```
install(TARGETS targets... [EXPORT <export-name>]
        [[ARCHIVE|LIBRARY|RUNTIME|OBJECTS|FRAMEWORK|BUNDLE|
          PRIVATE_HEADER|PUBLIC_HEADER|RESOURCE]
         [DESTINATION <dir>]
         [PERMISSIONS permissions...]
         [CONFIGURATIONS [Debug|Release|...]]
         [COMPONENT <component>]
         [NAMELINK_COMPONENT <component>]
         [OPTIONAL] [EXCLUDE_FROM_ALL]
         [NAMELINK_ONLY|NAMELINK_SKIP]
        ] [...]
        [INCLUDES DESTINATION [<dir> ...]]
        )
```

The `TARGETS` form specifies rules for installing targets from a project. There are several kinds of target files that may be installed:

- `ARCHIVE`

Static libraries are treated as `ARCHIVE` targets, except those marked with the `FRAMEWORK` property on macOS (see `FRAMEWORK` below.) For DLL platforms (all Windows-based systems including Cygwin), the DLL import library is treated as an `ARCHIVE` target.

- `LIBRARY`

  Module libraries are always treated as `LIBRARY` targets. For non- DLL platforms shared libraries are treated as `LIBRARY` targets, except those marked with the `FRAMEWORK` property on macOS (see `FRAMEWORK` below.)

- `RUNTIME`

  Executables are treated as `RUNTIME` objects, except those marked with the `MACOSX_BUNDLE` property on macOS (see `BUNDLE` below.) For DLL platforms (all Windows-based systems including Cygwin), the DLL part of a shared library is treated as a `RUNTIME` target.

- `OBJECTS`

  Object libraries (a simple group of object files) are always treated as `OBJECTS` targets.

- `FRAMEWORK`

  Both static and shared libraries marked with the `FRAMEWORK` property are treated as `FRAMEWORK` targets on macOS.

- `BUNDLE`

  Executables marked with the `MACOSX_BUNDLE` property are treated as `BUNDLE` targets on macOS.

- `PUBLIC_HEADER`

  Any `PUBLIC_HEADER` files associated with a library are installed in the destination specified by the `PUBLIC_HEADER` argument on non-Apple platforms. Rules defined by this argument are ignored for `FRAMEWORK` libraries on Apple platforms because the associated files are installed into the appropriate locations inside the framework folder. See `PUBLIC_HEADER` for details.

- `PRIVATE_HEADER`

  Similar to `PUBLIC_HEADER`, but for `PRIVATE_HEADER` files. See `PRIVATE_HEADER` for details.

- `RESOURCE`

  Similar to `PUBLIC_HEADER` and `PRIVATE_HEADER`, but for `RESOURCE` files. See `RESOURCE` for details.

For each of these arguments given, the arguments following them only apply to the target or file type specified in the argument. If none is given, the installation properties apply to all target types. If only one is given then only targets of that type will be installed (which can be used to install just a DLL or just an import library.)

In addition to the common options listed above, each target can accept the following additional arguments:

- `NAMELINK_COMPONENT`

  On some platforms a versioned shared library has a symbolic link such as: `lib<name>.so -> lib<name>.so.1` where `lib<name>.so.1` is the soname of the library and `lib<name>.so` is a “namelink” allowing linkers to find the library when given `-l<name>`. The `NAMELINK_COMPONENT` option is similar to the `COMPONENT` option, but it changes the installation component of a shared library namelink if one is generated. If not specified, this defaults to the value of `COMPONENT`. It is an error to use this parameter outside of a `LIBRARY` block.Consider the following example: `install(TARGETS mylib LIBRARY`

`DESTINATION lib COMPONENT Libraries NAMELINK_COMPONENT Development PUBLIC_HEADER DESTINATION include COMPONENT Development )` In this scenario, if you choose to install only the `Development` component, both the headers and namelink will be installed without the library. (If you don't also install the `Libraries` component, the namelink will be a dangling symlink, and projects that link to the library will have build errors.) If you install only the `Libraries` component, only the library will be installed, without the headers and namelink.This option is typically used for package managers that have separate runtime and development packages. For example, on Debian systems, the library is expected to be in the runtime package, and the headers and namelink are expected to be in the development package.See the `VERSION` and `SOVERSION` target properties for details on creating versioned shared libraries.

- `NAMELINK_ONLY`

  This option causes the installation of only the namelink when a library target is installed. On platforms where versioned shared libraries do not have namelinks or when a library is not versioned, the `NAMELINK_ONLY` option installs nothing. It is an error to use this parameter outside of a `LIBRARY` block.When `NAMELINK_ONLY` is given, either `NAMELINK_COMPONENT` or `COMPONENT` may be used to specify the installation component of the namelink, but `COMPONENT` should generally be preferred.

- `NAMELINK_SKIP`

  Similar to `NAMELINK_ONLY`, but it has the opposite effect: it causes the installation of library files other than the namelink when a library target is installed. When neither `NAMELINK_ONLY` or `NAMELINK_SKIP` are given, both portions are installed. On platforms where versioned shared libraries do not have symlinks or when a library is not versioned, `NAMELINK_SKIP` installs the library. It is an error to use this parameter outside of a `LIBRARY` block.If `NAMELINK_SKIP` is specified, `NAMELINK_COMPONENT` has no effect. It is not recommended to use `NAMELINK_SKIP` in conjunction with `NAMELINK_COMPONENT`.

The `install(TARGETS)` command can also accept the following options at the top level:

- `EXPORT`

  This option associates the installed target files with an export called `<export-name>`. It must appear before any target options. To actually install the export file itself, call `install(EXPORT)`, documented below.

- `INCLUDES DESTINATION`

  This option specifies a list of directories which will be added to the `INTERFACE_INCLUDE_DIRECTORIES` target property of the `<targets>` when exported by the `install(EXPORT)` command. If a relative path is specified, it is treated as relative to the `$<INSTALL_PREFIX>`.

One or more groups of properties may be specified in a single call to the `TARGETS` form of this command. A target may be installed more than once to different locations. Consider hypothetical targets `myExe`, `mySharedLib`, and `myStaticLib`. The code:

```
install(TARGETS myExe mySharedLib myStaticLib
        RUNTIME DESTINATION bin
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib/static)
install(TARGETS mySharedLib DESTINATION /some/full/path)
```

will install `myExe` to `<prefix>/bin` and `myStaticLib` to `<prefix>/lib/static`. On non-DLL platforms `mySharedLib` will be installed to `<prefix>/lib` and `/some/full/path`. On DLL platforms the `mySharedLib` DLL will be installed to `<prefix>/bin` and `/some/full/path` and its import library will be installed to `<prefix>/lib/static` and `/some/full/path`.

Interface Libraries may be listed among the targets to install. They install no artifacts but will be included in an associated `EXPORT`. If Object Libraries are listed but given no destination for their object files, they will be exported as Interface Libraries. This is sufficient to satisfy transitive usage requirements of other targets that link to the object libraries in their implementation.

Installing a target with the `EXCLUDE_FROM_ALL` target property set to `TRUE` has undefined behavior.

`install(TARGETS)` can install targets that were created in other directories. When using such cross-directory install rules, running `make install` (or similar) from a subdirectory will not guarantee that targets from other directories are up-to-date. You can use `target_link_libraries()` or `add_dependencies()` to ensure that such out-of-directory targets are built before the subdirectory-specific install rules are run.

The install destination given to the target install `DESTINATION` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions.

Installing Files

```
install(<FILES|PROGRAMS> files... DESTINATION <dir>
        [PERMISSIONS permissions...]
        [CONFIGURATIONS [Debug|Release|...]]
        [COMPONENT <component>]
        [RENAME <name>] [OPTIONAL] [EXCLUDE_FROM_ALL])
```

The `FILES` form specifies rules for installing files for a project. File names given as relative paths are interpreted with respect to the current source directory. Files installed by this form are by default given permissions `OWNER_WRITE`, `OWNER_READ`, `GROUP_READ`, and `WORLD_READ` if no `PERMISSIONS` argument is given.

The `PROGRAMS` form is identical to the `FILES` form except that the default permissions for the installed file also include `OWNER_EXECUTE`, `GROUP_EXECUTE`, and `WORLD_EXECUTE`. This form is intended to install programs that are not targets, such as shell scripts. Use the `TARGETS` form to install targets built within the project.

The list of `files...` given to `FILES` or `PROGRAMS` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. However, if any item begins in a generator expression it must evaluate to a full path.

The install destination given to the files install `DESTINATION` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions.

Installing Directories

```
install(DIRECTORY dirs... DESTINATION <dir>
        [FILE_PERMISSIONS permissions...]
        [DIRECTORY_PERMISSIONS permissions...]
        [USE_SOURCE_PERMISSIONS] [OPTIONAL] [MESSAGE_NEVER]
        [CONFIGURATIONS [Debug|Release|...]]
        [COMPONENT <component>] [EXCLUDE_FROM_ALL]
        [FILES_MATCHING]
        [[PATTERN <pattern> | REGEX <regex>]
         [EXCLUDE] [PERMISSIONS permissions...]] [...])
```

The `DIRECTORY` form installs contents of one or more directories to a given destination. The directory structure is copied verbatim to the destination. The last component of each directory name is appended to the destination directory but a trailing slash may be used to avoid this because it leaves the last component empty. Directory names given as relative paths are interpreted with respect to the current source directory. If no input directory names are given the destination directory will be created but nothing will be installed into it. The `FILE_PERMISSIONS` and `DIRECTORY_PERMISSIONS` options specify permissions given to files and directories in the destination. If `USE_SOURCE_PERMISSIONS` is specified and `FILE_PERMISSIONS` is not, file permissions will be copied from the source directory structure. If no permissions are specified files will be given the default permissions specified in the `FILES` form of the command, and the directories will be given the default permissions specified in the `PROGRAMS` form of the command.

The `MESSAGE_NEVER` option disables file installation status output.

Installation of directories may be controlled with fine granularity using the `PATTERN` or `REGEX` options. These “match” options specify a globbing pattern or regular expression to match directories or files encountered within input directories. They may be used to apply certain options (see below) to a subset of the files and directories encountered. The full path to each input file or directory (with forward slashes) is matched against the expression. A `PATTERN` will match only complete file names: the portion of the full path matching the pattern must occur at the end of the file name and be preceded by a slash. A `REGEX` will match any portion of the full path but it may use `/` and `$` to simulate the `PATTERN` behavior. By default all files and directories are installed whether or not they are matched. The `FILES_MATCHING` option may be given before the first match option to disable installation of files (but not directories) not matched by any expression. For example, the code

```
install(DIRECTORY src/ DESTINATION include/myproj
        FILES_MATCHING PATTERN "*.h")
```

will extract and install header files from a source tree.

Some options may follow a `PATTERN` or `REGEX` expression and are applied only to files or directories matching them. The `EXCLUDE` option will skip the matched file or directory. The `PERMISSIONS` option overrides the permissions setting for the matched file or directory. For example the code

```
install(DIRECTORY icons scripts/ DESTINATION share/myproj
        PATTERN "CVS" EXCLUDE
        PATTERN "scripts/*"
        PERMISSIONS OWNER_EXECUTE OWNER_WRITE OWNER_READ
                    GROUP_EXECUTE GROUP_READ)
```

will install the `icons` directory to `share/myproj/icons` and the `scripts` directory to `share/myproj`. The icons will get default file permissions, the scripts will be given specific permissions, and any `CVS` directories will be excluded.

The list of `dirs...` given to `DIRECTORY` and the install destination given to the directory install `DESTINATION` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions.

Custom Installation Logic

```
install([[SCRIPT <file>] [CODE <code>]]
        [COMPONENT <component>] [EXCLUDE_FROM_ALL] [...])
```

The `SCRIPT` form will invoke the given CMake script files during installation. If the script file name is a relative path it will be interpreted with respect to the current source directory. The `CODE` form will invoke the given CMake code during installation. Code is specified as a single argument inside a double-quoted string. For example, the code

```
install(CODE "MESSAGE(\"Sample install message.\")")
```

will print a message during installation.

Installing Exports

```
install(EXPORT <export-name> DESTINATION <dir>
        [NAMESPACE <namespace>] [[FILE <name>.cmake]|
        [PERMISSIONS permissions...]
        [CONFIGURATIONS [Debug|Release|...]]
        [EXPORT_LINK_INTERFACE_LIBRARIES]
        [COMPONENT <component>]
        [EXCLUDE_FROM_ALL])
install(EXPORT_ANDROID_MK <export-name> DESTINATION <dir> [...])
```

The `EXPORT` form generates and installs a CMake file containing code to import targets from the installation tree into another project. Target installations are associated with the export `<export-name>` using the `EXPORT` option of the `install(TARGETS)` signature documented above. The `NAMESPACE` option will prepend `<namespace>` to the target names as they are written to the import file. By default the generated file will be called `<export-name>.cmake` but the `FILE` option may be used to specify a different name. The value given to the `FILE` option must be a file name with the `.cmake` extension. If a `CONFIGURATIONS` option is given then the file will only be installed when one of the named configurations is installed. Additionally, the generated import file will reference only the matching target configurations. The `EXPORT_LINK_INTERFACE_LIBRARIES` keyword, if present, causes the contents of the properties matching `(IMPORTED_)?LINK_INTERFACE_LIBRARIES(_<CONFIG>)?` to be exported, when policy `CMP0022` is `NEW`.

When a `COMPONENT` option is given, the listed `<component>` implicitly depends on all components mentioned in the export set. The exported `<name>.cmake` file will require each of the exported components to be present in order for dependent projects to build properly. For example, a project may define components `Runtime` and `Development`, with shared libraries going into the `Runtime` component and static libraries and headers going into the `Development` component. The export set would also typically be part of the `Development` component, but it would export targets from both the `Runtime` and `Development` components. Therefore, the `Runtime` component would need to be installed if the `Development` component was installed, but not vice versa. If the `Development` component was installed without the `Runtime` component, dependent projects that try to link against it would have build errors. Package managers, such as APT and RPM, typically handle this by listing the `Runtime` component as a dependency of the `Development` component in the package metadata, ensuring that the library is always installed if the headers and CMake export file are present.

In addition to cmake language files, the `EXPORT_ANDROID_MK` mode maybe used to specify an export to the android ndk build system. This mode accepts the same options as the normal export mode. The Android NDK supports the use of prebuilt libraries, both static and shared. This allows cmake to build the libraries of a project and make them available to an ndk build system complete with transitive dependencies, include flags and defines required to use the libraries.

The `EXPORT` form is useful to help outside projects use targets built and installed by the current project. For example, the code

```
install(TARGETS myexe EXPORT myproj DESTINATION bin)
install(EXPORT myproj NAMESPACE mp_ DESTINATION lib/myproj)
install(EXPORT_ANDROID_MK myexp DESTINATION share/ndk-modules)
```

will install the executable myexe to `<prefix>/bin` and code to import it in the file `<prefix>/lib/myproj/myproj.cmake` and `<prefix>/share/ndk-modules/Android.mk`. An outside project may load this file with the include command and reference the `myexe` executable from the installation tree using the imported target name `mp_myexe` as if the target were built in its own tree.

Note

This command supercedes the `install_targets()` command and the `PRE_INSTALL_SCRIPT` and `POST_INSTALL_SCRIPT` target properties. It also replaces the `FILES` forms of the `install_files()` and `install_programs()` commands. The processing order of these install rules relative to those generated by `install_targets()`, `install_files()`, and `install_programs()` commands is not defined.

Generated Installation Script

The `install()` command generates a file, `cmake_install.cmake`, inside the build directory, which is used internally by the generated install target and by CPack. You can also invoke this script manually with `cmake -P`. This script accepts several variables:

- `COMPONENT`

  Set this variable to install only a single CPack component as opposed to all of them. For example, if you only want to install the `Development` component, run `cmake -DCOMPONENT=Development -P cmake_install.cmake`.

- `BUILD_TYPE`

  Set this variable to change the build type if you are using a multi-config generator. For example, to install with the `Debug` configuration, run `cmake -DBUILD_TYPE=Debug -P cmake_install.cmake`.

- `DESTDIR`

  This is an environment variable rather than a CMake variable. It allows you to change the installation prefix on UNIX systems. See `DESTDIR` for details.

link_directories

Add directories in which the linker will look for libraries.

```
link_directories([AFTER|BEFORE] directory1 [directory2 ...])
```

Add the paths in which the linker should search for libraries. Relative paths given to this command are interpreted as relative to the current source directory, see `CMP0015`.

The directories are added to the `LINK_DIRECTORIES` directory property for the current `CMakeLists.txt` file, converting relative paths to absolute as needed. The command will apply only to targets created after it is called.

By default the directories specified are appended onto the current list of directories. This default behavior can be changed by setting `CMAKE_LINK_DIRECTORIES_BEFORE` to `ON`. By using `AFTER` or `BEFORE` explicitly, you can select between appending and prepending, independent of the default.

Arguments to `link_directories` may use "generator expressions" with the syntax "$<...>". See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

Note

This command is rarely necessary and should be avoided where there are other choices. Prefer to pass full absolute paths to libraries where possible, since this ensures the correct library will always be linked. The `find_library()` command provides the full path, which can generally be used directly in calls to `target_link_libraries()`. Situations where a library search path may be needed include:

- Project generators like Xcode where the user can switch target architecture at build time, but a full path to a library cannot be used because it only provides one architecture (i.e. it is not a universal binary).
- Libraries may themselves have other private library dependencies that expect to be found via `RPATH` mechanisms, but some linkers are not able to fully decode those paths (e.g. due to the presence of things like `$ORIGIN`).

If a library search path must be provided, prefer to localize the effect where possible by using the `target_link_directories()` command rather than `link_directories()`. The target-specific command can also control how the search directories propagate to other dependent targets.

link_libraries

Link libraries to all targets added later.

```
link_libraries([item1 [item2 [...]]]
               [[debug|optimized|general] <item>] ...)
```

Specify libraries or flags to use when linking any targets created later in the current directory or below by commands such as `add_executable()` or `add_library()`. See the `target_link_libraries()` command for meaning of arguments.

Note

The `target_link_libraries()` command should be preferred whenever possible. Library dependencies are chained automatically, so directory-wide specification of link libraries is rarely needed.

load_cache

Load in the values from another project's CMake cache.

```
load_cache(pathToCacheFile READ_WITH_PREFIX
           prefix entry1...)
```

Read the cache and store the requested entries in variables with their name prefixed with the given prefix. This only reads the values, and does not create entries in the local project's cache.

```
load_cache(pathToCacheFile [EXCLUDE entry1...]
           [INCLUDE_INTERNALS entry1...])
```

Load in the values from another cache and store them in the local project's cache as internal entries. This is useful for a project that depends on another project built in a different tree. `EXCLUDE` option can be used to provide a list of entries to be excluded. `INCLUDE_INTERNALS` can be used to provide a list of internal entries to be included. Normally, no internal entries are brought in. Use of this form of the command is strongly discouraged, but it is provided for backward compatibility.

project¶

Sets project details such as name, version, etc. and enables languages.

```
project(<PROJECT-NAME> [LANGUAGES] [<language-name>...])
project(<PROJECT-NAME>
        [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]
        [DESCRIPTION <project-description-string>]
        [HOMEPAGE_URL <url-string>]
        [LANGUAGES <language-name>...])
```

Sets the name of the project and stores the name in the `PROJECT_NAME` variable. Additionally this sets variables

- `PROJECT_SOURCE_DIR`, `_SOURCE_DIR`
- `PROJECT_BINARY_DIR`, `_BINARY_DIR`

If `VERSION` is specified, given components must be non-negative integers. If `VERSION` is not specified, the default version is the empty string. The `VERSION` option may not be used unless policy `CMP0048` is set to `NEW`.

The `project()` command stores the version number and its components in variables

- `PROJECT_VERSION`, `_VERSION`
- `PROJECT_VERSION_MAJOR`, `_VERSION_MAJOR`
- `PROJECT_VERSION_MINOR`, `_VERSION_MINOR`
- `PROJECT_VERSION_PATCH`, `_VERSION_PATCH`
- `PROJECT_VERSION_TWEAK`, `_VERSION_TWEAK`

Variables corresponding to unspecified versions are set to the empty string (if policy `CMP0048` is set to `NEW`).

If the optional `DESCRIPTION` is given, then `PROJECT_DESCRIPTION` and `_DESCRIPTION` will be set to its argument. These variables will be cleared if `DESCRIPTION` is not given. The description is expected to be a relatively short string, usually no more than a few words.

The optional `HOMEPAGE_URL` sets the analogous variables `PROJECT_HOMEPAGE_URL` and `_HOMEPAGE_URL`. When this option is given, the URL provided should be the canonical home for the project. These variables will be cleared if `HOMEPAGE_URL` is not given.

Note that the description and homepage URL may be used as defaults for things like packaging meta-data, documentation, etc.

Optionally you can specify which languages your project supports. Example languages include `C`, `CXX` (i.e. C++), `CUDA`, `Fortran`, and `ASM`. By default `C` and `CXX` are enabled if no language options are given. Specify language `NONE`, or use the `LANGUAGES` keyword and list no languages, to skip enabling any languages.

If enabling `ASM`, list it last so that CMake can check whether compilers for other languages like `C` work for assembly too.

If a variable exists called `CMAKE_PROJECT_<PROJECT-NAME>_INCLUDE`, the file pointed to by that variable will be included as the last step of the project command.

The top-level `CMakeLists.txt` file for a project must contain a literal, direct call to the `project()` command; loading one through the `include()` command is not sufficient. If no such call exists CMake will implicitly add one to the top that enables the default languages (`C` and `CXX`). The name of the project set in the top level `CMakeLists.txt` file is available from the `CMAKE_PROJECT_NAME` variable, its description from `CMAKE_PROJECT_DESCRIPTION`, its homepage URL from `CMAKE_PROJECT_HOMEPAGE_URL` and its version from `CMAKE_PROJECT_VERSION`.

Note

Call the `cmake_minimum_required()` command at the beginning of the top-level `CMakeLists.txt` file even before calling the `project()` command. It is important to establish version and policy settings before invoking other commands whose behavior they may affect. See also policy `CMP0000`.

qt_wrap_cpp

Create Qt Wrappers.

```
qt_wrap_cpp(resultingLibraryName DestName
            SourceLists ...)
```

Produce moc files for all the .h files listed in the SourceLists. The moc files will be added to the library using the `DestName` source list.

qt_wrap_ui

Create Qt user interfaces Wrappers.

```
qt_wrap_ui(resultingLibraryName HeadersDestName
           SourcesDestName SourceLists ...)
```

Produce .h and .cxx files for all the .ui files listed in the `SourceLists`. The .h files will be added to the library using the `HeadersDestNamesource` list. The .cxx files will be added to the library using the `SourcesDestNamesource` list.

remove_definitions

Removes -D define flags added by `add_definitions()`.

```
remove_definitions(-DFOO -DBAR ...)
```

Removes flags (added by `add_definitions()`) from the compiler command line for sources in the current directory and below.

set_source_files_properties

Source files can have properties that affect how they are built.

```
set_source_files_properties([file1 [file2 [...]]]
                            PROPERTIES prop1 value1
                            [prop2 value2 [...]])
```

Set properties associated with source files using a key/value paired list. See Properties on Source Files for the list of properties known to CMake. Source file properties are visible only to targets added in the same directory (CMakeLists.txt).

set_target_properties

Targets can have properties that affect how they are built.

```
set_target_properties(target1 target2 ...
                      PROPERTIES prop1 value1
                      prop2 value2 ...)
```

Set properties on targets. The syntax for the command is to list all the targets you want to change, and then provide the values you want to set next. You can use any prop value pair you want and extract it later with the `get_property()` or `get_target_property()` command.

See also the `set_property(TARGET)` command.

See Properties on Targets for the list of properties known to CMake.

set_tests_properties

Set a property of the tests.

```
set_tests_properties(test1 [test2...] PROPERTIES prop1 value1 prop2 value2)
```

Set a property for the tests. If the test is not found, CMake will report an error. `Generator expressions` will be expanded the same as supported by the test's `add_test()` call. See Properties on Tests for the list of properties known to CMake.

source_group

Define a grouping for source files in IDE project generation. There are two different signatures to create source groups.

```
source_group(<name> [FILES <src>...] [REGULAR_EXPRESSION <regex>])
source_group(TREE <root> [PREFIX <prefix>] [FILES <src>...])
```

Defines a group into which sources will be placed in project files. This is intended to set up file tabs in Visual Studio. The options are:

- `TREE`

  CMake will automatically detect, from `<src>` files paths, source groups it needs to create, to keep structure of source groups analogically to the actual files and directories structure in the project. Paths of `<src>` files will be cut to be relative to `<root>`.

- `PREFIX`

  Source group and files located directly in `<root>` path, will be placed in `<prefix>` source groups.

- `FILES`

  Any source file specified explicitly will be placed in group `<name>`. Relative paths are interpreted with respect to the current source directory.

- `REGULAR_EXPRESSION`

  Any source file whose name matches the regular expression will be placed in group `<name>`.

If a source file matches multiple groups, the *last* group that explicitly lists the file with `FILES` will be favored, if any. If no group explicitly lists the file, the *last* group whose regular expression matches the file will be favored.

The `<name>` of the group and `<prefix>` argument may contain backslashes to specify subgroups:

```
source_group(outer\\inner ...)
source_group(TREE <root> PREFIX sources\\inc ...)
```

For backwards compatibility, the short-hand signature

```
source_group(<name> <regex>)
```

is equivalent to

```
source_group(<name> REGULAR_EXPRESSION <regex>)
```

target_compile_definitions

Add compile definitions to a target.

```
target_compile_definitions(<target>
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

Specify compile definitions to use when compiling a given `<target>`. The named `<target>` must have been created by a command such as `add_executable()` or `add_library()` and must not be an ALIAS target.

The `INTERFACE`, `PUBLIC` and `PRIVATE` keywords are required to specify the scope of the following arguments. `PRIVATE` and `PUBLIC` items will populate the `COMPILE_DEFINITIONS` property of `<target>`. `PUBLIC` and `INTERFACE` items will populate the `INTERFACE_COMPILE_DEFINITIONS` property of `<target>`. (IMPORTED targets only support `INTERFACE` items.) The following arguments specify compile definitions. Repeated calls for the same `<target>` append items in the order called.

Arguments to `target_compile_definitions` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

Any leading `-D` on an item will be removed. Empty items are ignored. For example, the following are all equivalent:

```
target_compile_definitions(foo PUBLIC FOO)
target_compile_definitions(foo PUBLIC -DFOO)  # -D removed
target_compile_definitions(foo PUBLIC "" FOO) # "" ignored
target_compile_definitions(foo PUBLIC -D FOO) # -D becomes "", then ignored
```

target_compile_features

Add expected compiler features to a target.

```
target_compile_features(<target> <PRIVATE|PUBLIC|INTERFACE> <feature> [...])
```

Specify compiler features required when compiling a given target. If the feature is not listed in the `CMAKE_C_COMPILE_FEATURES` variable or `CMAKE_CXX_COMPILE_FEATURES` variable, then an error will be reported by CMake. If the use of the feature requires an additional compiler flag, such as `-std=gnu++11`, the flag will be added automatically.

The `INTERFACE`, `PUBLIC` and `PRIVATE` keywords are required to specify the scope of the features. `PRIVATE` and `PUBLIC` items will populate the `COMPILE_FEATURES` property of `<target>`. `PUBLIC` and `INTERFACE` items will populate the `INTERFACE_COMPILE_FEATURES` property of `<target>`. (IMPORTED targets only support `INTERFACE` items.) Repeated calls for the same `<target>` append items.

The named `<target>` must have been created by a command such as `add_executable()` or `add_library()` and must not be an ALIAS target.

Arguments to `target_compile_features` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-compile-features(7)` manual for information on compile features and a list of supported compilers.

target_compile_options

Add compile options to a target.

```
target_compile_options(<target> [BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

Specify compile options to use when compiling a given target. The named `<target>` must have been created by a command such as `add_executable()` or `add_library()` and must not be an ALIAS target.

If `BEFORE` is specified, the content will be prepended to the property instead of being appended.

This command can be used to add any options, but alternative commands exist to add preprocessor definitions (`target_compile_definitions()` and `add_compile_definitions()`) or include directories (`target_include_directories()` and `include_directories()`). See documentation of the `directory` and `target` `COMPILE_OPTIONS` properties.

The `INTERFACE`, `PUBLIC` and `PRIVATE` keywords are required to specify the scope of the following arguments. `PRIVATE` and `PUBLIC` items will populate the `COMPILE_OPTIONS` property of `<target>`. `PUBLIC` and `INTERFACE` items will populate the `INTERFACE_COMPILE_OPTIONS` property of `<target>`. (IMPORTED targets only support `INTERFACE` items.) The following arguments specify compile options. Repeated calls for the same `<target>` append items in the order called.

Arguments to `target_compile_options` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

The final set of compile or link options used for a target is constructed by accumulating options from the current target and the usage requirements of it dependencies. The set of options is de-duplicated to avoid repetition. While beneficial for individual options, the de-duplication step can break up option groups. For example, `-D A -D B` becomes `-D A B`. One may specify a group of options using shell-like quoting along with a `SHELL:` prefix. The `SHELL:` prefix is dropped and the rest of the option string is parsed using the `separate_arguments()` `UNIX_COMMAND` mode. For example, `"SHELL:-D A" "SHELL:-D B"` becomes `-D A -D B`.

target_include_directories

Add include directories to a target.

```
target_include_directories(<target> [SYSTEM] [BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

Specify include directories to use when compiling a given target. The named `<target>` must have been created by a command such as `add_executable()` or `add_library()` and must not be an ALIAS target.

If `BEFORE` is specified, the content will be prepended to the property instead of being appended.

The `INTERFACE`, `PUBLIC` and `PRIVATE` keywords are required to specify the scope of the following arguments. `PRIVATE` and `PUBLIC` items will populate the `INCLUDE_DIRECTORIES` property of `<target>`. `PUBLIC` and `INTERFACE` items will populate the `INTERFACE_INCLUDE_DIRECTORIES` property of `<target>`. (IMPORTED targets only support `INTERFACE` items.) The following arguments specify include directories.

Specified include directories may be absolute paths or relative paths. Repeated calls for the same append items in the order called. If `SYSTEM` is specified, the compiler will be told the directories are meant as system include directories on some platforms (signalling this setting might achieve effects such as the compiler skipping warnings, or these fixed-install system files not being considered in dependency calculations - see compiler docs). If `SYSTEM` is used together with `PUBLIC` or `INTERFACE`, the `INTERFACE_SYSTEM_INCLUDE_DIRECTORIES` target property will be populated with the specified directories.

Arguments to `target_include_directories` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

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:

```
target_include_directories(mylib PUBLIC
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include/mylib>
  $<INSTALL_INTERFACE:include/mylib>  # <prefix>/include/mylib
)
```

Creating Relocatable Packages

Note that it is not advisable to populate the `INSTALL_INTERFACE` of the `INTERFACE_INCLUDE_DIRECTORIES` of a target with absolute paths to the include directories of dependencies. That would hard-code into installed packages the include directory paths for dependencies **as found on the machine the package was made on**.

The `INSTALL_INTERFACE` of the `INTERFACE_INCLUDE_DIRECTORIES` is only suitable for specifying the required include directories for headers provided with the target itself, not those provided by the transitive dependencies listed in its `INTERFACE_LINK_LIBRARIES` target property. Those dependencies should themselves be targets that specify their own header locations in `INTERFACE_INCLUDE_DIRECTORIES`.

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_link_directories

Add link directories to a target.

```
target_link_directories(<target> [BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

Specify the paths in which the linker should search for libraries when linking a given target. Each item can be an absolute or relative path, with the latter being interpreted as relative to the current source directory. These items will be added to the link command.

The named `<target>` must have been created by a command such as `add_executable()` or `add_library()` and must not be an ALIAS target.

The `INTERFACE`, `PUBLIC` and `PRIVATE` keywords are required to specify the scope of the items that follow them. `PRIVATE` and `PUBLIC` items will populate the `LINK_DIRECTORIES` property of `<target>`. `PUBLIC` and `INTERFACE` items will populate the `INTERFACE_LINK_DIRECTORIES` property of `<target>` (IMPORTED targets only support `INTERFACE` items). Each item specifies a link directory and will be converted to an absolute path if necessary before adding it to the relevant property. Repeated calls for the same `<target>` append items in the order called.

If `BEFORE` is specified, the content will be prepended to the relevant property instead of being appended.

Arguments to `target_link_directories` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

Note

This command is rarely necessary and should be avoided where there are other choices. Prefer to pass full absolute paths to libraries where possible, since this ensures the correct library will always be linked. The `find_library()` command provides the full path, which can generally be used directly in calls to `target_link_libraries()`. Situations where a library search path may be needed include:

- Project generators like Xcode where the user can switch target architecture at build time, but a full path to a library cannot be used because it only provides one architecture (i.e. it is not a universal binary).

- Libraries may themselves have other private library dependencies that expect to be found via `RPATH` mechanisms, but some linkers are not able to fully decode those paths (e.g. due to the presence of things like `$ORIGIN`).

target_link_libraries

Contents

Specify libraries or flags to use when linking a given target and/or its dependents. Usage requirements from linked library targets will be propagated. Usage requirements of a target's dependencies affect compilation of its own sources.

Overview

This command has several signatures as detailed in subsections below. All of them have the general form:

```
target_link_libraries(<target> ... <item>... ...)
```

The named `<target>` must have been created by a command such as `add_executable()` or `add_library()` and must not be an ALIAS target. If policy `CMP0079` is not set to `NEW` then the target must have been created in the current directory. Repeated calls for the same `<target>` append items in the order called.

Each `<item>` may be:

- **A library target name**: The generated link line will have the full path to the linkable library file associated with the target. The buildsystem will have a dependency to re-link `<target>` if the library file changes.

  The named target must be created by `add_library()` within the project or as an IMPORTED library. If it is created within the project an ordering dependency will automatically be added in the build system to make sure the named library target is up-to-date before the `<target>` links.

  If an imported library has the `IMPORTED_NO_SONAME` target property set, CMake may ask the linker to search for the library instead of using the full path (e.g. `/usr/lib/libfoo.so` becomes `-lfoo`).

  The full path to the target's artifact will be quoted/escaped for the shell automatically.

- **A full path to a library file**: The generated link line will normally preserve the full path to the file. The buildsystem will have a dependency to re-link `<target>` if the library file changes.

There are some cases where CMake may ask the linker to search for the library (e.g. `/usr/lib/libfoo.so` becomes `-lfoo`), such as when a shared library is detected to have no `SONAME` field. See policy `CMP0060` for discussion of another case.

If the library file is in a Mac OSX framework, the `Headers` directory of the framework will also be processed as a usage requirement. This has the same effect as passing the framework directory as an include directory.

On Visual Studio Generators for VS 2010 and above, library files ending in `.targets` will be treated as MSBuild targets files and imported into generated project files. This is not supported by other generators.

The full path to the library file will be quoted/escaped for the shell automatically.

- **A plain library name**: The generated link line will ask the linker to search for the library (e.g. `foo` becomes `-lfoo` or `foo.lib`).

  The library name/flag is treated as a command-line string fragment and will be used with no extra quoting or escaping.

- **A link flag**: Item names starting with `-`, but not `-l` or `-framework`, are treated as linker flags. Note that such flags will be treated like any other library link item for purposes of transitive dependencies, so they are generally safe to specify only as private link items that will not propagate to dependents.

  Link flags specified here are inserted into the link command in the same place as the link libraries. This might not be correct, depending on the linker. Use the `LINK_OPTIONS` target property or `target_link_options()` command to add link flags explicitly. The flags will then be placed at the toolchain-defined flag position in the link command.

  The link flag is treated as a command-line string fragment and will be used with no extra quoting or escaping.

- **A generator expression**: A `$<...>` `generator expression` may evaluate to any of the above items or to a ;-list of them. If the `...` contains any `;` characters, e.g. after evaluation of a `${list}` variable, be sure to use an explicitly quoted argument `"$<...>"` so that this command receives it as a single `<item>`.

  Additionally, a generator expression may be used as a fragment of any of the above items, e.g. `foo$<1:_d>`.

  Note that generator expressions will not be used in OLD handling of policy `CMP0003` or policy `CMP0004`.

- A `debug`, `optimized`, or `general` keyword immediately followed by another `<item>`. The item following such a keyword will be used only for the corresponding build configuration. The `debug` keyword corresponds to the `Debug` configuration (or to configurations named in the `DEBUG_CONFIGURATIONS` global property if it is set). The `optimized` keyword corresponds to all other configurations. The `general` keyword corresponds to all configurations, and is purely optional. Higher granularity may be achieved for per-configuration rules by creating and linking to IMPORTED library targets. These keywords are interpreted immediately by this command and therefore have no special meaning when produced by a generator expression.

Items containing `::`, such as `Foo::Bar`, are assumed to be IMPORTED or ALIAS library target names and will cause an error if no such target exists. See policy `CMP0028`.

See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

Libraries for a Target and/or its Dependents

```
target_link_libraries(<target>
                      <PRIVATE|PUBLIC|INTERFACE> <item>...
                     [<PRIVATE|PUBLIC|INTERFACE> <item>...]...)
```

The `PUBLIC`, `PRIVATE` and `INTERFACE` keywords can be used to specify both the link dependencies and the link interface in one command. Libraries and targets following `PUBLIC` are linked to, and are made part of the link interface. Libraries and targets following `PRIVATE` are linked to, but are not made part of the link interface. Libraries following `INTERFACE` are appended to the link interface and are not used for linking `<target>`.

Libraries for both a Target and its Dependents

```
target_link_libraries(<target> <item>...)
```

Library dependencies are transitive by default with this signature. When this target is linked into another target then the libraries linked to this target will appear on the link line for the other target too. This transitive "link interface" is stored in the `INTERFACE_LINK_LIBRARIES` target property and may be overridden by setting the property directly. When `CMP0022` is not set to `NEW`, transitive linking is built in but may be overridden by the `LINK_INTERFACE_LIBRARIES` property. Calls to other signatures of this command may set the property making any libraries linked exclusively by this signature private.

Libraries for a Target and/or its Dependents (Legacy)

```
target_link_libraries(<target>
                      <LINK_PRIVATE|LINK_PUBLIC> <lib>...
                     [<LINK_PRIVATE|LINK_PUBLIC> <lib>...]...)
```

The `LINK_PUBLIC` and `LINK_PRIVATE` modes can be used to specify both the link dependencies and the link interface in one command.

This signature is for compatibility only. Prefer the `PUBLIC` or `PRIVATE` keywords instead.

Libraries and targets following `LINK_PUBLIC` are linked to, and are made part of the `INTERFACE_LINK_LIBRARIES`. If policy `CMP0022` is not `NEW`, they are also made part of the `LINK_INTERFACE_LIBRARIES`. Libraries and targets following `LINK_PRIVATE` are linked to, but are not made part of the `INTERFACE_LINK_LIBRARIES` (or `LINK_INTERFACE_LIBRARIES`).

Libraries for Dependents Only (Legacy)

```
target_link_libraries(<target> LINK_INTERFACE_LIBRARIES <item>...)
```

The `LINK_INTERFACE_LIBRARIES` mode appends the libraries to the `INTERFACE_LINK_LIBRARIES` target property instead of using them for linking. If policy `CMP0022` is not `NEW`, then this mode also appends libraries to the `LINK_INTERFACE_LIBRARIES` and its per-configuration equivalent.

This signature is for compatibility only. Prefer the `INTERFACE` mode instead.

Libraries specified as `debug` are wrapped in a generator expression to correspond to debug builds. If policy `CMP0022` is not `NEW`, the libraries are also appended to the `LINK_INTERFACE_LIBRARIES_DEBUG` property (or to the properties corresponding to configurations listed in the `DEBUG_CONFIGURATIONS` global property if it is set). Libraries specified as `optimized` are appended to the `INTERFACE_LINK_LIBRARIES` property. If policy `CMP0022` is not `NEW`, they are also appended to the `LINK_INTERFACE_LIBRARIES` property. Libraries specified as `general` (or without any keyword) are treated as if specified for both `debug` and `optimized`.

Linking Object Libraries

Object Libraries may be used as the `<target>` (first) argument of `target_link_libraries` to specify dependencies of their sources on other libraries. For example, the code

```
add_library(A SHARED a.c)
target_compile_definitions(A PUBLIC A)

add_library(obj OBJECT obj.c)
target_compile_definitions(obj PUBLIC OBJ)
target_link_libraries(obj PUBLIC A)
```

compiles `obj.c` with `-DA -DOBJ` and establishes usage requirements for `obj` that propagate to its dependents.

Normal libraries and executables may link to Object Libraries to get their objects and usage requirements. Continuing the above example, the code

```
add_library(B SHARED b.c)
target_link_libraries(B PUBLIC obj)
```

compiles `b.c` with `-DA -DOBJ`, creates shared library `B` with object files from `b.c` and `obj.c`, and links `B` to `A`. Furthermore, the code

```
add_executable(main main.c)
target_link_libraries(main B)
```

compiles `main.c` with `-DA -DOBJ` and links executable `main` to `B` and `A`. The object library's usage requirements are propagated transitively through `B`, but its object files are not.

Object Libraries may "link" to other object libraries to get usage requirements, but since they do not have a link step nothing is done with their object files. Continuing from the above example, the code:

```
add_library(obj2 OBJECT obj2.c)
target_link_libraries(obj2 PUBLIC obj)

add_executable(main2 main2.c)
target_link_libraries(main2 obj2)
```

compiles `obj2.c` with `-DA -DOBJ`, creates executable `main2` with object files from `main2.c` and `obj2.c`, and links `main2` to `A`.

In other words, when Object Libraries appear in a target's `INTERFACE_LINK_LIBRARIES` property they will be treated as Interface Libraries, but when they appear in a target's `LINK_LIBRARIES` property their object files will be included in the link too.

Cyclic Dependencies of Static Libraries

The library dependency graph is normally acyclic (a DAG), but in the case of mutually-dependent `STATIC` libraries CMake allows the graph to contain cycles (strongly connected components). When another target links to one of the libraries, CMake repeats the entire connected component. For example, the code

```
add_library(A STATIC a.c)
add_library(B STATIC b.c)
target_link_libraries(A B)
target_link_libraries(B A)
add_executable(main main.c)
target_link_libraries(main A)
```

links `main` to `A B A B`. While one repetition is usually sufficient, pathological object file and symbol arrangements can require more. One may handle such cases by using the `LINK_INTERFACE_MULTIPLICITY` target property or by manually repeating the component in the last `target_link_libraries` call. However, if two archives are really so interdependent they should probably be combined into a single archive, perhaps by using Object Libraries.

Creating Relocatable Packages

Note that it is not advisable to populate the `INTERFACE_LINK_LIBRARIES` of a target with absolute paths to dependencies. That would hard-code into installed packages the library file paths for dependencies **as found on the machine the package was made on**.

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_link_options

Add link options to a target.

```
target_link_options(<target> [BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

Specify link options to use when linking a given target. The named `<target>` must have been created by a command such as `add_executable()` or `add_library()` and must not be an ALIAS target.

If `BEFORE` is specified, the content will be prepended to the property instead of being appended.

This command can be used to add any options, but alternative commands exist to add libraries (`target_link_libraries()` and `link_libraries()`). See documentation of the `directory` and `target` `LINK_OPTIONS` properties.

The `INTERFACE`, `PUBLIC` and `PRIVATE` keywords are required to specify the scope of the following arguments. `PRIVATE` and `PUBLIC` items will populate the `LINK_OPTIONS` property of `<target>`. `PUBLIC` and `INTERFACE` items will populate the `INTERFACE_LINK_OPTIONS` property of `<target>`. (IMPORTED targets only support `INTERFACE` items.) The following arguments specify link options. Repeated calls for the same `<target>` append items in the order called.

Arguments to `target_link_options` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

The final set of compile or link options used for a target is constructed by accumulating options from the current target and the usage requirements of it dependencies. The set of options is de-duplicated to avoid repetition. While beneficial for individual options, the de-duplication step can break up option groups. For example, `-D A -D B` becomes `-D A B`. One may specify a group of options using shell-like quoting along with a `SHELL:` prefix. The `SHELL:` prefix is dropped and the rest of the option string is parsed using the `separate_arguments()` `UNIX_COMMAND` mode. For example, `"SHELL:-D A" "SHELL:-D B"` becomes `-D A -D B`.

To pass options to the linker tool, each compiler driver has is own syntax. The `LINKER:` prefix can be used to specify, in a portable way, options to pass to the linker tool. The `LINKER:` prefix is replaced by the required driver option and the rest of the option string defines linker arguments using `,` as separator. These arguments will be formatted according to the `CMAKE_<LANG>_LINKER_WRAPPER_FLAG` and `CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP` variables.

For example, `"LINKER:-z,defs"` becomes `-Xlinker -z -Xlinker defs` for `Clang` and `-Wl,-z,defs` for `GNU GCC`.

The `LINKER:` prefix can be specified as part of a `SHELL:` prefix expression.

The `LINKER:` prefix supports, as alternate syntax, specification of arguments using `SHELL:` prefix and space as separator. Previous example becomes `"LINKER:SHELL:-z defs"`.

Note

Specifying `SHELL:` prefix elsewhere than at the beginning of the `LINKER:` prefix is not supported.

target_sources

Add sources to a target.

```
target_sources(<target>
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

Specify sources to use when compiling a given target. Relative source file paths are interpreted as being relative to the current source directory (i.e. `CMAKE_CURRENT_SOURCE_DIR`). The named `<target>` must have been created by a command such as `add_executable()` or `add_library()` and must not be an ALIAS target.

The `INTERFACE`, `PUBLIC` and `PRIVATE` keywords are required to specify the scope of the following arguments. `PRIVATE` and `PUBLIC` items will populate the `SOURCES` property of `<target>`. `PUBLIC` and `INTERFACE` items will populate the `INTERFACE_SOURCES` property of `<target>`. (IMPORTED targets only support `INTERFACE` items.) The following arguments specify sources. Repeated calls for the same `<target>` append items in the order called.

Arguments to `target_sources` may use "generator expressions" with the syntax `$<...>`. See the `cmake-generator-expressions(7)` manual for available expressions. See the `cmake-buildsystem(7)` manual for more on defining buildsystem properties.

See also the `CMP0076` policy for older behavior related to the handling of relative source file paths.

try_compile

Contents

Try building some code.

Try Compiling Whole Projects

```
try_compile(RESULT_VAR <bindir> <srcdir>
            <projectName> [<targetName>] [CMAKE_FLAGS <flags>...]
            [OUTPUT_VARIABLE <var>])
```

Try building a project. The success or failure of the `try_compile`, i.e. `TRUE` or `FALSE` respectively, is returned in `RESULT_VAR`.

In this form, `<srcdir>` should contain a complete CMake project with a `CMakeLists.txt` file and all sources. The `<bindir>` and `<srcdir>` will not be deleted after this command is run. Specify `<targetName>` to build a specific target instead of the `all` or `ALL_BUILD` target. See below for the meaning of other options.

Try Compiling Source Files

```
try_compile(RESULT_VAR <bindir> <srcfile|SOURCES srcfile...>
            [CMAKE_FLAGS <flags>...]
            [COMPILE_DEFINITIONS <defs>...]
            [LINK_LIBRARIES <libs>...]
            [OUTPUT_VARIABLE <var>]
            [COPY_FILE <fileName> [COPY_FILE_ERROR <var>]]
            [<LANG>_STANDARD <std>]
            [<LANG>_STANDARD_REQUIRED <bool>]
            [<LANG>_EXTENSIONS <bool>]
            )
```

Try building an executable from one or more source files. The success or failure of the `try_compile`, i.e. `TRUE` or `FALSE` respectively, is returned in `RESULT_VAR`.

In this form the user need only supply one or more source files that include a definition for `main`. CMake will create a `CMakeLists.txt` file to build the source(s) as an executable that looks something like this:

```
add_definitions(<expanded COMPILE_DEFINITIONS from caller>)
include_directories(${INCLUDE_DIRECTORIES})
link_directories(${LINK_DIRECTORIES})
add_executable(cmTryCompileExec <srcfile>...)
target_link_libraries(cmTryCompileExec ${LINK_LIBRARIES})
```

The options are:

- `CMAKE_FLAGS <flags>...`

  Specify flags of the form `-DVAR:TYPE=VALUE` to be passed to the `cmake` command-line used to drive the test build. The above example shows how values for variables `INCLUDE_DIRECTORIES`, `LINK_DIRECTORIES`, and `LINK_LIBRARIES` are used.

- `COMPILE_DEFINITIONS <defs>...`

  Specify `-Ddefinition` arguments to pass to `add_definitions` in the generated test project.

- `COPY_FILE <fileName>`

  Copy the linked executable to the given `<fileName>`.

- `COPY_FILE_ERROR <var>`

  Use after `COPY_FILE` to capture into variable `<var>` any error message encountered while trying to copy the file.

- `LINK_LIBRARIES <libs>...`

  Specify libraries to be linked in the generated project. The list of libraries may refer to system libraries and toImported Targets from the calling project.If this option is specified, any `-DLINK_LIBRARIES=...` value given to the `CMAKE_FLAGS` option will be ignored.

- `OUTPUT_VARIABLE <var>`

  Store the output from the build process the given variable.

- `<LANG>_STANDARD <std>`

  Specify the `C_STANDARD`, `CXX_STANDARD`, or `CUDA_STANDARD` target property of the generated project.

- `<LANG>_STANDARD_REQUIRED <bool>`

  Specify the `C_STANDARD_REQUIRED`, `CXX_STANDARD_REQUIRED`, or `CUDA_STANDARD_REQUIRED` target property of the generated project.

- `<LANG>_EXTENSIONS <bool>`

  Specify the `C_EXTENSIONS`, `CXX_EXTENSIONS`, or `CUDA_EXTENSIONS` target property of the generated project.

In this version all files in `<bindir>/CMakeFiles/CMakeTmp` will be cleaned automatically. For debugging, `--debug-trycompile` can be passed to `cmake` to avoid this clean. However, multiple sequential `try_compile` operations reuse this single output directory. If you use `--debug-trycompile`, you can only debug one `try_compile` call at a time. The recommended procedure is to protect all `try_compile` calls in your project by `if(NOT DEFINED RESULT_VAR)` logic, configure with cmake all the way through once, then delete the cache entry associated with the try_compile call of interest, and then re-run cmake again with `--debug-trycompile`.

Other Behavior Settings

If set, the following variables are passed in to the generated try_compile CMakeLists.txt to initialize compile target properties with default values:

- `CMAKE_ENABLE_EXPORTS`
- `CMAKE_LINK_SEARCH_START_STATIC`
- `CMAKE_LINK_SEARCH_END_STATIC`
- `CMAKE_POSITION_INDEPENDENT_CODE`

If `CMP0056` is set to `NEW`, then `CMAKE_EXE_LINKER_FLAGS` is passed in as well.

The current setting of `CMP0065` is set in the generated project.

Set the `CMAKE_TRY_COMPILE_CONFIGURATION` variable to choose a build configuration.

Set the `CMAKE_TRY_COMPILE_TARGET_TYPE` variable to specify the type of target used for the source file signature.

Set the `CMAKE_TRY_COMPILE_PLATFORM_VARIABLES` variable to specify variables that must be propagated into the test project. This variable is meant for use only in toolchain files.

If `CMP0067` is set to `NEW`, or any of the `<LANG>_STANDARD`, `<LANG>_STANDARD_REQUIRED`, or `<LANG>_EXTENSIONS` options are used, then the language standard variables are honored:

- `CMAKE_C_STANDARD`
- `CMAKE_C_STANDARD_REQUIRED`
- `CMAKE_C_EXTENSIONS`
- `CMAKE_CXX_STANDARD`
- `CMAKE_CXX_STANDARD_REQUIRED`
- `CMAKE_CXX_EXTENSIONS`
- `CMAKE_CUDA_STANDARD`
- `CMAKE_CUDA_STANDARD_REQUIRED`
- `CMAKE_CUDA_EXTENSIONS`

Their values are used to set the corresponding target properties in the generated project (unless overridden by an explicit option).

try_run

Contents

Try compiling and then running some code.

Try Compiling and Running Source Files

```
try_run(RUN_RESULT_VAR COMPILE_RESULT_VAR
        bindir srcfile [CMAKE_FLAGS <flags>...]
        [COMPILE_DEFINITIONS <defs>...]
        [LINK_LIBRARIES <libs>...]
        [COMPILE_OUTPUT_VARIABLE <var>]
        [RUN_OUTPUT_VARIABLE <var>]
        [OUTPUT_VARIABLE <var>]
        [ARGS <args>...])
```

Try compiling a `<srcfile>`. Returns `TRUE` or `FALSE` for success or failure in `COMPILE_RESULT_VAR`. If the compile succeeded, runs the executable and returns its exit code in `RUN_RESULT_VAR`. If the executable was built, but failed to run, then `RUN_RESULT_VAR` will be set to `FAILED_TO_RUN`. See the `try_compile()` command for information on how the test project is constructed to build the source file.

The options are:

- `CMAKE_FLAGS <flags>...`

  Specify flags of the form `-DVAR:TYPE=VALUE` to be passed to the `cmake` command-line used to drive the test build. The example in `try_compile()` shows how values for variables `INCLUDE_DIRECTORIES`, `LINK_DIRECTORIES`, and `LINK_LIBRARIES` are used.

- `COMPILE_DEFINITIONS <defs>...`

  Specify `-Ddefinition` arguments to pass to `add_definitions` in the generated test project.

- `COMPILE_OUTPUT_VARIABLE <var>`

  Report the compile step build output in a given variable.

- `LINK_LIBRARIES <libs>...`

  Specify libraries to be linked in the generated project. The list of libraries may refer to system libraries and toImported Targets from the calling project.If this option is specified, any `-DLINK_LIBRARIES=...` value given to the `CMAKE_FLAGS` option will be ignored.

- `OUTPUT_VARIABLE <var>`

  Report the compile build output and the output from running the executable in the given variable. This option exists for legacy reasons. Prefer `COMPILE_OUTPUT_VARIABLE` and `RUN_OUTPUT_VARIABLE` instead.

- `RUN_OUTPUT_VARIABLE <var>`

  Report the output from running the executable in a given variable.

Other Behavior Settings

Set the `CMAKE_TRY_COMPILE_CONFIGURATION` variable to choose a build configuration.

Behavior when Cross Compiling

When cross compiling, the executable compiled in the first step usually cannot be run on the build host. The `try_run` command checks the `CMAKE_CROSSCOMPILING` variable to detect whether CMake is in cross-compiling mode. If that is the case, it will still try to compile the executable, but it will not try to run the executable unless the `CMAKE_CROSSCOMPILING_EMULATOR` variable is set. Instead it will create cache variables which must be filled by the user or by presetting them in some CMake script file to the values the executable would have produced if it had been run on its actual target platform. These cache entries are:

- `<RUN_RESULT_VAR>`

  Exit code if the executable were to be run on the target platform.

- `<RUN_RESULT_VAR>__TRYRUN_OUTPUT`

  Output from stdout and stderr if the executable were to be run on the target platform. This is created only if the `RUN_OUTPUT_VARIABLE` or `OUTPUT_VARIABLE` option was used.

In order to make cross compiling your project easier, use `try_run` only if really required. If you use `try_run`, use the `RUN_OUTPUT_VARIABLE` or `OUTPUT_VARIABLE` options only if really required. Using them will require that when cross-compiling, the cache variables will have to be set manually to the output of the executable. You can also "guard" the calls to `try_run` with an `if()` block checking the `CMAKE_CROSSCOMPILING` variable and provide an easy-to-preset alternative for this case.

==========

CTest Commands

ctest_build

Perform the CTest Build Step as a Dashboard Client.

```
ctest_build([BUILD <build-dir>] [APPEND]
            [CONFIGURATION <config>]
            [FLAGS <flags>]
            [PROJECT_NAME <project-name>]
            [TARGET <target-name>]
            [NUMBER_ERRORS <num-err-var>]
            [NUMBER_WARNINGS <num-warn-var>]
            [RETURN_VALUE <result-var>]
            [CAPTURE_CMAKE_ERROR <result-var>]
            )
```

Build the project and store results in `Build.xml` for submission with the `ctest_submit()` command.

The `CTEST_BUILD_COMMAND` variable may be set to explicitly specify the build command line. Otherwise the build command line is computed automatically based on the options given.

The options are:

- `BUILD <build-dir>`

  Specify the top-level build directory. If not given, the `CTEST_BINARY_DIRECTORY` variable is used.

- `APPEND`

  Mark `Build.xml` for append to results previously submitted to a dashboard server since the last `ctest_start()` call. Append semantics are defined by the dashboard server in use. This does *not* cause results to be appended to a `.xml` file produced by a previous call to this command.

- `CONFIGURATION <config>`

  Specify the build configuration (e.g. `Debug`). If not specified the `CTEST_BUILD_CONFIGURATION` variable will be checked. Otherwise the `-C <cfg>` option given to the `ctest(1)` command will be used, if any.

- `FLAGS <flags>`

  Pass additional arguments to the underlying build command. If not specified the `CTEST_BUILD_FLAGS` variable will be checked. This can, e.g., be used to trigger a parallel build using the `-j` option of make. See the `ProcessorCount` module for an example.

- `PROJECT_NAME <project-name>`

  Set the name of the project to build. This should correspond to the top-level call to the `project()` command. If not specified the `CTEST_PROJECT_NAME` variable will be checked.

- `TARGET <target-name>`

  Specify the name of a target to build. If not specified the `CTEST_BUILD_TARGET` variable will be checked. Otherwise the default target will be built. This is the "all" target (called `ALL_BUILD` in Visual Studio Generators).

- `NUMBER_ERRORS <num-err-var>`

  Store the number of build errors detected in the given variable.

- `NUMBER_WARNINGS <num-warn-var>`

  Store the number of build warnings detected in the given variable.

- `RETURN_VALUE <result-var>`

  Store the return value of the native build tool in the given variable.

- `CAPTURE_CMAKE_ERROR <result-var>`

  Store in the `<result-var>` variable -1 if there are any errors running the command and prevent ctest from returning non-zero if an error occurs.

- `QUIET`

  Suppress any CTest-specific non-error output that would have been printed to the console otherwise. The summary of warnings / errors, as well as the output from the native build tool is unaffected by this option.

ctest_configure

Perform the CTest Configure Step as a Dashboard Client.

```
ctest_configure([BUILD <build-dir>] [SOURCE <source-dir>] [APPEND]
                [OPTIONS <options>] [RETURN_VALUE <result-var>] [QUIET]
                [CAPTURE_CMAKE_ERROR <result-var>])
```

Configure the project build tree and record results in `Configure.xml` for submission with the `ctest_submit()` command.

The options are:

- `BUILD <build-dir>`

  Specify the top-level build directory. If not given, the `CTEST_BINARY_DIRECTORY` variable is used.

- `SOURCE <source-dir>`

  Specify the source directory. If not given, the `CTEST_SOURCE_DIRECTORY` variable is used.

- `APPEND`

  Mark `Configure.xml` for append to results previously submitted to a dashboard server since the last `ctest_start()` call. Append semantics are defined by the dashboard server in use. This does *not* cause results to be appended to a `.xml` file produced by a previous call to this command.

- `OPTIONS <options>`

  Specify command-line arguments to pass to the configuration tool.

- `RETURN_VALUE <result-var>`

  Store in the `<result-var>` variable the return value of the native configuration tool.

- `CAPTURE_CMAKE_ERROR <result-var>`

  Store in the `<result-var>` variable -1 if there are any errors running the command and prevent ctest from returning non-zero if an error occurs.

- `QUIET`

  Suppress any CTest-specific non-error messages that would have otherwise been printed to the console. Output from the underlying configure command is not affected.

ctest_coverage

Perform the CTest Coverage Step as a Dashboard Client.

```
ctest_coverage([BUILD <build-dir>] [APPEND]
               [LABELS <label>...]
               [RETURN_VALUE <result-var>]
               [CAPTURE_CMAKE_ERROR <result-var]
               [QUIET]
               )
```

Collect coverage tool results and stores them in `Coverage.xml` for submission with the `ctest_submit()` command.

The options are:

- `BUILD <build-dir>`

  Specify the top-level build directory. If not given, the `CTEST_BINARY_DIRECTORY` variable is used.

- `APPEND`

  Mark `Coverage.xml` for append to results previously submitted to a dashboard server since the last `ctest_start()` call. Append semantics are defined by the dashboard server in use. This does *not* cause results to be appended to a `.xml` file produced by a previous call to this command.

- `LABELS`

  Filter the coverage report to include only source files labeled with at least one of the labels specified.

- `RETURN_VALUE <result-var>`

  Store in the `<result-var>` variable `0` if coverage tools ran without error and non-zero otherwise.

- `CAPTURE_CMAKE_ERROR <result-var>`

  Store in the `<result-var>` variable -1 if there are any errors running the command and prevent ctest from returning non-zero if an error occurs.

- `QUIET`

  Suppress any CTest-specific non-error output that would have been printed to the console otherwise. The summary indicating how many lines of code were covered is unaffected by this option.

ctest_empty_binary_directory¶

empties the binary directory

```
ctest_empty_binary_directory( directory )
```

Removes a binary directory. This command will perform some checks prior to deleting the directory in an attempt to avoid malicious or accidental directory deletion.

ctest_memcheck

Perform the CTest MemCheck Step as a Dashboard Client.

```
ctest_memcheck([BUILD <build-dir>] [APPEND]
               [START <start-number>]
               [END <end-number>]
               [STRIDE <stride-number>]
               [EXCLUDE <exclude-regex>]
               [INCLUDE <include-regex>]
               [EXCLUDE_LABEL <label-exclude-regex>]
               [INCLUDE_LABEL <label-include-regex>]
               [EXCLUDE_FIXTURE <regex>]
               [EXCLUDE_FIXTURE_SETUP <regex>]
               [EXCLUDE_FIXTURE_CLEANUP <regex>]
               [PARALLEL_LEVEL <level>]
               [TEST_LOAD <threshold>]
               [SCHEDULE_RANDOM <ON|OFF>]
```

```
                [STOP_TIME <time-of-day>]
                [RETURN_VALUE <result-var>]
                [DEFECT_COUNT <defect-count-var>]
                [QUIET]
                )
```

Run tests with a dynamic analysis tool and store results in `MemCheck.xml` for submission with the `ctest_submit()` command.

Most options are the same as those for the `ctest_test()` command.

The options unique to this command are:

- `DEFECT_COUNT <defect-count-var>`

  Store in the `<defect-count-var>` the number of defects found.

ctest_read_custom_files

read CTestCustom files.

```
ctest_read_custom_files( directory ... )
```

Read all the CTestCustom.ctest or CTestCustom.cmake files from the given directory.

By default, invoking `ctest(1)` without a script will read custom files from the binary directory.

ctest_run_script

runs a ctest -S script

```
ctest_run_script([NEW_PROCESS] script_file_name script_file_name1
            script_file_name2 ... [RETURN_VALUE var])
```

Runs a script or scripts much like if it was run from ctest -S. If no argument is provided then the current script is run using the current settings of the variables. If `NEW_PROCESS` is specified then each script will be run in a separate process.If `RETURN_VALUE` is specified the return value of the last script run will be put into `var`.

ctest_sleep

sleeps for some amount of time

```
ctest_sleep(<seconds>)
```

Sleep for given number of seconds.

```
ctest_sleep(<time1> <duration> <time2>)
```

Sleep for t=(time1 + duration - time2) seconds if t > 0.

ctest_start

Starts the testing for a given model

```
ctest_start(<model> [<source> [<binary>]] [TRACK <track>] [QUIET])

ctest_start([<model> [<source> [<binary>]]] [TRACK <track>] APPEND [QUIET])
```

Starts the testing for a given model. The command should be called after the binary directory is initialized.

The parameters are as follows:

- `<model>`

  Set the dashboard model. Must be one of `Experimental`, `Continuous`, or `Nightly`. This parameter is required unless `APPEND` is specified.

- `<source>`

  Set the source directory. If not specified, the value of `CTEST_SOURCE_DIRECTORY` is used instead.

- `<binary>`

  Set the binary directory. If not specified, the value of `CTEST_BINARY_DIRECTORY` is used instead.

- `TRACK <track>`

  If `TRACK` is used, the submissions will go to the specified track on the CDash server. If no `TRACK` is specified, the name of the model is used by default.

- `APPEND`

  If `APPEND` is used, the existing `TAG` is used rather than creating a new one based on the current time stamp. If you use `APPEND`, you can omit the `<model>` and `TRACK <track>` parameters, because they will be read from the generated `TAG` file. For example: `ctest_start(Experimental TRACK TrackExperimental)` Later, in another `ctest -S` script: `ctest_start(APPEND)` When the second script runs `ctest_start(APPEND)`, it will read the `Experimental` model and `TrackExperimental` track from the `TAG` file generated by the first `ctest_start()` command. Please note that if you call `ctest_start(APPEND)` and specify a different model or track than in the first `ctest_start()` command, a warning will be issued, and the new model and track will be used.

- `QUIET`

  If `QUIET` is used, CTest will suppress any non-error messages that it otherwise would have printed to the console.

The parameters for `ctest_start()` can be issued in any order, with the exception that `<model>`, `<source>`, and `<binary>` have to appear in that order with respect to each other. The following are all valid and equivalent:

```
ctest_start(Experimental path/to/source path/to/binary TRACK SomeTrack QUIET APPEND)

ctest_start(TRACK SomeTrack Experimental QUIET path/to/source APPEND path/to/binary)

ctest_start(APPEND QUIET Experimental path/to/source TRACK SomeTrack path/to/binary)
```

However, for the sake of readability, it is recommended that you order your parameters in the order listed at the top of this page.

If the `CTEST_CHECKOUT_COMMAND` variable (or the `CTEST_CVS_CHECKOUT` variable) is set, its content is treated as command-line. The command is invoked with the current working directory set to the parent of the source directory, even if the source directory already exists. This can be used to create the source tree from a version control repository.

ctest_submit

Perform the CTest Submit Step as a Dashboard Client.

```
ctest_submit([PARTS <part>...] [FILES <file>...]
             [HTTPHEADER <header>]
             [RETRY_COUNT <count>]
             [RETRY_DELAY <delay>]
             [RETURN_VALUE <result-var>]
             [CAPTURE_CMAKE_ERROR <result-var>]
             [QUIET]
             )
```

Submit results to a dashboard server. By default all available parts are submitted.

The options are:

- `PARTS <part>...`

  Specify a subset of parts to submit. Valid part names are: `Start = nothing Update = ctest_update results, in Update.xml Configure = ctest_configure results, in Configure.xml Build = ctest_build results, in Build.xml Test = ctest_test results, in Test.xml Coverage = ctest_coverage results, in Coverage.xml MemCheck = ctest_memcheck results, in DynamicAnalysis.xml Notes = Files listed by CTEST_NOTES_FILES, in Notes.xml ExtraFiles = Files listed by CTEST_EXTRA_SUBMIT_FILES Upload = Files prepared for upload by ctest_upload(), in Upload.xml Submit = nothing`

- `FILES <file>...`

  Specify an explicit list of specific files to be submitted. Each individual file must exist at the time of the call.

- `HTTPHEADER <HTTP-header>`

  Specify HTTP header to be included in the request to CDash during submission. This suboption can be repeated several times.

- `RETRY_COUNT <count>`

  Specify how many times to retry a timed-out submission.

- `RETRY_DELAY <delay>`

  Specify how long (in seconds) to wait after a timed-out submission before attempting to re-submit.

- `RETURN_VALUE <result-var>`

  Store in the `<result-var>` variable `0` for success and non-zero on failure.

- `CAPTURE_CMAKE_ERROR <result-var>`

  Store in the `<result-var>` variable -1 if there are any errors running the command and prevent ctest from returning non-zero if an error occurs.

- `QUIET`

  Suppress all non-error messages that would have otherwise been printed to the console.

Submit to CDash Upload API

```
ctest_submit(CDASH_UPLOAD <file> [CDASH_UPLOAD_TYPE <type>]
             [HTTPHEADER <header>]
             [RETRY_COUNT <count>]
             [RETRY_DELAY <delay>]
             [RETURN_VALUE <result-var>]
             [QUIET])
```

This second signature is used to upload files to CDash via the CDash file upload API. The API first sends a request to upload to CDash along with a content hash of the file. If CDash does not already have the file, then it is uploaded. Along with the file, a CDash type string is specified to tell CDash which handler to use to process the data.

This signature accepts the `HTTPHEADER`, `RETRY_COUNT`, `RETRY_DELAY`, `RETURN_VALUE` and `QUIET` options as described above.

ctest_test

Perform the CTest Test Step as a Dashboard Client.

```
ctest_test([BUILD <build-dir>] [APPEND]
           [START <start-number>]
           [END <end-number>]
           [STRIDE <stride-number>]
           [EXCLUDE <exclude-regex>]
           [INCLUDE <include-regex>]
           [EXCLUDE_LABEL <label-exclude-regex>]
           [INCLUDE_LABEL <label-include-regex>]
           [EXCLUDE_FIXTURE <regex>]
           [EXCLUDE_FIXTURE_SETUP <regex>]
           [EXCLUDE_FIXTURE_CLEANUP <regex>]
           [PARALLEL_LEVEL <level>]
           [TEST_LOAD <threshold>]
           [SCHEDULE_RANDOM <ON|OFF>]
           [STOP_TIME <time-of-day>]
           [RETURN_VALUE <result-var>]
```

```
            [CAPTURE_CMAKE_ERROR <result-var>]
            [QUIET]
            )
```

Run tests in the project build tree and store results in `Test.xml` for submission with the `ctest_submit()` command.

The options are:

- `BUILD <build-dir>`

  Specify the top-level build directory. If not given, the `CTEST_BINARY_DIRECTORY` variable is used.

- `APPEND`

  Mark `Test.xml` for append to results previously submitted to a dashboard server since the last `ctest_start()` call. Append semantics are defined by the dashboard server in use. This does *not* cause results to be appended to a `.xml` file produced by a previous call to this command.

- `START <start-number>`

  Specify the beginning of a range of test numbers.

- `END <end-number>`

  Specify the end of a range of test numbers.

- `STRIDE <stride-number>`

  Specify the stride by which to step across a range of test numbers.

- `EXCLUDE <exclude-regex>`

  Specify a regular expression matching test names to exclude.

- `INCLUDE <include-regex>`

  Specify a regular expression matching test names to include. Tests not matching this expression are excluded.

- `EXCLUDE_LABEL <label-exclude-regex>`

  Specify a regular expression matching test labels to exclude.

- `INCLUDE_LABEL <label-include-regex>`

  Specify a regular expression matching test labels to include. Tests not matching this expression are excluded.

- `EXCLUDE_FIXTURE <regex>`

  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.

- `EXCLUDE_FIXTURE_SETUP <regex>`

  Same as `EXCLUDE_FIXTURE` except only matching setup tests are excluded.

- `EXCLUDE_FIXTURE_CLEANUP <regex>`

  Same as `EXCLUDE_FIXTURE` except only matching cleanup tests are excluded.

- `PARALLEL_LEVEL <level>`

  Specify a positive number representing the number of tests to be run in parallel.

- `TEST_LOAD <threshold>`

  While running tests in parallel, try not to start tests when they may cause the CPU load to pass above a given threshold. If not specified the `CTEST_TEST_LOAD` variable will be checked, and then the `--test-load` command-line argument to `ctest(1)`. See also the `TestLoad` setting in the CTest Test Step.

- `SCHEDULE_RANDOM <ON|OFF>`

  Launch tests in a random order. This may be useful for detecting implicit test dependencies.

- `STOP_TIME <time-of-day>`

  Specify a time of day at which the tests should all stop running.

- `RETURN_VALUE <result-var>`

  Store in the `<result-var>` variable `0` if all tests passed. Store non-zero if anything went wrong.

- `CAPTURE_CMAKE_ERROR <result-var>`

  Store in the `<result-var>` variable -1 if there are any errors running the command and prevent ctest from returning non-zero if an error occurs.

- `QUIET`

  Suppress any CTest-specific non-error messages that would have otherwise been printed to the console. Output from the underlying test command is not affected. Summary info detailing the percentage of passing tests is also unaffected by the `QUIET` option.

See also the `CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE` and `CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE` variables.

ctest_update

Perform the CTest Update Step as a Dashboard Client.

```
ctest_update([SOURCE <source-dir>]
             [RETURN_VALUE <result-var>]
             [CAPTURE_CMAKE_ERROR <result-var>]
             [QUIET])
```

Update the source tree from version control and record results in `Update.xml` for submission with the `ctest_submit()` command.

The options are:

- `SOURCE <source-dir>`

  Specify the source directory. If not given, the `CTEST_SOURCE_DIRECTORY` variable is used.

- `RETURN_VALUE <result-var>`

  Store in the `<result-var>` variable the number of files updated or `-1` on error.

- `CAPTURE_CMAKE_ERROR <result-var>`

Store in the `<result-var>` variable -1 if there are any errors running the command and prevent ctest from returning non-zero if an error occurs.

- `QUIET`

  Tell CTest to suppress most non-error messages that it would have otherwise printed to the console. CTest will still report the new revision of the repository and any conflicting files that were found.

The update always follows the version control branch currently checked out in the source directory. See the CTest Update Step documentation for more information.

ctest_upload

Upload files to a dashboard server as a Dashboard Client.

```
ctest_upload(FILES <file>... [QUIET] [CAPTURE_CMAKE_ERROR <result-var>])
```

The options are:

- `FILES <file>...`

  Specify a list of files to be sent along with the build results to the dashboard server.

- `QUIET`

  Suppress any CTest-specific non-error output that would have been printed to the console otherwise.

- `CAPTURE_CMAKE_ERROR <result-var>`

  Store in the `<result-var>` variable -1 if there are any errors running the command and prevent ctest from returning non-zero if an error occurs.

==========

Deprecated Commands

build_name

Disallowed. See CMake Policy `CMP0036`.

Use `${CMAKE_SYSTEM}` and `${CMAKE_CXX_COMPILER}` instead.

```
build_name(variable)
```

Sets the specified variable to a string representing the platform and compiler settings. These values are now available through the `CMAKE_SYSTEM` and `CMAKE_CXX_COMPILER` variables.

exec_program

Deprecated. Use the `execute_process()` command instead.

Run an executable program during the processing of the CMakeList.txt file.

```
exec_program(Executable [directory in which to run]
             [ARGS <arguments to executable>]
             [OUTPUT_VARIABLE <var>]
             [RETURN_VALUE <var>])
```

The executable is run in the optionally specified directory. The executable can include arguments if it is double quoted, but it is better to use the optional `ARGS` argument to specify arguments to the program. This is because cmake will then be able to escape spaces in the executable path. An optional argument `OUTPUT_VARIABLE` specifies a variable in which to store the output. To capture the return value of the execution, provide a `RETURN_VALUE`. If `OUTPUT_VARIABLE` is specified, then no output will go to the stdout/stderr of the console running cmake.

export_library_dependencies

Disallowed. See CMake Policy `CMP0033`.

Use `install(EXPORT)` or `export()` command.

This command generates an old-style library dependencies file. Projects requiring CMake 2.6 or later should not use the command. Use instead the `install(EXPORT)` command to help export targets from an installation tree and the `export()` command to export targets from a build tree.

The old-style library dependencies file does not take into account per-configuration names of libraries or the `LINK_INTERFACE_LIBRARIES` target property.

```
export_library_dependencies(<file> [APPEND])
```

Create a file named `<file>` that can be included into a CMake listfile with the INCLUDE command. The file will contain a number of SET commands that will set all the variables needed for library dependency information. This should be the last command in the top level CMakeLists.txt file of the project. If the `APPEND` option is specified, the SET commands will be appended to the given file instead of replacing it.

install_files

Deprecated. Use the `install(FILES)` command instead.

This command has been superceded by the `install()` command. It is provided for compatibility with older CMake code. The `FILES` form is directly replaced by the `FILES` form of the `install()` command. The regexp form can be expressed more clearly using the `GLOB` form of the `file()` command.

```
install_files(<dir> extension file file ...)
```

Create rules to install the listed files with the given extension into the given directory. Only files existing in the current source tree or its corresponding location in the binary tree may be listed. If a file specified already has an extension, that extension will be removed first. This is useful for providing lists of source files such as foo.cxx when you want the corresponding foo.h to be installed. A typical extension is '.h'.

```
install_files(<dir> regexp)
```

Any files in the current source directory that match the regular expression will be installed.

```
install_files(<dir> FILES file file ...)
```

Any files listed after the `FILES` keyword will be installed explicitly from the names given. Full paths are allowed in this form.

The directory `<dir>` is relative to the installation prefix, which is stored in the variable `CMAKE_INSTALL_PREFIX`.

install_programs

Deprecated. Use the `install(PROGRAMS)` command instead.

This command has been superceded by the `install()` command. It is provided for compatibility with older CMake code. The `FILES` form is directly replaced by the `PROGRAMS` form of the `install()` command. The regexp form can be expressed more clearly using the `GLOB` form of the `file()` command.

```
install_programs(<dir> file1 file2 [file3 ...])
install_programs(<dir> FILES file1 [file2 ...])
```

Create rules to install the listed programs into the given directory. Use the `FILES` argument to guarantee that the file list version of the command will be used even when there is only one argument.

```
install_programs(<dir> regexp)
```

In the second form any program in the current source directory that matches the regular expression will be installed.

This command is intended to install programs that are not built by cmake, such as shell scripts. See the `TARGETS` form of the `install()` command to create installation rules for targets built by cmake.

The directory `<dir>` is relative to the installation prefix, which is stored in the variable `CMAKE_INSTALL_PREFIX`.

install_targets

Deprecated. Use the `install(TARGETS)` command instead.

This command has been superceded by the `install()` command. It is provided for compatibility with older CMake code.

```
install_targets(<dir> [RUNTIME_DIRECTORY dir] target target)
```

Create rules to install the listed targets into the given directory. The directory `<dir>` is relative to the installation prefix, which is stored in the variable `CMAKE_INSTALL_PREFIX`. If `RUNTIME_DIRECTORY` is specified, then on systems with special runtime files (Windows DLL), the files will be copied to that directory.

load_command

Disallowed. See CMake Policy `CMP0031`.

Load a command into a running CMake.

```
load_command(COMMAND_NAME <loc1> [loc2 ...])
```

The given locations are searched for a library whose name is cmCOMMAND_NAME. If found, it is loaded as a module and the command is added to the set of available CMake commands. Usually, `try_compile()` is used before this command to compile the module. If the command is successfully loaded a variable named

```
CMAKE_LOADED_COMMAND_<COMMAND_NAME>
```

will be set to the full path of the module that was loaded. Otherwise the variable will not be set.

make_directory

Deprecated. Use the `file(MAKE_DIRECTORY)` command instead.

```
make_directory(directory)
```

Creates the specified directory. Full paths should be given. Any parent directories that do not exist will also be created. Use with care.

output_required_files

Disallowed. See CMake Policy `CMP0032`.

Approximate C preprocessor dependency scanning.

This command exists only because ancient CMake versions provided it. CMake handles preprocessor dependency scanning automatically using a more advanced scanner.

```
output_required_files(srcfile outputfile)
```

Outputs a list of all the source files that are required by the specified srcfile. This list is written into outputfile. This is similar to writing out the dependencies for srcfile except that it jumps from .h files into .cxx, .c and .cpp files if possible.

remove

Deprecated. Use the `list(REMOVE_ITEM)` command instead.

```
remove(VAR VALUE VALUE ...)
```

Removes `VALUE` from the variable `VAR`. This is typically used to remove entries from a vector (e.g. semicolon separated list). `VALUE` is expanded.

subdir_depends

Disallowed. See CMake Policy `CMP0029`.

Does nothing.

```
subdir_depends(subdir dep1 dep2 ...)
```

Does not do anything. This command used to help projects order parallel builds correctly. This functionality is now automatic.

subdirs

Deprecated. Use the `add_subdirectory()` command instead.

Add a list of subdirectories to the build.

```
subdirs(dir1 dir2 ...[EXCLUDE_FROM_ALL exclude_dir1 exclude_dir2 ...]
        [PREORDER] )
```

Add a list of subdirectories to the build. The `add_subdirectory()` command should be used instead of `subdirs` although `subdirs` will still work. This will cause any CMakeLists.txt files in the sub directories to be processed by CMake. Any directories after the `PREORDER` flag are traversed first by makefile builds, the `PREORDER` flag has no effect on IDE projects. Any directories after the `EXCLUDE_FROM_ALL` marker will not be included in the top level makefile or project file. This is useful for having CMake create makefiles or projects for a set of examples in a project. You would want CMake to generate makefiles or project files for all the examples at the same time, but you would not want them to show up in the top level project or be built each time make is run from the top.

use_mangled_mesa

Disallowed. See CMake Policy `CMP0030`.

Copy mesa headers for use in combination with system GL.

```
use_mangled_mesa(PATH_TO_MESA OUTPUT_DIRECTORY)
```

The path to mesa includes, should contain gl_mangle.h. The mesa headers are copied to the specified output directory. This allows mangled mesa headers to override other GL headers by being added to the include directory path earlier.

utility_source

Disallowed. See CMake Policy `CMP0034`.

Specify the source tree of a third-party utility.

```
utility_source(cache_entry executable_name
               path_to_source [file1 file2 ...])
```

When a third-party utility's source is included in the distribution, this command specifies its location and name. The cache entry will not be set unless the `path_to_source` and all listed files exist. It is assumed that the source tree of the utility will have been built before it is needed.

When cross compiling CMake will print a warning if a `utility_source()` command is executed, because in many cases it is used to build an executable which is executed later on. This doesn't work when cross compiling, since the executable can run only on their target platform. So in this case the cache entry has to be adjusted manually so it points to an executable which is runnable on the build host.

variable_requires

Disallowed. See CMake Policy `CMP0035`.

Use the `if()` command instead.

Assert satisfaction of an option's required variables.

```
variable_requires(TEST_VARIABLE RESULT_VARIABLE
                  REQUIRED_VARIABLE1
                  REQUIRED_VARIABLE2 ...)
```

The first argument (`TEST_VARIABLE`) is the name of the variable to be tested, if that variable is false nothing else is done. If `TEST_VARIABLE` is true, then the next argument (`RESULT_VARIABLE`) is a variable that is set to true if all the required variables are set. The rest of the arguments are variables that must be true or not set to NOTFOUND to avoid an error. If any are not true, an error is reported.

write_file

Deprecated. Use the `file(WRITE)` command instead.

```
write_file(filename "message to write"... [APPEND])
```

The first argument is the file name, the rest of the arguments are messages to write. If the argument `APPEND` is specified, then the message will be appended.

NOTE 1: `file(WRITE)` and `file(APPEND)` do exactly the same as this one but add some more functionality.

NOTE 2: When using `write_file` the produced file cannot be used as an input to CMake (CONFIGURE_FILE, source file ...) because it will lead to an infinite loop. Use `configure_file()` if you want to generate input files to CMake.