Appendix C – Listfile Commands

Current Commands

The following is an alphabetical listing of the commands that are currently used by CMake. Later in the appendix there is a section on commands that are deprecated but may still be handled by CMake for backwards compatibility.

add_custom_command: Add a custom build rule to the generated build system.

There are two main signatures for add_custom_command 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...]]
                   [IMPLICIT_DEPENDS <lang1> depend1 ...]
                   [WORKING_DIRECTORY dir]
                   [COMMENT comment] [VERBATIM] [APPEND])
```

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. If an output name is a relative path it will be interpreted relative to the build tree directory corresponding to the

current source directory. Note that MAIN_DEPENDENCY is completely optional and is used as a suggestion to Visual Studio about where to hang the custom command. In Makefile terms this creates a new target of the following form:

```
OUTPUT: MAIN_DEPENDENCY DEPENDS
          COMMAND
```

If more than one command is specified they will be executed in order. The optional ARGS argument is for backward compatibility and will be ignored.

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...] ...]
                   [WORKING_DIRECTORY dir]
                   [COMMENT comment] [VERBATIM])
```

This defines a new command that will be associated with building the specified target. When the command will happen is determined by which of the following is specified:

```
PRE_BUILD  - run before all other dependencies
PRE_LINK   - run after other dependencies
POST_BUILD - run after the target has been built
```

Note that the PRE_BUILD option is only supported on Visual Studio 7 or later. For all other generators PRE_BUILD will be treated as PRE_LINK.

If WORKING_DIRECTORY is specified the command will be executed in the directory given. If COMMENT is set, the value will be displayed as a message before the commands are executed at build time. If APPEND is specified the COMMAND and DEPENDS option values are appended 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, WORKING_DIRECTORY, and MAIN_DEPENDENCY options are currently ignored when APPEND is given, but may be used in the future.

If VERBATIM is given then 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.

If the output of the custom command is not actually created as a file on disk it should be marked as SYMBOLIC with SET_SOURCE_FILES_PROPERTIES.

The IMPLICIT_DEPENDS option requests 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, C++ and Fortran language scanners are supported. 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.

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. 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.

The DEPENDS option specifies 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 specifies any target (created by an ADD_* 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.

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 ... ]
                  [WORKING_DIRECTORY dir]
                  [COMMENT comment] [VERBATIM]
                  [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 ADD_CUSTOM_COMMAND to generate a file with dependencies. By default nothing depends on the custom target. Use ADD_DEPENDENCIES to add dependencies to or from other targets. If the ALL option is

specified it indicates 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). The command and arguments are optional and if not specified an empty target will be created. If WORKING_DIRECTORY is set, then the command will be run in that directory. If COMMENT is set, the value will be displayed as a message before the commands are executed at build time. Dependencies listed with the DEPENDS argument may reference files and outputs of custom commands created with add_custom_command() in the same directory (CMakeLists.txt file).

If VERBATIM is given then 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.

The SOURCES option specifies 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 not build rules.

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

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

Adds flags to the compiler command line for sources in the current directory and below. This command can be used to add any flags, but it was originally intended to add preprocessor definitions. Flags beginning in -D or /D that look like preprocessor definitions are automatically added to the COMPILE_DEFINITIONS property for the current directory. Definitions with non-trival values may be left in the set of flags instead of being converted for reasons of backwards compatibility. See documentation of the directory, target, and source file COMPILE_DEFINITIONS properties for details on adding preprocessor definitions to specific scopes and configurations.

add_dependencies: Add a dependency between top-level targets.

```
add_dependencies(target-name depend-target1
                 depend-target2 ...)
```

Make a top-level target depend on other top-level targets. A top-level target is one created by ADD_EXECUTABLE, ADD_LIBRARY, or ADD_CUSTOM_TARGET. Adding dependencies with this command can be used to make sure one target is built before another target. See the DEPENDS option of ADD_CUSTOM_TARGET and ADD_CUSTOM_COMMAND for adding file-level dependencies in custom rules. See the

OBJECT_DEPENDS option in SET_SOURCE_FILES_PROPERTIES 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 ... sourceN)
```

Adds an executable target called <name> to be built from the source files listed in the command invocation. 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.

The add_executable command can also create IMPORTED executable targets using this signature:

```
add_executable(<name> IMPORTED)
```

An IMPORTED executable target references an executable file located outside the project. No rules are generated to build it. The target name has scope in the directory in which it is created and below. 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_<CONFIG>) which specifies the

location of the main executable file on disk. See documentation of the IMPORTED_* properties for more information.

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

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

Adds a library target called <name> to be built from the source files listed in the command invocation. 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 true.

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.

The add_library command can also create IMPORTED library targets using this signature:

```
add_library(<name> <SHARED|STATIC|MODULE|UNKNOWN> IMPORTED)
```

An IMPORTED library target references a library file located outside the project. No rules are generated to build it. The target name has scope in the directory in which it is created and below. 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_". The most important such property is IMPORTED_LOCATION (and its per-configuration

version IMPORTED_LOCATION_<CONFIG>) which specifies the location of the main library file on disk. See documentation of the IMPORTED_* properties for more information.

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 supercede 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 with the specified arguments.

```
add_test(testname Exename arg1 arg2 ...)
```

If the ENABLE_TESTING command has been run, this command adds a test target to the current directory. If ENABLE_TESTING has not been run, this command does nothing. The tests are run by the testing subsystem by executing Exename with the specified arguments. Exename can be either an executable built by this project or an arbitrary executable on the system (like tclsh). The test will be run with the current working directory set to the CMakeList.txt files corresponding directory in the binary tree.

```
add_test(NAME <name> [CONFIGURATIONS [Debug|Release|...]]
         COMMAND <command> [arg1 [arg2 ...]])
```

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. If a CONFIGURATIONS option is given then the test will be executed only when testing under one of the named configurations.

Arguments after COMMAND may use "generator expressions" with the syntax "$<...>". These expressions are evaluted during build system generation and produce information specific to each generated build configuration. Valid expressions are:

```
$<CONFIGURATION>           = configuration name
$<TARGET_FILE:tgt>         = main file (.exe, .so.1.2, .a)
$<TARGET_LINKER_FILE:tgt> = file used to link (.a, .lib, .so)
$<TARGET_SONAME_FILE:tgt> = file with soname (.so.3)
```

where "tgt" is the name of a target. Target file expressions produce a full path, but _DIR and _NAME versions can produce the directory and file name components:

```
$<TARGET_FILE_DIR:tgt>/$<TARGET_FILE_NAME:tgt>
  $<TARGET_LINKER_FILE_DIR:tgt>/$<TARGET_LINKER_FILE_NAME:tgt>
  $<TARGET_SONAME_FILE_DIR:tgt>/$<TARGET_SONAME_FILE_NAME:tgt>
```

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".

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.

break: Break from an enclosing foreach or while loop.

```
break()
```

Breaks from an enclosing foreach loop or while loop

build_command: Get the command line that will build this project.

```
build_command(<variable> <makecommand>)
```

Sets the given <variable> to a string containing the command that will build this project from the root of the build tree using the build tool given by <makecommand>. <makecommand> should be msdev, nmake, make or one of the end user build tools. This is useful for configuring testing systems.

cmake_minimum_required: Set the minimum required version of cmake for a project.

```
cmake_minimum_required(VERSION major[.minor[.patch]]
                       [FATAL_ERROR])
```

If the current version of CMake is lower than that required it will stop processing the project and report an error. When a version higher than 2.4 is specified the command implicitly invokes

```
cmake_policy(VERSION major[.minor[.patch]])
```

which sets the cmake policy version level to the version specified. When version 2.4 or lower is given the command implicitly invokes

```
cmake_policy(VERSION 2.4)
```

which enables compatibility features for CMake 2.4 and lower. 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.

cmake_policy: Manage CMake Policy settings.

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.

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 major.minor[.patch])
```

Specify that the current CMake list file is written for the given version of CMake. All policies introduced in the specified version or earlier will be set to use NEW behavior. All policies introduced after the specified version 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. The policy version specified must be at least 2.4 or the command will report an error. In order to get compatibility features supporting versions earlier than 2.4 see documentation of policy CMP0001.

```
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.

```
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 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.

Functions and macros 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])
```

Copies a file <input> to file <output> and substitutes variable values referenced in the file content. If <input> is a relative path it is evaluated with respect to the current source directory. The <input> must be a file, not a directory. If <output> is a relative path it is evaluated with respect to the current binary directory. If <output> names an existing directory the input file is placed in that directory with its original name.

This command replaces any variables in the input file referenced as ${VAR} or @VAR@ with their values as determined by CMake. If a variable is not defined, it will be replaced with nothing. If COPYONLY is specified, then no variable expansion will take place. If ESCAPE_QUOTES is specified then any substituted quotes will be C-style escaped. The file will be configured with the current values of CMake variables. If @ONLY is specified, only variables of the form @VAR@ will be replaces and ${VAR} will be ignored. This is useful for configuring scripts that use ${VAR}. Any occurrences of #cmakedefine VAR will be replaced with either #define VAR or /* #undef VAR */ depending on the setting of VAR in CMake. Any occurrences of #cmakedefine01 VAR will be replaced with either #define VAR 1 or #define VAR 0 depending on whether VAR evaluates to TRUE or FALSE in CMake

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 variable CMAKE_TESTDRIVER_BEFORE_TESTMAIN 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 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 GLOBAL. TARGET, SOURCE, and TEST chain to DIRECTORY.

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.

else: Starts the else portion of an if block. See the if command.

elseif: Starts the elseif portion of an if block. See the if command.

enable_language: Enable a language (CXX/C/Fortran/etc)

```
enable_language(languageName [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, Fortran. If OPTIONAL is used, use the CMAKE_<languageName>_COMPILER_WORKS variable to check whether the language has been enabled successfully.

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.

endforeach: Ends a list of commands in a foreach block. See the foreach command.

endfunction: Ends a list of commands in a function block. See the function command.

endif: Ends a list of commands in an if block. See the if command.

endmacro: Ends a list of commands in a macro block. See the macro command.

endwhile: Ends a list of commands in a while block. 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>]
                [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])
```

Runs the given sequence of one or more commands 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. If WORKING_DIRECTORY is given the named directory will be set as the current working directory of the child processes. If TIMEOUT is given the child processes will be terminated if they do not finish in the specified number of seconds (fractions are allowed). If RESULT_VARIABLE is given the variable will be set to contain the result of running the processes. This will be an integer return code from the last child or a string describing an error condition. If OUTPUT_VARIABLE or ERROR_VARIABLE are given 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. If INPUT_FILE, OUTPUT_FILE, or ERROR_FILE is given 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 OUTPUT_QUIET or ERROR_QUIET is given then the standard output or standard error results will be quietly ignored. 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.

export: Export targets from the build tree for use by outside projects.

```
export(TARGETS [target1 [target2 [...]]] [NAMESPACE <namespace>]
       [APPEND] 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. If the APPEND option is given the generated code will be appended to the file instead of overwriting it. If a library target is included in the export but a target to which it links is not included the behavior is unspecified.

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.

```
export(PACKAGE <name>)
```

Store the current build directory in the CMake user package registry for package <name>. The find_package command may consider the directory while searching for package <name>. 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 (<name>Config.cmake) that works with the build tree.

file: File manipulation command.

```
file(WRITE filename "message to write"... )
file(APPEND filename "message to write"... )
file(READ filename variable [LIMIT numBytes]
       [OFFSET offset]
       [HEX])
file(STRINGS filename variable [LIMIT_COUNT num]
       [LIMIT_INPUT numBytes] [LIMIT_OUTPUT numBytes]
       [LENGTH_MINIMUM numBytes] [LENGTH_MAXIMUM numBytes]
       [NEWLINE_CONSUME] [REGEX regex]
       [NO_HEX_CONVERSION])
file(GLOB variable [RELATIVE path] [globbing expressions]...)
file(GLOB_RECURSE variable [RELATIVE path]
       [FOLLOW_SYMLINKS] [globbing expressions]...)
file(RENAME <oldname> <newname>)
file(REMOVE [file1 ...])
file(REMOVE_RECURSE [file1 ...])
file(MAKE_DIRECTORY [directory1 directory2 ...])
file(RELATIVE_PATH variable directory file)
file(TO_CMAKE_PATH path result)
file(TO_NATIVE_PATH path result)
file(DOWNLOAD url file [TIMEOUT timeout]
        [STATUS status] [LOG log])
```

WRITE will write a message into a file called 'filename'. It overwrites the file if it already exists, and creates the file if it does not exist.

APPEND will write a message into a file same as WRITE, except it will append it to the end of the file

READ will read the content of a file and store it into the variable. It will start at the given offset and read up to numBytes. If the argument HEX is given, the binary data will be converted to hexadecimal representation and this will be stored in the variable.

STRINGS will parse a list of ASCII strings from a file and store it in a variable. Binary data in the file are ignored. Carriage return (CR) characters are ignored. It works also for Intel Hex and Motorola S-record files, which are automatically converted to binary format when reading them. Disable this using NO_HEX_CONVERSION.

LIMIT_COUNT sets the maximum number of strings to return. LIMIT_INPUT sets the maximum number of bytes to read from the input file. LIMIT_OUTPUT sets the maximum number of bytes to store in the output variable. LENGTH_MINIMUM sets the minimum length of a string to return. Shorter strings are ignored. LENGTH_MAXIMUM sets the maximum length of a string to return. Longer strings are split into strings no longer than the maximum length. NEWLINE_CONSUME allows newlines to be included in strings instead of terminating them.

REGEX specifies a regular expression that a string must match to be returned. Typical usage

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

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

GLOB will generate a list of all 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 for an expression, the results will be returned as a relative path to the given path. 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
```

GLOB_RECURSE will generate a list similar to the regular GLOB, except it 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 cmake policy CMP0009 is not set to NEW. See cmake --help-policy CMP0009 for more information. Examples of recursive globbing include:

```
/dir/*.py  - match all python files in /dir and subdirectories
```

MAKE_DIRECTORY will create the given directories, also if their parent directories don't exist yet

RENAME moves a file or directory within a filesystem, replacing the destination atomically.

REMOVE will remove the given files, also in subdirectories

REMOVE_RECURSE will remove the given files and directories, also non-empty directories

RELATIVE_PATH will determine relative path from directory to the given file.

TO_CMAKE_PATH will convert path into a CMake style path with UNIX /. The input can be a single path or a system path like "$ENV{PATH}". Note the double quotes around the ENV call TO_CMAKE_PATH only takes one argument.

TO_NATIVE_PATH works just like TO_CMAKE_PATH, but will convert from a cmake style path into the native path style \ for windows and / for UNIX.

DOWNLOAD will download the given URL to the given file. If LOG var is specified a log of the download will be put in var. If STATUS var is specified the status of the operation will be put in var. The status is returned in a 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. If TIMEOUT time is specified, the operation will timeout after time seconds, time can be specified as a float.

The file() command also provides COPY and INSTALL signatures:

```
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, PATTERN, REGEX, and EXCLUDE options.

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

find_file: Find the full path to a file.

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

This is the short-hand signature for the command that is sufficient in many cases. It is the same as find_file(<VAR> name1 [PATHS path1 path2 ...])

```
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_CMAKE_ENVIRONMENT_PATH]
          [NO_CMAKE_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. The name of the full path to a file that is searched for is specified by the names listed after the NAMES argument. Additional search locations can be specified after the PATHS argument. If ENV var is found in the HINTS or PATHS section the environment variable var will be read and converted from a system environment variable to a cmake style list of paths. For example ENV PATH would be a way to list the system path variable. The argument after DOC will be used for the documentation

string in the cache. PATH_SUFFIXES specifies additional subdirectories to check below each search path.

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. Search paths specified in cmake-specific cache variables. These are intended to be used on the command line with a -DVAR=value. This can be skipped if NO_CMAKE_PATH is passed.

```
<prefix>/include for each <prefix> in CMAKE_PREFIX_PATH,
   CMAKE_INCLUDE_PATH and CMAKE_FRAMEWORK_PATH
```

2. Search paths specified in cmake-specific environment variables. These are intended to be set in the user's shell configuration. This can be skipped if NO_CMAKE_ENVIRONMENT_PATH is passed.

```
<prefix>/include for each <prefix> in CMAKE_PREFIX_PATH,
   CMAKE_INCLUDE_PATH and CMAKE_FRAMEWORK_PATH
```

3. 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.

4. Search the standard system environment variables. This can be skipped if NO_SYSTEM_ENVIRONMENT_PATH is an argument. Sepcifically PATH and INCLUDE

5. 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 for each <prefix> in CMAKE_SYSTEM_PREFIX_PATH,
CMAKE_SYSTEM_INCLUDE_PATH and CMAKE_SYSTEM_FRAMEWORK_PATH
```

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

On Darwin or systems supporting OS X Frameworks, the cmake variable CMAKE_FIND_FRAMEWORK can be set to empty or one of the following:

```
"FIRST"  - Try to find frameworks before standard
           libraries or headers. This is the default on Darwin.
"LAST"   - Try to find frameworks after standard
           libraries or headers.
"ONLY"   - Only try to find frameworks.
"NEVER"  - Never try to find frameworks.
```

On Darwin or systems supporting OS X Application Bundles, the cmake variable CMAKE_FIND_APPBUNDLE can be set to empty or one of the following:

```
"FIRST"  - Try to find application bundles before standard
           programs. This is the default on Darwin.
"LAST"   - Try to find application bundles after standard
           programs.
"ONLY"   - Only try to find application bundles.
"NEVER"  - Never try to find application bundles.
```

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. By default it is empty. It is 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 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. By using CMAKE_FIND_ROOT_PATH_BOTH the search order will be as described above. If NO_CMAKE_FIND_ROOT_PATH is used then CMAKE_FIND_ROOT_PATH will not be used. If ONLY_CMAKE_FIND_ROOT_PATH is used then only the re-rooted directories will be searched.

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: Find a library.

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

This is the short-hand signature for the command that is sufficient in many cases. It is the same as find_library(<VAR> name1 [PATHS path1 path2 ...])

```
find_library(
             <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_CMAKE_ENVIRONMENT_PATH]
             [NO_CMAKE_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. The name of the library that is searched for is specified by the names listed after the NAMES argument. Additional search locations can be specified after the PATHS argument. If ENV var is found in the HINTS or PATHS section the environment variable var will be read and converted from a system environment variable to a cmake style list of paths. For example ENV PATH would be a way to list the system path variable. The argument after DOC will be used for the documentation string in the cache. PATH_SUFFIXES specifies additional subdirectories to check below each search path.

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. Search paths specified in cmake-specific cache variables. These are intended to be used on the command line with a -DVAR=value. This can be skipped if NO_CMAKE_PATH is passed.

```
<prefix>/lib for each <prefix> in CMAKE_PREFIX_PATH
   CMAKE_LIBRARY_PATH and CMAKE_FRAMEWORK_PATH
```

2. Search paths specified in cmake-specific environment variables. These are intended to be set in the user's shell configuration. This can be skipped if NO_CMAKE_ENVIRONMENT_PATH is passed.

```
<prefix>/lib for each <prefix> in CMAKE_PREFIX_PATH
   CMAKE_LIBRARY_PATH and CMAKE_FRAMEWORK_PATH
```

3. 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.

4. Search the standard system environment variables. This can be skipped if NO_SYSTEM_ENVIRONMENT_PATH is an argument. Sepcifically PATH and LIB.

5. 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 for each <prefix> in CMAKE_SYSTEM_PREFIX_PATH
   CMAKE_SYSTEM_LIBRARY_PATH and CMAKE_SYSTEM_FRAMEWORK_PATH
```

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

On Darwin or systems supporting OS X Frameworks, the cmake variable CMAKE_FIND_FRAMEWORK can be set to empty or one of the following:

```
"FIRST"  - Try to find frameworks before standard
           libraries or headers. This is the default on Darwin.
"LAST"   - Try to find frameworks after standard
           libraries or headers.
"ONLY"   - Only try to find frameworks.
"NEVER"  - Never try to find frameworks.
```

On Darwin or systems supporting OS X Application Bundles, the cmake variable CMAKE_FIND_APPBUNDLE can be set to empty or one of the following:

```
"FIRST"  - Try to find application bundles before standard
           programs. This is the default on Darwin.
"LAST"   - Try to find application bundles after standard
           programs.
"ONLY"   - Only try to find application bundles.
"NEVER"  - Never try to find application bundles.
```

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. By default it is empty. It is 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 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. By using CMAKE_FIND_ROOT_PATH_BOTH the search order will be as described above. If NO_CMAKE_FIND_ROOT_PATH is used then CMAKE_FIND_ROOT_PATH will not be used. If ONLY_CMAKE_FIND_ROOT_PATH is used then only the re-rooted directories will be searched.

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.

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.

find_package: Load settings for an external project.

```
find_package(<package> [version] [EXACT] [QUIET]
             [[REQUIRED|COMPONENTS] [components...]]
             [NO_POLICY_SCOPE])
```

Finds and loads settings from an external project. <package>_FOUND will be set to indicate whether the package was found. When the package is found package-specific information is provided through variables 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 components may be listed after the REQUIRED option or after the COMPONENTS option if no REQUIRED option is given. 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] is given to a recursive invocation inside a find-module, the [version] and EXACT arguments are forwarded automatically from the outer call. Version support is currently provided only on a package-by-package basis (details below).

User code should generally look for packages using the above simple 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 command has two modes by which it searches for packages: "Module" mode and "Config" mode. Module mode is available when the command is invoked with the above reduced signature. CMake searches for a file called "Find<package>.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. Many find-modules provide limited or no support for versioning; check the module documentation. If no module is found the command proceeds to Config mode.

The complete Config mode command signature is:

```
find_package(<package> [version] [EXACT] [QUIET]
             [[REQUIRED|COMPONENTS] [components...]] [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_CMAKE_ENVIRONMENT_PATH]
             [NO_CMAKE_PATH]
             [NO_SYSTEM_ENVIRONMENT_PATH]
             [NO_CMAKE_PACKAGE_REGISTRY]
             [NO_CMAKE_BUILDS_PATH]
             [NO_CMAKE_SYSTEM_PATH]
             [CMAKE_FIND_ROOT_PATH_BOTH |
              ONLY_CMAKE_FIND_ROOT_PATH |
              NO_CMAKE_FIND_ROOT_PATH])
```

The NO_MODULE option may be used to skip Module mode explicitly. It is also implied by use of options not specified in the reduced signature.

Config mode attempts to locate a configuration file provided by the package to be found. A cache entry called <package>_DIR is created to hold the directory containing the file. By default the command searches for a package with the name <package>. If the NAMES option is given the names following it are used instead of <package>. The command searches for a file called "<name>Config.cmake" or "<lower-case-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 <package>_CONFIG.

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 <package>_DIR has been set to a directory not containing a configuration file CMake will ignore it and search from scratch.

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. 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 <package> name
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:

```
<package>_VERSION       = full provided version string
<package>_VERSION_MAJOR = major version if provided, else 0
<package>_VERSION_MINOR = minor version if provided, else 0
<package>_VERSION_PATCH = patch version if provided, else 0
<package>_VERSION_TWEAK = tweak version if provided, else 0
<package>_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. No attempt is made to choose a highest or closest version number.

Config mode provides an elaborate interface and search procedure. Much of the interface is provided for completeness and for use internally by find-modules loaded by Module mode. Most user code should simply call

```
find_package(<package> [major[.minor]] [EXACT] [REQUIRED|QUIET])
```

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

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

On systems supporting OS X 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 (<package> or names given by NAMES). 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 variables CMAKE_FIND_FRAMEWORK and CMAKE_FIND_APPBUNDLE determine the order of preference as specified below.

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 cmake-specific cache variables. These are intended to be used on the command line with a -DVAR=value. This can be skipped if NO_CMAKE_PATH is passed. Specifically CMAKE_PREFIX_PATH CMAKE_FRAMEWORK_PATH and CMAKE_APPBUNDLE_PATH.

2. Search paths specified in cmake-specific environment variables. These are intended to be set in the user's shell configuration. This can be skipped if

NO_CMAKE_ENVIRONMENT_PATH is passed. Specifically CMAKE_PREFIX_PATH CMAKE_FRAMEWORK_PATH and CMAKE_APPBUNDLE_PATH.

3. 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.

4. 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. Specifically: PATH.

5. Search project build trees recently configured in a CMake GUI. This can be skipped if NO_CMAKE_BUILDS_PATH is passed. It is intended for the case when a user is building multiple dependent projects one after another.

6. Search paths stored in the CMake user package registry. This can be skipped if NO_CMAKE_PACKAGE_REGISTRY is passed. Paths are stored in the registry when CMake configures a project that invokes export(PACKAGE <name>). See the export(PACKAGE) command documentation for more details.

7. Search cmake variables defined in the Platform files for the current system. This can be skipped if NO_CMAKE_SYSTEM_PATH is passed. Specifically: CMAKE_SYSTEM_PREFIX_PATH CMAKE_SYSTEM_FRAMEWORK_PATH and CMAKE_SYSTEM_APPBUNDLE_PATH

8. Search paths specified by the PATHS option. These are typically hard-coded guesses.

On Darwin or systems supporting OS X Frameworks, the cmake variable CMAKE_FIND_FRAMEWORK can be set to empty or one of the following:

```
   "FIRST"  - Try to find frameworks before standard
              libraries or headers. This is the default on
Darwin.
   "LAST"   - Try to find frameworks after standard
              libraries or headers.
   "ONLY"   - Only try to find frameworks.
   "NEVER" - Never try to find frameworks.
```

On Darwin or systems supporting OS X Application Bundles, the cmake variable CMAKE_FIND_APPBUNDLE can be set to empty or one of the following:

```
"FIRST"   - Try to find application bundles before standard
            programs. This is the default on Darwin.
"LAST"    - Try to find application bundles after standard
            programs.
"ONLY"    - Only try to find application bundles.
"NEVER"   - Never try to find application bundles.
```

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. By default it is empty. It is 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 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. By using CMAKE_FIND_ROOT_PATH_BOTH the search order will be as described above. If NO_CMAKE_FIND_ROOT_PATH is used then CMAKE_FIND_ROOT_PATH will not be used. If ONLY_CMAKE_FIND_ROOT_PATH is used then only the re-rooted directories will be searched.

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

Once one of the calls succeeds the result variable will be set and stored in the cache so that no call will search again. See the cmake_policy() command documentation for discussion of the NO_POLICY_SCOPE option.

find_path: Find the directory containing a file.

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

This is the short-hand signature for the command that is sufficient in many cases. It is the same as find_path(<VAR> name1 [PATHS path1 path2 ...])

```
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_CMAKE_ENVIRONMENT_PATH]
          [NO_CMAKE_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. The name of the file in a directory that is searched for is specified by the names listed after the NAMES argument. Additional search locations can be specified after the PATHS argument. If ENV var is found in the HINTS or PATHS section the environment variable var will be read and converted from a system environment variable to a cmake style list of paths. For example ENV PATH would be a way to list the system path variable. The argument after DOC will be used for the documentation string in the cache. PATH_SUFFIXES specifies additional subdirectories to check below each search path.

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. Search paths specified in cmake-specific cache variables. These are intended to be used on the command line with a -DVAR=value. This can be skipped if NO_CMAKE_PATH is passed.

```
<prefix>/include for each <prefix> in CMAKE_PREFIX_PATH
CMAKE_INCLUDE_PATH and CMAKE_FRAMEWORK_PATH
```

2. Search paths specified in cmake-specific environment variables. These are intended to be set in the user's shell configuration. This can be skipped if NO_CMAKE_ENVIRONMENT_PATH is passed.

```
<prefix>/include for each <prefix> in CMAKE_PREFIX_PATH
CMAKE_INCLUDE_PATH and CMAKE_FRAMEWORK_PATH
```

3. 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.

4. Search the standard system environment variables. This can be skipped if NO_SYSTEM_ENVIRONMENT_PATH is an argument. Specifically: PATHand INCLUDE

5. 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 for each <prefix> in CMAKE_SYSTEM_PREFIX_PATH
CMAKE_SYSTEM_INCLUDE_PATH and CMAKE_SYSTEM_FRAMEWORK_PATH
```

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

On Darwin or systems supporting OS X Frameworks, the cmake variable CMAKE_FIND_FRAMEWORK can be set to empty or one of the following:

```
"FIRST"  - Try to find frameworks before standard
           libraries or headers. This is the default on Darwin.
"LAST"   - Try to find frameworks after standard
           libraries or headers.
"ONLY"   - Only try to find frameworks.
"NEVER"  - Never try to find frameworks.
```

On Darwin or systems supporting OS X Application Bundles, the cmake variable CMAKE_FIND_APPBUNDLE can be set to empty or one of the following:

```
"FIRST"  - Try to find application bundles before standard
           programs. This is the default on Darwin.
"LAST"   - Try to find application bundles after standard
           programs.
"ONLY"   - Only try to find application bundles.
"NEVER"  - Never try to find application bundles.
```

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. By default it is empty. It is 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 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. By using CMAKE_FIND_ROOT_PATH_BOTH the search order will be as described above. If NO_CMAKE_FIND_ROOT_PATH is used then CMAKE_FIND_ROOT_PATH will not be used. If ONLY_CMAKE_FIND_ROOT_PATH is used then only the re-rooted directories will be searched.

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: Find an executable program.

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

This is the short-hand signature for the command that is sufficient in many cases. It is the same as find_program(<VAR> name1 [PATHS path1 path2 ...])

```
find_program(
             <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_CMAKE_ENVIRONMENT_PATH]
             [NO_CMAKE_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. The name of the program that is searched for is specified by the names listed after the NAMES argument. Additional search locations can be specified after the PATHS argument. If ENV var is found in the HINTS or PATHS section the environment variable var will be read and converted from a system environment variable to a cmake style list of paths. For example ENV PATH would be a way to list the system path variable. The argument after DOC will be used for the documentation string in the cache. PATH_SUFFIXES specifies additional subdirectories to check below each search path.

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. Search paths specified in cmake-specific cache variables. These are intended to be used on the command line with a -DVAR=value. This can be skipped if NO_CMAKE_PATH is passed.

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

2. Search paths specified in cmake-specific environment variables. These are intended to be set in the user's shell configuration. This can be skipped if NO_CMAKE_ENVIRONMENT_PATH is passed.

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

3. 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.

4. Search the standard system environment variables. This can be skipped if NO_SYSTEM_ENVIRONMENT_PATH is an argument. Specifically: PATH.

5. 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 and CMAKE_SYSTEM_APPBUNDLE_PATH
```

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

On Darwin or systems supporting OS X Frameworks, the cmake variable CMAKE_FIND_FRAMEWORK can be set to empty or one of the following:

```
"FIRST"  - Try to find frameworks before standard
           libraries or headers. This is the default on Darwin.
"LAST"   - Try to find frameworks after standard
           libraries or headers.
"ONLY"   - Only try to find frameworks.
"NEVER"  - Never try to find frameworks.
```

On Darwin or systems supporting OS X Application Bundles, the cmake variable CMAKE_FIND_APPBUNDLE can be set to empty or one of the following:

```
"FIRST"  - Try to find application bundles before standard
           programs. This is the default on Darwin.
"LAST"   - Try to find application bundles after standard
           programs.
"ONLY"   - Only try to find application bundles.
"NEVER"  - Never try to find application bundles.
```

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. By default it is empty. It is 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 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. By using CMAKE_FIND_ROOT_PATH_BOTH the search order will be as described above. If NO_CMAKE_FIND_ROOT_PATH is used then CMAKE_FIND_ROOT_PATH will not be used. If ONLY_CMAKE_FIND_ROOT_PATH is used then only the re-rooted directories will be searched.

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.

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.

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".

* When specifying two numbers, the range will have elements from the first number to the second number.

* The third optional number is the increment used to iterate from the first number to the second number.

```
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). 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 variable 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 functions with optional arguments. Additionally ARGV holds the list of all arguments given to the function and ARGN holds the list of argument past the last expected argument.

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

get_cmake_property: Get a property of the CMake instance.

```
get_cmake_property(VAR property)
```

Get a property from the CMake instance. The value of the property is stored in the variable VAR. If the property is not found, CMake will report an error. Some supported properties

include: VARIABLES, CACHE_VARIABLES, COMMANDS, MACROS, and COMPONENTS.

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. If the property is not defined the empty-string is returned. The DIRECTORY argument specifies another directory from which to retrieve the property value. The specified directory must have already been traversed by CMake.

```
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.

get_filename_component: Get a specific component of a full filename.

```
get_filename_component(VarName FileName
                       PATH|ABSOLUTE|NAME|EXT|NAME_WE|REALPATH
                       [CACHE])
```

Set VarName to be the path (PATH), file name (NAME), file extension (EXT), file name without extension (NAME_WE) of FileName, the full path (ABSOLUTE), or the full path with all symlinks resolved (REALPATH). Note that the path is converted to UNIX slashes format and has 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(VarName FileName
                       PROGRAM [PROGRAM_ARGS ArgVar]
                       [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 ArgVar. 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>  |
              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.

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. 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 define_property. 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.

`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 property is not found, VAR will be set to "NOTFOUND". 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

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 property is not found, VAR will be set to "NOTFOUND". Use set_target_properties to set 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.

get_test_property: Get a property of the test.

```
get_test_property(test VAR property)
```

Get a property from the Test. The value of the property is stored in the variable VAR. If the property is not found, CMake will report an error. For a list of standard properties you can type cmake --help-property-list

if: Conditionally execute a group of commands.

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

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 operators such as EXISTS, COMMAND, and DEFINED. Then any EQUAL, LESS, GREATER, STRLESS, STRGREATER, STREQUAL, MATCHES will be evaluated. Then NOT operators and finally AND, OR operators will be evaluated. Possible expressions are:

```
if(variable)
```

True if the variable's value is not empty, 0, N, NO, OFF, FALSE, NOTFOUND, or <variable>-NOTFOUND.

```
if(NOT variable)
```

True if the variable's value is empty, 0, N, NO, OFF, FALSE, NOTFOUND, or <variable>-NOTFOUND.

```
if(variable1 AND variable2)
```

True if both variables would be considered true individually.

```
if(variable1 OR variable2)
```

True if either variable 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 target, built or imported.

```
if(EXISTS file-name)
if(EXISTS directory-name)
```

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(IS_DIRECTORY directory-name)
```

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

```
if(IS_ABSOLUTE path)
```

True if the given path is an absolute path.

```
if(variable MATCHES regex)
if(string MATCHES regex)
```

True if the given string or variable's value matches the given regular expression.

```
if(variable LESS number)
if(string LESS number)
if(variable GREATER number)
if(string GREATER number)
if(variable EQUAL number)
if(string EQUAL number)
```

True if the given string or variable's value is a valid number and the inequality or equality is true.

```
if(variable STRLESS string)
if(string STRLESS string)
if(variable STRGREATER string)
if(string STRGREATER string)
```

```
if(variable STREQUAL string)
if(string STREQUAL string)
```

True if the given string or variable's value is lexicographically less (or greater, or equal) than the string or variable on the right.

```
if(version1 VERSION_LESS version2)
if(version1 VERSION_EQUAL version2)
if(version1 VERSION_GREATER version2)
```

Component-wise integer version number comparison (version format is major[.minor[.patch[.tweak]]]).

```
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.

```
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 statement was written fairly early in CMake's history and it has some convenience features that are worth covering. The if statement reduces operations until there is a single remaining value, at that point if the case insensitive value is: ON, 1, YES, TRUE, Y it returns true, if it is OFF, 0, NO, FALSE, N, NOTFOUND, *-NOTFOUND, IGNORE it will return false.

This is fairly reasonable. The convenience feature that sometimes throws new authors is how CMake handles values that do not match the true or false list. Those values are treated as variables and are dereferenced even though they do not have the required ${} syntax. This means that if you write

```
if (boobah)
```

CMake will treat it as if you wrote

```
if (${boobah})
```

likewise if you write

```
if (fubar AND sol)
```

CMake will conveniently treat it as

```
if ("${fubar}" AND "${sol}")
```

The later is really the correct way to write it, but the former will work as well. Only some operations in the if statement have this special handling of arguments. The specific details follow:

1) 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.

2) If the left hand argument to MATCHES is missing it returns false without error

3) Both left and right hand arguments to LESS 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.

4) Both left and right hand arguments to STRLESS STREQUAL STRGREATER are independently tested to see if they are defined variables, if so their defined values are used otherwise the original value is used.

5) Both left and right hand argumemnts to VERSION_LESS VERSION_EQUAL VERSION_GREATER are independently tested to see if they are defined variables, if so their defined values are used otherwise the original value is used.

6) 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.

7) The left and right hand arguments to 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.

include: Read CMake listfile code from the given file.

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

Reads CMake listfile code from the given file. Commands in the file are processed immediately as if they were written in place of the include command. 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 in the CMAKE_MODULE_PATH.

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

include_directories: Add include directories to the build.

```
include_directories([AFTER|BEFORE] [SYSTEM] dir1 dir2 ...)
```

Add the given directories to those searched by the compiler for include files. By default the directories are appended onto the current list of directories. This default behavior can be changed by setting CMAKE_include_directories_BEFORE to ON. By using BEFORE or AFTER you can select between appending and prepending, independent from the default. If the SYSTEM option is given the compiler will be told that the directories are meant as system include directories on some platforms.

include_external_msproject: Include an external Microsoft project file in a workspace.

```
include_external_msproject(projectname location
                           dep1 dep2 ...)
```

Includes an external Microsoft project in the generated workspace file. Currently does nothing on UNIX. This will create a target named INCLUDE_EXTERNAL_MSPROJECT_[projectname]. This can be used in the add_dependencies command to make things depend on the external project.

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.

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 properties for files and targets. Properties common to multiple signatures are covered here but they are valid only for signatures that specify them.

DESTINATION arguments 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 CMAKE_INSTALL_PREFIX.

PERMISSIONS arguments 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.

The CONFIGURATIONS argument specifies a list of build configurations for which the install rule applies (Debug, Release, etc.).

The COMPONENT argument specifies 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.

The RENAME argument specifies 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.

The OPTIONAL argument specifies that it is not an error if the file to be installed does not exist.

The TARGETS signature:

```
install(TARGETS targets... [EXPORT <export-name>]
        [[ARCHIVE|LIBRARY|RUNTIME|FRAMEWORK|BUNDLE|
          PRIVATE_HEADER|PUBLIC_HEADER|RESOURCE]
         [DESTINATION <dir>]
         [PERMISSIONS permissions...]
         [CONFIGURATIONS [Debug|Release|...]]
         [COMPONENT <component>]
         [OPTIONAL] [NAMELINK_ONLY|NAMELINK_SKIP]
        ] [...])
```

The TARGETS form specifies rules for installing targets from a project. There are five kinds of target files that may be installed: ARCHIVE, LIBRARY, RUNTIME, FRAMEWORK, and BUNDLE. Executables are treated as RUNTIME targets, except that those marked with the MACOSX_BUNDLE property are treated as BUNDLE targets on OS X. Static libraries are always treated as ARCHIVE targets. Module libraries are always treated as LIBRARY targets. For non-DLL platforms shared libraries are treated as LIBRARY targets, except that those marked with the FRAMEWORK property are treated as FRAMEWORK targets on OS X. For DLL platforms the DLL part of a shared library is treated as a RUNTIME target and the corresponding import library is treated as an ARCHIVE target. All Windows-based systems including Cygwin are DLL platforms. The ARCHIVE, LIBRARY, RUNTIME, and FRAMEWORK arguments change the type of target to which the subsequent properties apply. 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).

The PRIVATE_HEADER, PUBLIC_HEADER, and RESOURCE arguments cause subsequent properties to be applied to installing a FRAMEWORK shared library target's associated files on non-Apple platforms. Rules defined by these arguments are ignored on Apple platforms because the associated files are installed into the appropriate locations inside the framework folder. See documentation of the PRIVATE_HEADER, PUBLIC_HEADER, and RESOURCE target properties for details.

Either NAMELINK_ONLY or NAMELINK_SKIP may be specified as a LIBRARY option. 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_ONLY option causes installation of only the namelink when a library target is installed. The NAMELINK_SKIP option causes installation of library files other than the namelink when a library target is installed. When neither option is given both portions are installed. On platforms where versioned shared libraries do not have namelinks or when a library is not

versioned the NAMELINK_SKIP option installs the library and the NAMELINK_ONLY option installs nothing. See the VERSION and SOVERSION target properties for details on creating versioned shared libraries.

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. On non-DLL platforms mySharedLib will be installed to <prefix>/lib and /some/full/path.

The EXPORT option associates the installed target files with an export called <export-name>. It must appear before any RUNTIME, LIBRARY, or ARCHIVE options. See documentation of the install(EXPORT ...) signature below for details.

Installing a target with EXCLUDE_FROM_ALL set to true has undefined behavior.

The FILES signature:

```
install(FILES files... DESTINATION <dir>
          [PERMISSIONS permissions...]
          [CONFIGURATIONS [Debug|Release|...]]
          [COMPONENT <component>]
          [RENAME <name>] [OPTIONAL])
```

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 signature:

```
install(PROGRAMS files... DESTINATION <dir>
        [PERMISSIONS permissions...]
        [CONFIGURATIONS [Debug|Release|...]]
        [COMPONENT <component>]
        [RENAME <name>] [OPTIONAL])
```

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 DIRECTORY signature:

```
install(DIRECTORY dirs... DESTINATION <dir>
        [FILE_PERMISSIONS permissions...]
        [DIRECTORY_PERMISSIONS permissions...]
        [USE_SOURCE_PERMISSIONS] [OPTIONAL]
        [CONFIGURATIONS [Debug|Release|...]]
        [COMPONENT <component>]
        [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.

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 SCRIPT and CODE signatures:

```
install([[SCRIPT <file>] [CODE <code>]] [...])
```

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.

The EXPORT signature:

```
install(EXPORT <export-name> DESTINATION <dir>
        [NAMESPACE <namespace>] [FILE <name>.cmake]
        [PERMISSIONS permissions...]
        [CONFIGURATIONS [Debug|Release|...]]
        [COMPONENT <component>])
```

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. If a COMPONENT option is specified that does not match that given to the targets associated with <export-name> the behavior is undefined. If a library target is included in the export but a target to which it links is not included the behavior is unspecified.

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)
```

will install the executable myexe to <prefix>/bin and code to import it in the file "<prefix>/lib/myproj/myproj.cmake". 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 target properties PRE_INSTALL_SCRIPT and POST_INSTALL_SCRIPT. 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.

link_directories: Specify directories in which the linker will look for libraries.

```
link_directories(directory1 directory2 ...)
```

Specify the paths in which the linker should search for libraries. The command will apply only to targets created after it is called. For historical reasons, relative paths given to this command are passed to the linker unchanged (unlike many CMake commands which interpret them relative to the current source directory).

list: List operations.

```
list(LENGTH <list> <output variable>)
list(GET <list> <element index> [<element index> ...]
     <output variable>)
list(APPEND <list> <element> [<element> ...])
list(FIND <list> <value> <output variable>)
list(INSERT <list> <element_index> <element> [<element> ...])
list(REMOVE_ITEM <list> <value> [<value> ...])
list(REMOVE_AT <list> <index> [<index> ...])
list(REMOVE_DUPLICATES <list>)
list(REVERSE <list>)
list(SORT <list>)
```

LENGTH will return a given list's length.

GET will return list of elements specified by indices from the list.

APPEND will append elements to the list.

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

INSERT will insert elements to the list to the specified location.

REMOVE_AT and REMOVE_ITEM will remove items from the list. The difference is that REMOVE_ITEM will remove the given items, while REMOVE_AT will remove the items at the given indices.

REMOVE_DUPLICATES will remove duplicated items in the list.

REVERSE reverses the contents of the list in-place.

SORT sorts the list in-place alphabetically.

NOTES: 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.

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.

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.

load_command: 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.

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 argument past the last expected argument. 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. If you want true CMake variables you should look at the function command.

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

mark_as_advanced: Mark cmake cached variables as advanced.

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

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. This command does nothing in script mode.

math: Mathematical expressions.

```
math(EXPR <output variable> <math expression>)
```

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

message: Display a message to the user.

```
message([STATUS|WARNING|AUTHOR_WARNING|FATAL_ERROR|SEND_ERROR]
        "message to display" ...)
```

The optional 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 but skip generation
FATAL_ERROR     = CMake Error, stop all processing
```

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.

output_required_files: Output a list of required source files for a specified source file.

```
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.

project: Set a name for the entire project.

```
project(<projectname> [languageName1 languageName2 ... ] )
```

Sets the name of the project. Additionally this sets the variables <projectName>_BINARY_DIR and <projectName>_SOURCE_DIR to the respective values.

Optionally you can specify which languages your project supports. Example languages are CXX (i.e. C++), C, Fortran, etc. By default C and CXX are enabled. E.g. if you do not have a C++ compiler, you can disable the check for it by explicitely listing the languages you want to support, e.g. C. By using the special language "NONE" all checks for any language can be disabled.

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.

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> <UNIX|WINDOWS>_COMMAND "<args>")
```

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

```
separate_arguments(VARIABLE)
```

Convert the value of VARIABLE to a semi-colon separated list. All spaces are replaced with ';'. This helps with generating command lines.

set: Set a CMAKE variable to a given value.

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

Within CMake sets <variable> to the value <value>. <value> is expanded before <variable> is set to it. If CACHE is present and <variable> is not yet in the cache, then <variable> is put in the cache. If it is already in the cache, <variable> is assigned the value stored in the cache. If CACHE is present, also <type> and <docstring> are required. <type> is used by the CMake GUI to choose a widget with which the user sets a value. The value for <type> may be one of

```
FILEPATH = File chooser dialog.
PATH     = Directory chooser dialog.
STRING   = Arbitrary string.
```

```
BOOL     = Boolean ON/OFF checkbox.
INTERNAL = No GUI entry (used for persistent variables).
```

If <type> is INTERNAL, then the <value> is always written into the cache, replacing any values existing in the cache. If it is not a cache variable, then this always writes into the current Makefile. The FORCE option will overwrite the cache value removing any changes by the user.

If PARENT_SCOPE is present, 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).

If <value> is not specified then the variable is removed instead of set. See also: the unset() command.

```
set(<variable> <value1> ... <valueN>)
```

In this case <variable> is set to a semicolon separated list of values. <variable> can be an environment variable such as:

```
set( ENV{PATH} /home/martink )
```

in which case the environment variable will be set.

set_directory_properties: Set a property of the directory.

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

Set a property for the current directory and subdirectories. If the property is not found, CMake will report an error. The properties include: INCLUDE_DIRECTORIES, LINK_DIRECTORIES, INCLUDE_REGULAR_EXPRESSION, and ADDITIONAL_MAKE_CLEAN_FILES. ADDITIONAL_MAKE_CLEAN_FILES is a list of files that will be cleaned as a part of "make clean" stage.

set_property: Set a named property in a given scope.

```
set_property(<GLOBAL                                |
             DIRECTORY [dir]                        |
             TARGET    [target1 [target2 ...]]      |
```

```
    SOURCE     [src1 [src2 ...]]        |
    TEST       [test1 [test2 ...]]      |
    CACHE      [entry1 [entry2 ...]]>
  [APPEND]
  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.

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.

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 on a file. The syntax for the command is to list all the files you want to change, and then provide the values you want to set next. You can make up your own properties as well. The following are used by CMake. The ABSTRACT flag (boolean) is used by some class wrapping commands. If WRAP_EXCLUDE (boolean) is true then many wrapping commands will ignore this file. If GENERATED (boolean) is true then it is not an error if this source file does not exist when it is added to a target. Obviously, it must be created (presumably by a custom command) before the target is built. If the HEADER_FILE_ONLY (boolean) property is true then the file is not compiled. This is useful if you want to add extra non build files to an IDE. OBJECT_DEPENDS (string) adds dependencies to the object file.

COMPILE_FLAGS (string) is passed to the compiler as additional command line arguments when the source file is compiled. LANGUAGE (string) CXX|C will change the default compiler used to compile the source file. The languages used need to be enabled in the PROJECT command. If SYMBOLIC (boolean) is set to true the build system will be informed that the source file is not actually created on disk but instead used as a symbolic name for a build rule.

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 a target. The syntax for the command is to list all the files 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_TARGET_PROPERTY command. See Appendix E - Properties for a full list of target properties.

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 property is not found, CMake will report an error. See Appendix E - Properties for a list of test properties.

site_name: Set the given variable to the name of the computer.

```
site_name(variable)
```

source_group: Define a grouping for sources in the Makefile.

```
source_group(name [REGULAR_EXPRESSION regex]
                  [FILES src1 src2 ...])
```

Defines a group into which sources will be placed in project files. This is mainly used to setup file tabs in Visual Studio. Any file whose name is listed or matches the regular expression will be placed in this group. If a file matches multiple groups, the LAST group that explicitly lists the file 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 may contain backslashes to specify subgroups:

```
source_group(outer\\inner ...)
```

For backwards compatibility, this command is also supports the format:

```
source_group(name regex)
```

`string`: String operations.

```
string(REGEX MATCH <regular_expression>
       <output variable> <input> [<input>...])
string(REGEX MATCHALL <regular_expression>
       <output variable> <input> [<input>...])
string(REGEX REPLACE <regular_expression>
       <replace_expression> <output variable>
       <input> [<input>...])
string(REPLACE <match_string>
       <replace_string> <output variable>
       <input> [<input>...])
string(COMPARE EQUAL <string1> <string2> <output variable>)
string(COMPARE NOTEQUAL <string1> <string2> <output variable>)
string(COMPARE LESS <string1> <string2> <output variable>)
string(COMPARE GREATER <string1> <string2> <output variable>)
string(ASCII <number> [<number> ...] <output variable>)
string(CONFIGURE <string1> <output variable>
       [@ONLY] [ESCAPE_QUOTES])
string(TOUPPER <string1> <output variable>)
string(TOLOWER <string1> <output variable>)
string(LENGTH <string> <output variable>)
string(SUBSTRING <string> <begin> <length> <output variable>)
string(STRIP <string> <output variable>)
string(RANDOM [LENGTH <length>] [ALPHABET <alphabet>]
       <output variable>)
```

REGEX MATCH will match the regular expression once and store the match in the output variable.

REGEX MATCHALL will match the regular expression as many times as possible and store the matches in the output variable as a list.

REGEX REPLACE will match the regular expression as many times as possible and substitute the replacement expression for the match in the output. 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.

REPLACE will replace all occurrences of match_string in the input with replace_string and store the result in the output.

COMPARE EQUAL/NOTEQUAL/LESS/GREATER will compare the strings and store true or false in the output variable.

ASCII will convert all numbers into corresponding ASCII characters.

CONFIGURE will transform a string like CONFIGURE_FILE transforms a file.

TOUPPER/TOLOWER will convert string to upper/lower characters.

LENGTH will return a given string's length.

SUBSTRING will return a substring of a given string.

STRIP will return a substring of a given string with leading and trailing spaces removed.

RANDOM will 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.

The following characters have special meaning in regular expressions:

```
^          Matches at beginning of a line
$          Matches at end of a line
.          Matches any single character
[ ]        Matches any character(s) inside the brackets
[^ ]       Matches any character(s) not inside the brackets
 -         Matches characters in range on either side of a dash
*          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_(0..9).
```

target_link_libraries: Link a target to given libraries.

```
target_link_libraries(<target> [item1 [item2 [...]]]
                      [[debug|optimized|general] <item>] ...)
```

Specify libraries or flags to use when linking a given target. If a library name matches that of another target in the project a dependency will automatically be added in the build system to make sure the library being linked is up-to-date before the target links. Item names starting with '-', but not '-l' or '-framework', are treated as linker flags.

A "debug", "optimized", or "general" keyword indicates that the library immediately following it is to 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 (assumed if omitted). Higher granularity may be achieved for per-configuration rules by creating and linking to IMPORTED library targets. See the IMPORTED mode of the add_library command for more information.

Library dependencies are transitive by default. 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. See the LINK_INTERFACE_LIBRARIES target property to override the set of transitive link dependencies for a target.

```
target_link_libraries(<target> LINK_INTERFACE_LIBRARIES
                      [[debug|optimized|general] <lib>] ...)
```

The LINK_INTERFACE_LIBRARIES mode appends the libraries to the LINK_INTERFACE_LIBRARIES and its per-configuration equivalent target properties instead of using them for linking. Libraries specified as "debug" are appended to the 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 the LINK_INTERFACE_LIBRARIES property. Libraries specified as "general" (or without any keyword) are treated as if specified for both "debug" and "optimized".

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 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.)

try_compile: Try compiling some code.

```
try_compile(RESULT_VAR bindir srcdir
            projectName <targetname> [CMAKE_FLAGS <Flags>]
            [OUTPUT_VARIABLE var])
```

Try compiling a program. 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. If <target name> is specified then build just that target otherwise the all or ALL_BUILD target is built.

```
try_compile(RESULT_VAR bindir srcfile
            [CMAKE_FLAGS <Flags>]
            [COMPILE_DEFINITIONS <flags> ...]
            [OUTPUT_VARIABLE var]
            [COPY_FILE <filename> )
```

Try compiling a srcfile. In this case, the user need only supply a source file. CMake will create the appropriate CMakeLists.txt file to build the source. If COPY_FILE is used, the compiled file will be copied to the given file.

In this version all files in bindir/CMakeFiles/CMakeTmp, will be cleaned automatically, for debugging a --debug-trycompile can be passed to cmake to avoid the clean. Some extra flags that can be included are, INCLUDE_DIRECTORIES, LINK_DIRECTORIES, and LINK_LIBRARIES. COMPILE_DEFINITIONS are -Ddefinition that will be passed to the compile line. try_compile creates a CMakeList.txt file on the fly that looks like this:

```
add_definitions(<expanded COMPILE_DEFINITIONS from cmake>)
include_directories(${INCLUDE_DIRECTORIES})
```

```
link_directories(${LINK_DIRECTORIES})
add_executable(cmTryCompileExec sources)
target_link_libraries(cmTryCompileExec ${LINK_LIBRARIES})
```

In both versions of the command, if OUTPUT_VARIABLE is specified, then the output from the build process is stored in the given variable. Return the success or failure in RESULT_VAR. CMAKE_FLAGS can be used to pass -DVAR:TYPE=VALUE flags to the cmake that is run during the build.

try_run: Try compiling and then running some code.

```
try_run(RUN_RESULT_VAR COMPILE_RESULT_VAR
        bindir srcfile [CMAKE_FLAGS <Flags>]
        [COMPILE_DEFINITIONS <flags>]
        [COMPILE_OUTPUT_VARIABLE comp]
        [RUN_OUTPUT_VARIABLE run]
        [OUTPUT_VARIABLE var]
        [ARGS <arg1> <arg2>...])
```

Try compiling a srcfile. Return TRUE or FALSE for success or failure in COMPILE_RESULT_VAR. Then if the compile succeeded, run the executable and return 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. COMPILE_OUTPUT_VARIABLE specifies the variable where the output from the compile step goes. RUN_OUTPUT_VARIABLE specifies the variable where the output from the running executable goes.

For compatibility reasons OUTPUT_VARIABLE is still supported, which gives you the output from the compile and run step combined.

Cross compiling issues

When cross compiling, the executable compiled in the first step usually cannot be run on the build host. try_run() checks the CMAKE_CROSSCOMPILING variable to detect whether CMake is in crosscompiling mode. If that's the case, it will still try to compile the executable, but it will not try to run the executable. 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 would have been run on its actual target platform. These variables are RUN_RESULT_VAR (explanation see above) and if RUN_OUTPUT_VARIABLE (or OUTPUT_VARIABLE) was used, an additional cache variable RUN_RESULT_VAR__COMPILE_RESULT_VAR__TRYRUN_OUTPUT.This is intended to hold stdout and stderr from the executable.

In order to make cross compiling your project easier, use try_run only if really required. If you use try_run, use RUN_OUTPUT_VARIABLE (or OUTPUT_VARIABLE) only if really required. Using them will require that when crosscompiling, 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 if(CMAKE_CROSSCOMPILING) and provide an easy-to-preset alternative for this case.

unset: Unset a variable, cache variable, or environment variable.

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

Removes the specified variable causing it to become undefined. If CACHE is present then the variable is removed from the cache instead of the current scope. <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(<variable> <access> <value> <current list file> <stack>)

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.

Compatibility Commands

This is the documentation for now obsolete listfile commands from previous CMake versions, which are still supported for compatibility reasons. You should instead use the newer, preferred commands.

build_name: Deprecated. 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: Deprecated. 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.

link_libraries: Deprecated. Use the target_link_libraries() command instead.

Link libraries to all targets added later.

```
link_libraries(library1 <debug | optimized> library2 ...)
```

Specify a list of libraries to be linked into any following targets (typically added with the add_executable or add_library calls). This command is passed down to all subdirectories. The debug and optimized strings may be used to indicate that the next library listed is to be used only for that specific type of build.

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.

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: Deprecated. 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 ...]
        [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: 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: 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: Deprecated. 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.