Master Index CMake 2.8.9

Name

cmake - Cross-Platform Makefile Generator.

Usage

```
cmake [options] <path-to-source>
cmake [options] <path-to-existing-build>
```

Description

The "cmake" executable is the CMake command-line interface. It may be used to configure projects in scripts. Project configuration settings may be specified on the command line with the -D option. The -i option will cause cmake to interactively prompt for such settings.

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

Options

- -C <initial-cache>
- -D <var>:<type>=<value>
- -U <globbing_expr>
- -G <generator-name>
- -Wno-dev
- -Wdev
- -E
- -i
- -L[A][H]
- --build <dir>
- -N

- -P <file>
- --find-package
- --graphviz=[file]
- --system-information [file]
- --debug-trycompile
- --debug-output
- --trace
- --warn-uninitialized
- --warn-unused-vars
- --no-warn-unused-cli
- --check-system-vars
- --help-command cmd [file]
- --help-command-list [file]
- --help-commands [file]
- --help-compatcommands [file]
- --help-module module [file]
- --help-module-list [file]
- --help-modules [file]
- --help-custom-modules [file]
- --help-policy cmp [file]
- --help-policies [file]
- --help-property prop [file]
- --help-property-list [file]
- --help-properties [file]
- --help-variable var [file]
- --help-variable-list [file]
- --help-variables [file]
- --copyright [file]
- --help,-help,-usage,-h,-H,/?

- --help-full [file]
- --help-html [file]
- --help-man [file]
- --version,-version,/V [file]

- -C <initial-cache>: Pre-load a script to populate the cache.

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

- -D <var>:<type>=<value>: Create a cmake cache entry.

  When cmake is first run in an empty build tree, it creates a CMakeCache.txt file and populates it with customizable settings for the project. This option may be used to specify a setting that takes priority over the project's default value. The option may be repeated for as many cache entries as desired.

- -U <globbing_expr>: Remove matching entries from CMake cache.

  This option may be used to remove one or more variables from the CMakeCache.txt file, globbing expressions using * and ? are supported. The option may be repeated for as many cache entries as desired.

  Use with care, you can make your CMakeCache.txt non-working.

- -G <generator-name>: Specify a makefile generator.

CMake may support multiple native build systems on certain platforms. A makefile generator is responsible for generating a particular build system. Possible generator names are specified in the Generators section.

- `-Wno-dev`: Suppress developer warnings.

  Suppress warnings that are meant for the author of the CMakeLists.txt files.

- `-Wdev`: Enable developer warnings.

  Enable warnings that are meant for the author of the CMakeLists.txt files.

- `-E`: CMake command mode.

  For true platform independence, CMake provides a list of commands that can be used on all systems. Run with -E help for the usage information. Commands available are: chdir, compare_files, copy, copy_directory, copy_if_different, echo, echo_append, environment, make_directory, md5sum, remove, remove_directory, rename, tar, time, touch, touch_nocreate. In addition, some platform specific commands are available. On Windows: comspec, delete_regv, write_regv. On UNIX: create_symlink.

- `-i`: Run in wizard mode.

  Wizard mode runs cmake interactively without a GUI. The user is prompted to answer questions about the project configuration. The answers are used to set cmake cache values.

- `-L[A][H]`: List non-advanced cached variables.

  List cache variables will run CMake and list all the variables from the CMake cache that are not marked as INTERNAL or ADVANCED. This will effectively display current CMake settings, which can be then changed with -D option. Changing some of the variable may

result in more variables being created. If A is specified, then it will display also advanced variables. If H is specified, it will also display help for each variable.

- --build <dir>: Build a CMake-generated project binary tree.

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

  ```
  <dir>          = Project binary directory to be built.
  --target <tgt> = Build <tgt> instead of default targets.
  --config <cfg> = For multi-configuration tools, choose <cfg>.
  --clean-first  = Build target 'clean' first, then build.
                   (To clean only, use --target 'clean'.)
  --use-stderr   =  Don't merge stdout/stderr.
  --             = Pass remaining options to the native tool.
  ```

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

- -N: View mode only.

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

- -P <file>: Process script mode.

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

- --find-package: Run in pkg-config like mode.

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

- `--graphviz=[file]`: Generate graphviz of dependencies.

  Generate a graphviz input file that will contain all the library and executable dependencies in the project.

- `--system-information [file]`: Dump information about this system.

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

- `--debug-trycompile`: Do not delete the try_compile build tree. Only useful on one try_compile at a time.

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

- `--debug-output`: Put cmake in a debug mode.

  Print extra stuff during the cmake run like stack traces with message(send_error) calls.

- `--trace`: Put cmake in trace mode.

  Print a trace of all calls made and from where with message(send_error) calls.

- `--warn-uninitialized`: Warn about uninitialized values.

  Print a warning when an uninitialized variable is used.

- `--warn-unused-vars`: Warn about unused variables.

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

- `--no-warn-unused-cli`: Don't warn about command line options.

  Don't find variables that are declared on the command line, but not used.

- `--check-system-vars`: Find problems with variable usage in system files.

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

- `--help-command cmd [file]`: Print help for a single command and exit.

  Full documentation specific to the given command is displayed. If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- `--help-command-list [file]`: List available listfile commands and exit.

  The list contains all commands for which help may be obtained by using the --help-command argument followed by a command name. If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- `--help-commands [file]`: Print help for all commands and exit.

  Full documentation specific for all current command is displayed.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- `--help-compatcommands [file]`: Print help for compatibility commands.

Full documentation specific for all compatibility commands is displayed.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- --help-module module [file]: Print help for a single module and exit.

  Full documentation specific to the given module is displayed.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- --help-module-list [file]: List available modules and exit.

  The list contains all modules for which help may be obtained by using the --help-module argument followed by a module name. If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- --help-modules [file]: Print help for all modules and exit.

  Full documentation for all modules is displayed. If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- --help-custom-modules [file]: Print help for all custom modules and exit.

  Full documentation for all custom modules is displayed. If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- --help-policy cmp [file]: Print help for a single policy and exit.

Full documentation specific to the given policy is displayed.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- --help-policies [file]: Print help for all policies and exit.

  Full documentation for all policies is displayed.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- --help-property prop [file]: Print help for a single property and exit.

  Full documentation specific to the given property is displayed.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- --help-property-list [file]: List available properties and exit.

  The list contains all properties for which help may be obtained by using the --help-property argument followed by a property name. If a file is specified, the help is written into it.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- --help-properties [file]: Print help for all properties and exit.

  Full documentation for all properties is displayed.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- --help-variable var [file]: Print help for a single variable and exit.

Full documentation specific to the given variable is displayed.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- `--help-variable-list [file]`: List documented variables and exit.

  The list contains all variables for which help may be obtained by using the --help-variable argument followed by a variable name. If a file is specified, the help is written into it.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- `--help-variables [file]`: Print help for all variables and exit.

  Full documentation for all variables is displayed.If a file is specified, the documentation is written into and the output format is determined depending on the filename suffix. Supported are man page, HTML, DocBook and plain text.

- `--copyright [file]`: Print the CMake copyright and exit.

  If a file is specified, the copyright is written into it.

- `--help, -help, -usage, -h, -H, /?`: Print usage information and exit.

  Usage describes the basic command line interface and its options.

- `--help-full [file]`: Print full help and exit.

  Full help displays most of the documentation provided by the UNIX man page. It is provided for use on non-UNIX platforms, but is also convenient if the man page is not installed. If a file is specified, the help is written into it.

- `--help-html [file]`: Print full help in HTML format.

This option is used by CMake authors to help produce web pages. If a file is specified, the help is written into it.

- `--help-man [file]`: Print full help as a UNIX man page and exit.

  This option is used by the cmake build to generate the UNIX man page. If a file is specified, the help is written into it.

- `--version,-version,/V [file]`: Show program name/version banner and exit.

  If a file is specified, the version is written into it.

Generators

- Borland Makefiles
- MSYS Makefiles
- MinGW Makefiles
- NMake Makefiles
- NMake Makefiles JOM
- Ninja
- Unix Makefiles
- Visual Studio 10
- Visual Studio 10 IA64
- Visual Studio 10 Win64
- Visual Studio 11
- Visual Studio 11 ARM
- Visual Studio 11 Win64
- Visual Studio 6
- Visual Studio 7
- Visual Studio 7 .NET 2003
- Visual Studio 8 2005

- Visual Studio 8 2005 Win64
- Visual Studio 9 2008
- Visual Studio 9 2008 IA64
- Visual Studio 9 2008 Win64
- Watcom WMake
- Xcode
- CodeBlocks - MinGW Makefiles
- CodeBlocks - NMake Makefiles
- CodeBlocks - Ninja
- CodeBlocks - Unix Makefiles
- Eclipse CDT4 - MinGW Makefiles
- Eclipse CDT4 - NMake Makefiles
- Eclipse CDT4 - Ninja
- Eclipse CDT4 - Unix Makefiles

The following generators are available on this platform:

- Borland Makefiles: Generates Borland makefiles.
- MSYS Makefiles: Generates MSYS makefiles.

  The makefiles use /bin/sh as the shell. They require msys to be installed on the machine.

- MinGW Makefiles: Generates a make file for use with mingw32-make.

  The makefiles generated use cmd.exe as the shell. They do not require msys or a unix shell.

- NMake Makefiles: Generates NMake makefiles.
- NMake Makefiles JOM: Generates JOM makefiles.
- Ninja: Generates build.ninja files (experimental).

A build.ninja file is generated into the build tree. Recent versions of the ninja program can build the project through the "all" target. An "install" target is also provided.

- Unix Makefiles: Generates standard UNIX makefiles.

  A hierarchy of UNIX makefiles is generated into the build tree. Any standard UNIX-style make program can build the project through the default make target. A "make install" target is also provided.

- Visual Studio 10: Generates Visual Studio 10 project files.
- Visual Studio 10 IA64: Generates Visual Studio 10 Itanium project files.
- Visual Studio 10 Win64: Generates Visual Studio 10 Win64 project files.
- Visual Studio 11: Generates Visual Studio 11 project files.
- Visual Studio 11 ARM: Generates Visual Studio 11 ARM project files.
- Visual Studio 11 Win64: Generates Visual Studio 11 Win64 project files.
- Visual Studio 6: Generates Visual Studio 6 project files.
- Visual Studio 7: Generates Visual Studio .NET 2002 project files.
- Visual Studio 7 .NET 2003: Generates Visual Studio .NET 2003 project files.
- Visual Studio 8 2005: Generates Visual Studio .NET 2005 project files.
- Visual Studio 8 2005 Win64: Generates Visual Studio .NET 2005 Win64 project files.
- Visual Studio 9 2008: Generates Visual Studio 9 2008 project files.
- Visual Studio 9 2008 IA64: Generates Visual Studio 9 2008 Itanium project files.
- Visual Studio 9 2008 Win64: Generates Visual Studio 9 2008 Win64 project files.
- Watcom WMake: Generates Watcom WMake makefiles.
- Xcode: Generates Xcode project files.
- CodeBlocks - MinGW Makefiles: Generates CodeBlocks project files.

  Project files for CodeBlocks will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a PROJECT() call. Additionally a hierarchy

of makefiles is generated into the build tree. The appropriate make program can build the project through the default make target. A "make install" target is also provided.

- CodeBlocks - NMake Makefiles: Generates CodeBlocks project files.

  Project files for CodeBlocks will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a PROJECT() call. Additionally a hierarchy of makefiles is generated into the build tree. The appropriate make program can build the project through the default make target. A "make install" target is also provided.

- CodeBlocks - Ninja: Generates CodeBlocks project files.

  Project files for CodeBlocks will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a PROJECT() call. Additionally a hierarchy of makefiles is generated into the build tree. The appropriate make program can build the project through the default make target. A "make install" target is also provided.

- CodeBlocks - Unix Makefiles: Generates CodeBlocks project files.

  Project files for CodeBlocks will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a PROJECT() call. Additionally a hierarchy of makefiles is generated into the build tree. The appropriate make program can build the project through the default make target. A "make install" target is also provided.

- Eclipse CDT4 - MinGW Makefiles: Generates Eclipse CDT 4.0 project files.

  Project files for Eclipse will be created in the top directory. In out of source builds, a linked resource to the top level source directory will be created.Additionally a hierarchy of makefiles is generated into the build tree. The appropriate make program can build the project through the default make target. A "make install" target is also provided.

- Eclipse CDT4 - NMake Makefiles: Generates Eclipse CDT 4.0 project files.

Project files for Eclipse will be created in the top directory. In out of source builds, a linked resource to the top level source directory will be created.Additionally a hierarchy of makefiles is generated into the build tree. The appropriate make program can build the project through the default make target. A "make install" target is also provided.

- Eclipse CDT4 - Ninja: Generates Eclipse CDT 4.0 project files.

  Project files for Eclipse will be created in the top directory. In out of source builds, a linked resource to the top level source directory will be created.Additionally a hierarchy of makefiles is generated into the build tree. The appropriate make program can build the project through the default make target. A "make install" target is also provided.

- Eclipse CDT4 - Unix Makefiles: Generates Eclipse CDT 4.0 project files.

  Project files for Eclipse will be created in the top directory. In out of source builds, a linked resource to the top level source directory will be created.Additionally a hierarchy of makefiles is generated into the build tree. The appropriate make program can build the project through the default make target. A "make install" target is also provided.

Commands

- add_custom_command
- add_custom_target
- add_definitions
- add_dependencies
- add_executable
- add_library
- add_subdirectory
- add_test
- aux_source_directory
- break

- build_command
- cmake_minimum_required
- cmake_policy
- configure_file
- create_test_sourcelist
- define_property
- else
- elseif
- enable_language
- enable_testing
- endforeach
- endfunction
- endif
- endmacro
- endwhile
- execute_process
- export
- file
- find_file
- find_library
- find_package
- find_path
- find_program
- fltk_wrap_ui
- foreach
- function
- get_cmake_property
- get_directory_property
- get_filename_component

- get_property
- get_source_file_property
- get_target_property
- get_test_property
- if
- include
- include_directories
- include_external_msproject
- include_regular_expression
- install
- link_directories
- list
- load_cache
- load_command
- macro
- mark_as_advanced
- math
- message
- option
- project
- qt_wrap_cpp
- qt_wrap_ui
- remove_definitions
- return
- separate_arguments
- set
- set_directory_properties
- set_property
- set_source_files_properties

- set_target_properties
- set_tests_properties
- site_name
- source_group
- string
- target_link_libraries
- try_compile
- try_run
- unset
- variable_watch
- while

- 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. Do not list the output in more than one independent target that may build in parallel or the two

instances of the rule may conflict (instead use add_custom_target to drive the command and make the other targets depend on that one). 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 in 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 it is a relative path it will be interpreted relative to the build tree directory corresponding to the current source directory. 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 and CXX 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.

Arguments to COMMAND may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluted during build system generation to produce information specific to each 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>
```

References to target names in generator expressions imply target-level dependencies, but NOT file-level dependencies. List target names with the DEPENDS option to add file dependencies.

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 is not specified the command will run whenever the OUTPUT is missing; if the command does not actually create the OUTPUT then the rule will always run. If DEPENDS specifies any target (created by 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 it is a relative path it will be interpreted relative to the build tree directory corresponding to the current source 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. Dependencies added to an IMPORTED target are followed transitively in its place since the

target itself does not build. See the DEPENDS option of ADD_CUSTOM_TARGET and ADD_CUSTOM_COMMAND 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 [GLOBAL])
```

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, but the GLOBAL option extends visibility. It may be referenced like any target built within the project. IMPORTED executables are useful for convenient reference from commands like add_custom_command. Details about the imported executable are specified by setting properties whose names begin in "IMPORTED_". The most important such property is IMPORTED_LOCATION (and its per-configuration version IMPORTED_LOCATION_<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
            [GLOBAL])
```

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, but the GLOBAL option extends visibility. It may be referenced like any target built within the project. IMPORTED libraries are useful for convenient reference from commands like target_link_libraries. Details about the imported library are specified by setting properties whose names begin in "IMPORTED_". 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.

The signature

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

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

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

will include objlib's object files in a library and an executable along with those compiled from their own sources. Object libraries may contain only sources (and headers) that compile to object files. They may contain custom commands generating such sources, but not PRE_BUILD, PRE_LINK, or POST_BUILD commands. Object libraries cannot be imported, exported, installed, or linked.

- 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|...]]
         [WORKING_DIRECTORY dir]
         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. If a WORKING_DIRECTORY option is given then the test will be executed in the given directory.

Arguments after COMMAND may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluted during build system generation to produce information specific to each 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 to build this project.

```
build_command(<variable>
              [CONFIGURATION <config>]
              [PROJECT_NAME <projname>]
              [TARGET <target>])
```

Sets the given <variable> to a string containing the command line for building one configuration of a target in a project using the build tool appropriate for the current CMAKE_GENERATOR.

If CONFIGURATION is omitted, CMake chooses a reasonable default value for multi-configuration generators. CONFIGURATION is ignored for single-configuration generators.

If PROJECT_NAME is omitted, the resulting command line will build the top level PROJECT in the current build tree.

If TARGET is omitted, the resulting command line will build everything, effectively using build target 'all' or 'ALL_BUILD'.

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

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

Sets the given <cachevariable> to a string containing the command to build this project from the root of the build tree using the build tool given by <makecommand>. <makecommand> should be the full path to msdev, devenv, nmake, make or one of the end user build tools.

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

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

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[.tweak]])
```

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 (unless variable CMAKE_POLICY_DEFAULT_CMP<NNNN> sets a default). This effectively requests behavior preferred as of a given CMake version and tells newer CMake versions to warn

about their new policies. 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]
               [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])
```

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.

With NEWLINE_STYLE the line ending could be adjusted:

'UNIX' or 'LF' for \n, 'DOS', 'WIN32' or 'CRLF' for \r\n.

COPYONLY must not be used with NEWLINE_STYLE.

- 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.

```
else(expression)
```

See the if command.

- elseif: Starts the elseif portion of an if block.

  ```
  elseif(expression)
  ```

  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.

  ```
  endforeach(expression)
  ```

  See the FOREACH command.

- endfunction: Ends a list of commands in a function block.

  ```
  endfunction(expression)
  ```

See the function command.

- endif: Ends a list of commands in an if block.

  ```
  endif(expression)
  ```

  See the if command.

- endmacro: Ends a list of commands in a macro block.

  ```
  endmacro(expression)
  ```

  See the macro command.

- endwhile: Ends a list of commands in a while block.

  ```
  endwhile(expression)
  ```

  See the while command.

- execute_process: Execute one or more child processes.

  ```
  execute_process(COMMAND <cmd1> [args1...]]
                  [COMMAND <cmd2> [args2...] [...]]
                  [WORKING_DIRECTORY <directory>]
                  [TIMEOUT <seconds>]
                  [RESULT_VARIABLE <variable>]
                  [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.

Do not set properties that affect the location of a target after passing it to this command. These include properties whose names match "(RUNTIME|LIBRARY|ARCHIVE)_OUTPUT_(NAME|DIRECTORY)(_<CONFIG>)?" or "(IMPLIB_)?(PREFIX|SUFFIX)". Failure to follow this rule is not diagnosed and leaves the location of the target undefined.

```
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(<MD5|SHA1|SHA224|SHA256|SHA384|SHA512> filename variable)
  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 [INACTIVITY_TIMEOUT timeout]
       [TIMEOUT timeout] [STATUS status] [LOG log] [SHOW_PROGRESS]
       [EXPECTED_MD5 sum])
  file(UPLOAD filename url [INACTIVITY_TIMEOUT timeout]
       [TIMEOUT    timeout]    [STATUS    status]    [LOG    log]
[SHOW_PROGRESS])
```

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.

MD5, SHA1, SHA224, SHA256, SHA384, and SHA512 will compute a cryptographic hash of the content of a file.

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. (We do not recommend using GLOB to collect a list of source files from your