CMake 2.8.12 Documentation

This is not the latest CMake version. See the main CMake Documentation index for newer versions.

Generated by `cmake --help-html`.

Master Index CMake 2.8.12

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>`
- `-T <toolset-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 build system generator.

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

- **-T <toolset-name>**: Specify toolset name if supported by generator.

  Some CMake generators support a toolset name to be given to the native build system to choose a compiler. This is supported only on specific generators:

  ```
  Visual Studio >= 10
  Xcode >= 3.0
  ```

  See native build system documentation for allowed toolset names.

- **-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 then be changed with -D option. Changing some of the variables may result in more variables being created. If A is specified, then it will display also advanced variables. If H is specified, it will also display help for each variable.

- **--build <dir>**: 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 output and pass the
                   original stdout/stderr handles to the native
                   tool so it can use the capabilities of the
                   calling terminal (e.g. colored output).
  --             = 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, see CMakeGraphVizOptions.cmake for more.

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

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

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

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

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

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

  Print extra 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 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-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

- **`Visual Studio 6`**
- **`Visual Studio 7`**
- **`Visual Studio 10`**
- **`Visual Studio 11`**
- **`Visual Studio 12`**
- **`Visual Studio 7 .NET 2003`**
- **`Visual Studio 8 2005`**
- **`Visual Studio 9 2008`**
- **`Borland Makefiles`**
- **`NMake Makefiles`**
- **`NMake Makefiles JOM`**
- **`Watcom WMake`**
- **`MSYS Makefiles`**
- **`MinGW Makefiles`**
- **`Unix Makefiles`**
- **`Ninja`**
- **`Ninja`**
- **`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`**
- **`KDevelop3`**
- **`KDevelop3 - Unix Makefiles`**
- **`Sublime Text 2 - MinGW Makefiles`**
- **`Sublime Text 2 - NMake Makefiles`**
- **`Sublime Text 2 - Ninja`**
- **`Sublime Text 2 - Unix Makefiles`**

The following generators are available on this platform:

- **`Visual Studio 6`**: Generates Visual Studio 6 project files.
- **`Visual Studio 7`**: Generates Visual Studio .NET 2002 project files.
- **`Visual Studio 10`**: Generates Visual Studio 10 (2010) project files.

  It is possible to append a space followed by the platform name to create project files for a specific target platform. E.g. "Visual Studio 10 Win64" will create project files for the x64 processor; "Visual Studio 10 IA64" for Itanium.

- **`Visual Studio 11`**: Generates Visual Studio 11 (2012) project files.

  It is possible to append a space followed by the platform name to create project files for a specific target platform. E.g. "Visual Studio 11 Win64" will create project files for the x64 processor; "Visual Studio 11 ARM" for ARM.

- **`Visual Studio 12`**: Generates Visual Studio 12 (2013) project files.

  It is possible to append a space followed by the platform name to create project files for a specific target platform. E.g. "Visual Studio 12 Win64" will create project files for the x64 processor; "Visual Studio 12 ARM" for ARM.

- **`Visual Studio 7 .NET 2003`**: Generates Visual Studio .NET 2003 project files.
- **`Visual Studio 8 2005`**: Generates Visual Studio 8 2005 project files.

It is possible to append a space followed by the platform name to create project files for a specific target platform. E.g. "Visual Studio 8 2005 Win64" will create project files for the x64 processor.

- **`Visual Studio 9 2008`**: Generates Visual Studio 9 2008 project files.

  It is possible to append a space followed by the platform name to create project files for a specific target platform. E.g. "Visual Studio 9 2008 Win64" will create project files for the x64 processor; "Visual Studio 9 2008 IA64" for Itanium.

- **`Borland Makefiles`**: Generates Borland makefiles.
- **`NMake Makefiles`**: Generates NMake makefiles.
- **`NMake Makefiles JOM`**: Generates JOM makefiles.
- **`Watcom WMake`**: Generates Watcom WMake 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.

- **`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.

- **`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.

- **`Xcode`**: Generate 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.

- **`KDevelop3`**: Generates KDevelop 3 project files.

  Project files for KDevelop 3 will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a PROJECT() call. If you change the settings using KDevelop cmake will try its best to keep your changes when

regenerating the project files. Additionally 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.

- **`KDevelop3 - Unix Makefiles`**: Generates KDevelop 3 project files.

  Project files for KDevelop 3 will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a PROJECT() call. If you change the settings using KDevelop cmake will try its best to keep your changes when regenerating the project files. Additionally 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.

- **`Sublime Text 2 - MinGW Makefiles`**: Generates Sublime Text 2 project files.

  Project files for Sublime Text 2 will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a PROJECT() call. Additionally Makefiles (or build.ninja files) are 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.

- **`Sublime Text 2 - NMake Makefiles`**: Generates Sublime Text 2 project files.

  Project files for Sublime Text 2 will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a PROJECT() call. Additionally Makefiles (or build.ninja files) are 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.

- **`Sublime Text 2 - Ninja`**: Generates Sublime Text 2 project files.

  Project files for Sublime Text 2 will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a PROJECT() call. Additionally Makefiles (or build.ninja files) are 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.

- **`Sublime Text 2 - Unix Makefiles`**: Generates Sublime Text 2 project files.

  Project files for Sublime Text 2 will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a PROJECT() call. Additionally Makefiles (or build.ninja files) are 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_compile_options`
- `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_host_system_information`
- `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_compile_definitions
- target_compile_options
- target_include_directories
- target_link_libraries
- try_compile
- try_run
- unset
- variable_watch
- while

- **add_compile_options**: Adds options to the compilation of source files.

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

  Adds options to the compiler command line for sources in the current directory and below. This command can be used to add any options, but alternative commands exist to add preprocessor definitions or include directories. See documentation of the directory and target COMPILE_OPTIONS properties for details. Arguments to add_compile_options may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

  ```
  $<0:...>                  = empty string (ignores "...")
  $<1:...>                  = content of "..."
  $<CONFIG:cfg>             = '1' if config is "cfg", else '0'
  $<CONFIGURATION>          = configuration name
  $<BOOL:...>               = '1' if the '...' is true, else '0'
  $<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
  $<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
  $<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
  $<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
  $<JOIN:list,...>          = joins the list with the content of "..."
  $<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
  $<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
  $<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
  $<C_COMPILER_ID>          = The CMake-id of the C compiler used.
  $<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
  $<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
  $<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
  ```

```
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **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
                                    [<lang2> depend2] ...]
                   [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. The language has to be specified for every file in the IMPLICIT_DEPENDS list. Dependencies discovered from the scanning are added to those of the custom command at build time. Note that the IMPLICIT_DEPENDS option is currently supported only for Makefile generators and will be ignored by other generators.

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 evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
$<0:...>                  = empty string (ignores "...")
$<1:...>                  = content of "..."
$<CONFIG:cfg>             = '1' if config is "cfg", else '0'
$<CONFIGURATION>          = configuration name
$<BOOL:...>               = '1' if the '...' is true, else '0'
$<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
$<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
$<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>          = joins the list with the content of "..."
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>



$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

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-trivial values may be left in the set of flags instead of being converted for reasons of backwards compatibility. See documentation of the directory, target, 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.

The signature

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

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

- **`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. For SHARED and MODULE libraries the POSITION_INDEPENDENT_CODE target property is set to TRUE automatically.

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

The signature

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

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

- **`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 ...]])
  ```

  Add a test called <name>. The test name may not contain spaces, quotes, or other characters special in CMake syntax. 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 evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

  ```
  $<0:...>                = empty string (ignores "...")
  $<1:...>                = content of "..."
  $<CONFIG:cfg>           = '1' if config is "cfg", else '0'
  $<CONFIGURATION>        = configuration name
  $<BOOL:...>             = '1' if the '...' is true, else '0'
  $<STREQUAL:a,b>         = '1' if a is STREQUAL b, else '0'
  ```

```
$<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
$<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>          = joins the list with the content of "..."
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

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_host_system_information**: Query host system specific information.

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

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

<key> can be one of the following values:

```
NUMBER_OF_LOGICAL_CORES   = Number of logical cores.
NUMBER_OF_PHYSICAL_CORES  = Number of physical cores.
HOSTNAME                  = Hostname.
FQDN                      = Fully qualified domain name.
TOTAL_VIRTUAL_MEMORY      = Total virtual memory in megabytes.
AVAILABLE_VIRTUAL_MEMORY  = Available virtual memory in megabytes.
TOTAL_PHYSICAL_MEMORY     = Total physical memory in megabytes.
AVAILABLE_PHYSICAL_MEMORY = Available physical memory in megabytes.
```

- **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.

  If the <input> file is modified the build system will re-run CMake to re-configure the file and generate the build system again.

  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 replaced and ${VAR} will be ignored. This is useful for configuring scripts that use ${VAR}.

  Input file lines of the form "#cmakedefine VAR ..." will be replaced with either "#define VAR ..." or "/* #undef VAR */" depending on whether VAR is set in CMake to any value not considered a false constant by the if() command. (Content of "...", if any, is processed as above.) Input file lines of the form "#cmakedefine01 VAR" will be replaced with either "#define VAR 1" or "#define VAR 0" similarly.

  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(<lang> [OPTIONAL] )
  ```

  This command enables support for the named language in CMake. This is the same as the project command but does not create any of the extra variables that are created by the project command. Example languages are CXX, C, Fortran.

  This command must be called in file scope, not in a function call. Furthermore, it must be called in the highest directory common to all targets using the named language directly for compiling sources or indirectly through link dependencies. It is simplest to enable all needed languages in the top-level directory of a project.

  The OPTIONAL keyword is a placeholder for future implementation and does not currently work.

- **`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> [EXPORT_LINK_INTERFACE_LIBRARIES])
```

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. The EXPORT_LINK_INTERFACE_LIBRARIES keyword, if present, causes the contents of the properties matching (IMPORTED_)?LINK_INTERFACE_LIBRARIES(_<CONFIG>)? to be exported, when policy CMP0022 is NEW. If a library target is included in the export but a target to which it links is not included the behavior is unspecified.

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>)?", "(IMPLIB_)?(PREFIX|SUFFIX)", or "LINKER_LANGUAGE". 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_HASH ALGO=value] [EXPECTED_MD5 sum]
     [TLS_VERIFY on|off] [TLS_CAINFO file])
file(UPLOAD filename url [INACTIVITY_TIMEOUT timeout]
     [TIMEOUT timeout] [STATUS status] [LOG log] [SHOW_PROGRESS])
file(TIMESTAMP filename variable [<format string>] [UTC])
file(GENERATE OUTPUT output_file
     <INPUT input_file|CONTENT input_content>
     [CONDITION expression])
```

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

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 source tree. If no CMakeLists.txt file changes when a source is added or removed then the generated build system cannot know when to ask CMake to regenerate.)

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. This command will also convert the native list delimiters for a list of paths like the PATH environment variable.

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 should be specified as an integer. The INACTIVITY_TIMEOUT specifies an integer number of seconds of inactivity after which the operation should terminate. If EXPECTED_HASH ALGO=value is specified, the operation will verify that the downloaded file's actual hash matches the expected value, where ALGO is one of MD5, SHA1, SHA224, SHA256, SHA384, or SHA512. If it does not match, the operation fails with an error. ("EXPECTED_MD5 sum" is short-hand for "EXPECTED_HASH MD5=sum".) If SHOW_PROGRESS is specified, progress information will be printed as status messages until the operation is complete. For https URLs CMake must be built with OpenSSL. TLS/SSL certificates are not checked by default. Set TLS_VERIFY to ON to check certificates and/or use EXPECTED_HASH to verify downloaded content. Set TLS_CAINFO to specify a custom Certificate Authority file. If either TLS option is not given CMake will check variables CMAKE_TLS_VERIFY and CMAKE_TLS_CAINFO, respectively.

UPLOAD will upload the given file to the given URL. If LOG var is specified a log of the upload 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 should be specified as an integer. The INACTIVITY_TIMEOUT specifies an integer number of seconds of inactivity after which the operation should terminate. If SHOW_PROGRESS is specified, progress information will be printed as status messages until the operation is complete.

TIMESTAMP will write a string representation of the modification time of filename to variable.

Should the command be unable to obtain a timestamp variable will be set to the empty string "".

See documentation of the string TIMESTAMP sub-command for more details.

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).

GENERATE will write an <output_file> with content from an <input_file>, or from <input_content>. The output is generated conditionally based on the content of the <condition>. The file is written at CMake generate-time and the input may contain generator expressions. The <condition>, <output_file> and <input_file> may also contain generator expressions. The <condition> must evaluate to either '0' or '1'. The <output_file> must evaluate to a unique name among all configurations and among all invocations of file(GENERATE).

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

3. Search 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.

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

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 ...] [NAMES_PER_DIR]
          [HINTS path1 [path2 ... ENV var]]
          [PATHS path1 [path2 ... ENV var]]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [DOC "cache documentation string"]
          [NO_DEFAULT_PATH]
          [NO_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/<arch> if CMAKE_LIBRARY_ARCHITECTURE is set, and
<prefix>/lib for each <prefix> in CMAKE_PREFIX_PATH
CMAKE_LIBRARY_PATH
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/<arch> if CMAKE_LIBRARY_ARCHITECTURE is set, and
<prefix>/lib for each <prefix> in CMAKE_PREFIX_PATH
CMAKE_LIBRARY_PATH
CMAKE_FRAMEWORK_PATH
```

3. Search 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.

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

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.

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

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

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

- **`find_package`**: Load settings for an external project.

  ```
  find_package(<package> [version] [EXACT] [QUIET] [MODULE]
               [REQUIRED] [[COMPONENTS] [components...]]
               [OPTIONAL_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 and imported targets documented by the package itself. The QUIET option disables messages if the package cannot be found. The MODULE option disables the second signature documented below. The REQUIRED option stops processing with an error message if the package cannot be found.

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

  The [version] argument requests a version with which the package found should be compatible (format is major[.minor[.patch[.tweak]]]). The EXACT option requests that the version be matched exactly. If no [version] and/or component list is given to a recursive invocation inside a find-module, the corresponding arguments are forwarded automatically from the outer call (including the EXACT flag for [version]). Version support is currently provided only on a package-by-package basis (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 and the MODULE option is not

given the command proceeds to Config mode.

The complete Config mode command signature is:

```
find_package(<package> [version] [EXACT] [QUIET]
             [REQUIRED] [[COMPONENTS] [components...]]
             [CONFIG|NO_MODULE]
             [NO_POLICY_SCOPE]
             [NAMES name1 [name2 ...]]
             [CONFIGS config1 [config2 ...]]
             [HINTS path1 [path2 ... ]]
             [PATHS path1 [path2 ... ]]
             [PATH_SUFFIXES suffix1 [suffix2 ...]]
             [NO_DEFAULT_PATH]
             [NO_CMAKE_ENVIRONMENT_PATH]
             [NO_CMAKE_PATH]
             [NO_SYSTEM_ENVIRONMENT_PATH]
             [NO_CMAKE_PACKAGE_REGISTRY]
             [NO_CMAKE_BUILDS_PATH]
             [NO_CMAKE_SYSTEM_PATH]
             [NO_CMAKE_SYSTEM_PACKAGE_REGISTRY]
             [CMAKE_FIND_ROOT_PATH_BOTH |
              ONLY_CMAKE_FIND_ROOT_PATH |
              NO_CMAKE_FIND_ROOT_PATH])
```

The CONFIG option may be used to skip Module mode explicitly and switch to Config mode. It is synonymous to using NO_MODULE. Config mode 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.

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

If the package configuration file cannot be found CMake will generate an error describing the problem unless the QUIET argument is specified. If REQUIRED is specified and the package is not found a fatal error is generated and the configure step stops executing. If <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. A basic version file containing generic version matching code can be created using the macro write_basic_package_version_file(), see its documentation for more details. 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>/(lib/<arch>|lib|share)/cmake/<name>*/       (U)
<prefix>/(lib/<arch>|lib|share)/<name>*/             (U)
<prefix>/(lib/<arch>|lib|share)/<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). Paths with lib/<arch> are enabled if CMAKE_LIBRARY_ARCHITECTURE is set. 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.

```
CMAKE_PREFIX_PATH
CMAKE_FRAMEWORK_PATH
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.

```
<package>_DIR
CMAKE_PREFIX_PATH
CMAKE_FRAMEWORK_PATH
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.

```
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.

On Windows a <package> may appear under registry key

```
HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\<package>
```

as a REG_SZ value, with arbitrary name, that specifies the directory containing the package configuration file. On UNIX platforms a <package> may appear under the directory

```
~/.cmake/packages/<package>
```

as a file, with arbitrary name, whose content specifies the directory containing the package configuration file. See the export(PACKAGE) command to create user package registry entries for project build trees.

7. Search cmake variables defined in the Platform files for the current system. This can be skipped if NO_CMAKE_SYSTEM_PATH is passed.

```
CMAKE_SYSTEM_PREFIX_PATH
CMAKE_SYSTEM_FRAMEWORK_PATH
CMAKE_SYSTEM_APPBUNDLE_PATH
```

8. Search paths stored in the CMake system package registry. This can be skipped if NO_CMAKE_SYSTEM_PACKAGE_REGISTRY is passed. On Windows a <package> may appear under registry key

```
HKEY_LOCAL_MACHINE\Software\Kitware\CMake\Packages\<package>
```

as a REG_SZ value, with arbitrary name, that specifies the directory containing the package configuration file. There is no system package registry on non-Windows platforms.

9. 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.

Every non-REQUIRED find_package() call can be disabled by setting the variable CMAKE_DISABLE_FIND_PACKAGE_<package> to TRUE. See the documentation for the CMAKE_DISABLE_FIND_PACKAGE_<package> variable for more information.

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

```
<package>_FIND_REQUIRED      = true if REQUIRED option was given
<package>_FIND_QUIETLY       = true if QUIET option was given
<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
<package>_FIND_VERSION_EXACT = true if EXACT option was given
<package>_FIND_COMPONENTS    = list of requested components
<package>_FIND_REQUIRED_<c>  = true if component <c> is required
                               false if component <c> is optional
```

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

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

3. Search 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.

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

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
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
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.

```
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
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 arguments past the last expected argument.

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

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

- **`get_cmake_property`**: Get a 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, VAR will be set to "NOTFOUND". Some supported properties include: VARIABLES, CACHE_VARIABLES, COMMANDS, MACROS, and COMPONENTS.

See also the more general get_property() command.

- **`get_directory_property`**: Get a property of DIRECTORY scope.

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

Store a property of directory scope in the named variable. 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.

See also the more general get_property() command.

- **`get_filename_component`**: Get a specific component of a full filename.

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

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

```
DIRECTORY = Directory without file name
NAME      = File name without directory
EXT       = File name longest extension (.b.c from d/a.b.c)
NAME_WE   = File name without directory or longest extension
ABSOLUTE  = Full path to file
REALPATH  = Full path to existing file with symlinks resolved
PATH      = Legacy alias for DIRECTORY (use for CMake <= 2.8.11)
```

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

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

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

- **`get_property`**: Get a property.

  ```
  get_property(<variable>
               <GLOBAL             |
                DIRECTORY [dir]    |
                TARGET    <target> |
                SOURCE    <source> |
                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

  See also the more general get_property() command.

- **`get_target_property`**: Get a property from a target.

  ```
  get_target_property(VAR target property)
  ```

  Get a property from a target. The value of the property is stored in the variable VAR. If the 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.

  See also the more general get_property() command.

- **`get_test_property`**: Get a property of the test.

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

Get a property from the Test. The value of the property is stored in the variable VAR. If the property is not found, VAR will be set to "NOTFOUND". For a list of standard properties you can type cmake --help-property-list

See also the more general get_property() command.

- **`if`**: Conditionally execute a group of commands.

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

Evaluates the given expression. If the result is true, the commands in the THEN section are invoked. Otherwise, the commands in the else section are invoked. The elseif and else sections are optional. You may have multiple elseif clauses. Note that the expression in the else and endif clause is optional. Long expressions can be used and there is a traditional order of precedence. Parenthetical expressions are evaluated first followed by unary 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(<constant>)
```

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

```
if(<variable>)
```

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

```
if(NOT <expression>)
```

True if the expression is not true.

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

True if both expressions would be considered true individually.

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

True if either expression would be considered true individually.

```
if(COMMAND command-name)
```

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

```
if(POLICY policy-id)
```

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

```
if(TARGET target-name)
```

True if the given name is an existing 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 the file time stamps are exactly the same, an IS_NEWER_THAN comparison returns true, so that any dependent build operations will occur in the event of a tie. This includes the case of passing the same file name for both file1 and file2.

```
if(IS_DIRECTORY directory-name)
```

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

```
if(IS_SYMLINK file-name)
```

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

```
if(IS_ABSOLUTE path)
```

True if the given path is an absolute path.

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

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

```
if(<variable|string> LESS <variable|string>)
if(<variable|string> GREATER <variable|string>)
if(<variable|string> EQUAL <variable|string>)
```

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

```
if(<variable|string> STRLESS <variable|string>)
if(<variable|string> STRGREATER <variable|string>)
if(<variable|string> STREQUAL <variable|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(<variable|string> VERSION_LESS <variable|string>)
if(<variable|string> VERSION_EQUAL <variable|string>)
if(<variable|string> VERSION_GREATER <variable|string>)
```

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

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

appears to the if command as

```
if(var1)
```

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

```
if(var2)
```

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

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

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**: Load and run CMake code from a file or module.

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

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

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

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

- **include_directories**: Add include directories to the build.

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

Add the given directories to those the compiler uses to search for include files. Relative paths are interpreted as relative to the current source directory.

The include directories are added to the directory property INCLUDE_DIRECTORIES for the current CMakeLists file. They are also added to the target property INCLUDE_DIRECTORIES for each target in the current CMakeLists file. The target property values are the ones used by the generators.

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 AFTER or BEFORE explicitly, you can select between appending and prepending, independent of the default.

If the SYSTEM option is given, the compiler will be told the directories are meant as system include directories on some platforms (signalling this setting might achieve effects such as the compiler skipping warnings, or these fixed-install system files not being considered in dependency calculations - see compiler docs).

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

```
include_external_msproject(projectname location
                           [TYPE projectTypeGUID]
                           [GUID projectGUID]
                           [PLATFORM platformName]
                           dep1 dep2 ...)
```

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

TYPE, GUID and PLATFORM are optional parameters that allow one to specify the type of project, id (GUID) of the project and the name of the target platform. This is useful for projects requiring values other than the default (e.g. WIX projects). These options are not supported by the Visual Studio 6 generator.

- **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. The prefix can be relocated at install time using DESTDIR mechanism explained in the CMAKE_INSTALL_PREFIX variable documentation.

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. If COMPONENT is not provided a default component "Unspecified" is created. The default component name may be controlled with the CMAKE_INSTALL_DEFAULT_COMPONENT_NAME variable.

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>]
         [INCLUDES 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 INCLUDES DESTINATION specifies a list of directories which will be added to the INTERFACE_INCLUDE_DIRECTORIES of the <targets> when exported by install(EXPORT). If a relative path is specified, it is treated as relative to the $<INSTALL_PREFIX>.

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.

The EXPORT option associates the installed target files with an export called <export-name>. It must appear before any RUNTIME, LIBRARY, or ARCHIVE options. To actually install the export file itself, call install(EXPORT). 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 signature:

```
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|...]]
[EXPORT_LINK_INTERFACE_LIBRARIES]
[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. The EXPORT_LINK_INTERFACE_LIBRARIES keyword, if present, causes the contents of the properties matching (IMPORTED_)?LINK_INTERFACE_LIBRARIES(_<CONFIG>)? to be exported, when policy CMP0022 is NEW. If a 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. Relative paths given to this command are interpreted as relative to the current source directory, see CMP0015.

  Note that this command is rarely necessary. Library locations returned by find_package() and find_library() are absolute paths. Pass these absolute library file paths directly to the target_link_libraries() command. CMake will ensure the linker finds them.

- **`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.

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

  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 arguments 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 and/or better CMake scope control 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.

It does nothing in script mode.

- **math**: Mathematical expressions.

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

EXPR evaluates mathematical expression and returns 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 processing,
                              but skip generation
FATAL_ERROR    = CMake Error, stop processing and generation
```

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

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

- **option**: Provides an option that the user can optionally select.

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

  Provide an option for the user to select as ON or OFF. If no initial value is provided, OFF is used.

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

- **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 explicitly listing the languages you want to support, e.g. C. By using the special language "NONE" all checks for any language can be disabled. If a variable exists called CMAKE_PROJECT_<projectName>_INCLUDE, the file pointed to by that variable will be included as the last step of the project command.

  The top-level CMakeLists.txt file for a project must contain a literal, direct call to the project() command; loading one through the include() command is not sufficient. If no such call exists CMake will implicitly add one to the top that enables the default languages (C and CXX).

- **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, cache or environment 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. Normally, set will set a regular CMake variable. If CACHE is present, then the <variable> is put in the cache instead, unless it is already in the cache. See section 'Variable types in CMake' below for details of regular and cache variables and their interactions. If CACHE is used, <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, the cache variable is marked as internal, and will not be shown to the user in tools like cmake-gui. This is intended for values that should be persisted in the cache, but which users should not normally change. INTERNAL implies FORCE.

Normally, set(...CACHE...) creates cache variables, but does not modify them. If FORCE is specified, the value of the cache variable is set, even if the variable is already in the cache. This should normally be avoided, as it will remove any changes to the cache variable's value 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). PARENT_SCOPE cannot be combined with CACHE.

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.

*** Variable types in CMake ***

In CMake there are two types of variables: normal variables and cache variables. Normal variables are meant for the internal use of the script (just like variables in most programming languages); they are not persisted across CMake runs. Cache variables (unless set with INTERNAL) are mostly intended for configuration settings where the first CMake run determines a suitable default value, which the user can then override, by editing the cache with tools such as ccmake or cmake-gui. Cache variables are stored in the CMake cache file, and are persisted across CMake runs.

Both types can exist at the same time with the same name but different values. When ${FOO} is evaluated, CMake first looks for a normal variable 'FOO' in scope and uses it if set. If and only if no normal variable exists then it falls back to the cache variable 'FOO'.

Some examples:

The code 'set(FOO "x")' sets the normal variable 'FOO'. It does not touch the cache, but it will hide any existing cache value 'FOO'.

The code 'set(FOO "x" CACHE ...)' checks for 'FOO' in the cache, ignoring any normal variable of the same name. If 'FOO' is in the cache then nothing happens to either the normal variable or the cache variable. If 'FOO' is not in the cache, then it is added to the cache.

Finally, whenever a cache variable is added or modified by a command, CMake also *removes* the normal variable of the same name from the current scope so that an immediately following evaluation of it will expose the newly cached value.

Normally projects should avoid using normal and cache variables of the same name, as this interaction can be hard to follow. However, in some situations it can be useful. One example (used by some projects):

A project has a subproject in its source tree. The child project has its own CMakeLists.txt, which is included from the parent CMakeLists.txt using add_subdirectory(). Now, if the parent and the child project provide the same option (for example a compiler option), the parent gets the first chance to add a user-editable option to the cache. Normally, the child would then use the same value that the parent uses. However, it may be necessary to hard-code the value for the child project's option while still allowing the user to edit the value used by the parent project. The parent project can achieve this simply by setting a normal variable with the same name as the option in a scope sufficient to hide the option's cache variable from the child completely. The parent has already set the cache variable, so the child's set(...CACHE...) will do nothing, and evaluating the option variable will use the value from the normal variable, which hides the cache variable.

- **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] [APPEND_STRING]
             PROPERTY <name> [value1 [value2 ...]])
```

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

GLOBAL scope is unique and does not accept a name.

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

TARGET scope may name zero or more existing targets.

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

TEST scope may name zero or more existing tests.

CACHE scope must name zero or more cache existing entries.

The required PROPERTY option is immediately followed by the name of the property to set. Remaining arguments are used to compose the property value in the form of a semicolon-separated list. If the APPEND option is given the list is appended to any existing property value.If the APPEND_STRING option is given the string is append to any existing property value as string, i.e. it results in a longer string and not a list of strings.

- **set_source_files_properties**: Source files can have properties that affect how they are built.

```
set_source_files_properties([file1 [file2 [...]]]
                            PROPERTIES prop1 value1
                            [prop2 value2 [...]])
```

Set properties associated with source files using a key/value paired list. See properties documentation for those known to CMake. Unrecognized properties are ignored. Source file properties are visible only to targets added in the same directory (CMakeLists.txt).

- **set_target_properties**: Targets can have properties that affect how they are built.

```
set_target_properties(target1 target2 ...
                      PROPERTIES prop1 value1
                      prop2 value2 ...)
```

Set properties on 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.

Properties that affect the name of a target's output file are as follows. The PREFIX and SUFFIX properties override the default target name prefix (such as "lib") and suffix (such as ".so"). IMPORT_PREFIX and IMPORT_SUFFIX are the equivalent properties for the import library corresponding to a DLL (for SHARED library targets). OUTPUT_NAME sets the real name of a target when it is built and can be used to help create two targets of the same name even though CMake requires unique logical target names. There is also a <CONFIG>_OUTPUT_NAME that can set the output name on a per-configuration basis. <CONFIG>_POSTFIX sets a postfix for the real name of the target when it is built under the configuration named by <CONFIG> (in upper-case, such as "DEBUG_POSTFIX"). The value of this property is initialized when the target is created to the value of the variable CMAKE_<CONFIG>_POSTFIX (except for executable targets because earlier CMake versions which did not use this variable for executables).

The LINK_FLAGS property can be used to add extra flags to the link step of a target. LINK_FLAGS_<CONFIG> will add to the configuration <CONFIG>, for example, DEBUG, RELEASE, MINSIZEREL, RELWITHDEBINFO. DEFINE_SYMBOL sets the name of the preprocessor symbol defined when compiling sources in a shared library. If not set here then it is set to target_EXPORTS by default (with some substitutions if the target is not a valid C identifier). This is useful for headers to know whether they are being included from inside their library or outside to properly setup dllexport/dllimport decorations. The COMPILE_FLAGS property sets additional compiler flags used to build sources within the target. It may also be used to pass additional preprocessor definitions.

The LINKER_LANGUAGE property is used to change the tool used to link an executable or shared library. The default is set the language to match the files in the library. CXX and C are common values for this property.

For shared libraries VERSION and SOVERSION can be used to specify the build version and API version respectively. When building or installing appropriate symlinks are created if the platform supports symlinks and the linker supports so-names. If only one of both is specified the missing is assumed to have the same version number. For executables VERSION can be used to specify the build version. When building or installing appropriate symlinks are created if the platform supports symlinks. For shared libraries and executables on Windows the VERSION attribute is parsed to extract a "major.minor" version number. These numbers are used as the image version of the binary.

There are a few properties used to specify RPATH rules. INSTALL_RPATH is a semicolon-separated list specifying the rpath to use in installed targets (for platforms that support it). INSTALL_RPATH_USE_LINK_PATH is a boolean that if set to true will append directories in the linker search path and outside the project to the INSTALL_RPATH. SKIP_BUILD_RPATH is a boolean specifying whether to skip automatic generation of an rpath allowing the target to run from the build tree. BUILD_WITH_INSTALL_RPATH is a boolean specifying whether to link the target in the build tree with the INSTALL_RPATH. This takes precedence over SKIP_BUILD_RPATH and avoids the need for relinking before installation. INSTALL_NAME_DIR is a string specifying the directory portion of the "install_name" field of shared libraries on Mac OSX to use in the installed targets. When the target is created the values of the variables CMAKE_INSTALL_RPATH, CMAKE_INSTALL_RPATH_USE_LINK_PATH, CMAKE_SKIP_BUILD_RPATH, CMAKE_BUILD_WITH_INSTALL_RPATH, and CMAKE_INSTALL_NAME_DIR are used to initialize these properties.

PROJECT_LABEL can be used to change the name of the target in an IDE like visual studio. VS_KEYWORD can be set to change the visual studio keyword, for example Qt integration works better if this is set to Qt4VSv1.0.

VS_SCC_PROJECTNAME, VS_SCC_LOCALPATH, VS_SCC_PROVIDER and VS_SCC_AUXPATH can be set to add support for source control bindings in a Visual Studio project file.

VS_GLOBAL_<variable> can be set to add a Visual Studio project-specific global variable. Qt integration works better if VS_GLOBAL_QtVersion is set to the Qt version FindQt4.cmake found. For example, "4.7.3"

The PRE_INSTALL_SCRIPT and POST_INSTALL_SCRIPT properties are the old way to specify CMake scripts to run before and after installing a target. They are used only when the old INSTALL_TARGETS command is used to install the target. Use the INSTALL command instead.

The EXCLUDE_FROM_DEFAULT_BUILD property is used by the visual studio generators. If it is set to 1 the target will not be part of the default build when you select "Build Solution". This can also be set on a per-configuration basis using EXCLUDE_FROM_DEFAULT_BUILD_<CONFIG>.

- **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. The properties include:

  WILL_FAIL: If set to true, this will invert the pass/fail flag of the test.

  PASS_REGULAR_EXPRESSION: If set, the test output will be checked against the specified regular expressions and at least one of the regular expressions has to match, otherwise the test will fail.

  ```
  Example: PASS_REGULAR_EXPRESSION "TestPassed;All ok"
  ```

  FAIL_REGULAR_EXPRESSION: If set, if the output will match to one of specified regular expressions, the test will fail.

  ```
  Example: PASS_REGULAR_EXPRESSION "[^a-z]Error;ERROR;Failed"
  ```

  Both PASS_REGULAR_EXPRESSION and FAIL_REGULAR_EXPRESSION expect a list of regular expressions.

  TIMEOUT: Setting this will limit the test runtime to the number of seconds specified.

- **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 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(<MD5|SHA1|SHA224|SHA256|SHA384|SHA512>
      <output variable> <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>]
      [RANDOM_SEED <seed>] <output variable>)
string(FIND <string> <substring> <output variable> [REVERSE])
string(TIMESTAMP <output variable> [<format string>] [UTC])
string(MAKE_C_IDENTIFIER <input string> <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.

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

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. If length is -1 the remainder of the string starting at begin will be returned.

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. If an integer RANDOM_SEED is given, its value will be used to seed the random number generator.

FIND will return the position where the given substring was found in the supplied string. If the REVERSE flag was used, the command will search for the position of the last occurrence of the specified substring.

The following characters have special meaning in regular expressions:

```
^         Matches at beginning of input
$         Matches at end of input
.         Matches any single character
[ ]       Matches any character(s) inside the brackets
[^ ]      Matches any character(s) not inside the brackets
 -        Inside brackets, specifies an inclusive range between
          characters on either side e.g. [a-f] is [abcdef]
          To match a literal - using brackets, make it the first
          or the last character e.g. [+*/-] matches basic
          mathematical operators.
*         Matches preceding pattern zero or more times
+         Matches preceding pattern one or more times
```

```
?         Matches preceding pattern zero or once only
|         Matches a pattern on either side of the |
()        Saves a matched subexpression, which can be referenced
          in the REGEX REPLACE operation. Additionally it is saved
          by all regular expression-related commands, including
          e.g. if( MATCHES ), in the variables CMAKE_MATCH_(0..9).
```

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

TIMESTAMP will write a string representation of the current date and/or time to the output variable.

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

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

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

```
%d        The day of the current month (01-31).
%H        The hour on a 24-hour clock (00-23).
%I        The hour on a 12-hour clock (01-12).
%j        The day of the current year (001-366).
%m        The month of the current year (01-12).
%M        The minute of the current hour (00-59).
%S        The second of the current minute.
          60 represents a leap second. (00-60)
%U        The week number of the current year (00-53).
%w        The day of the current week. 0 is Sunday. (0-6)
%y        The last two digits of the current year (00-99)
%Y        The current year.
```

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

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

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

MAKE_C_IDENTIFIER will write a string which can be used as an identifier in C.

- **`target_compile_definitions`**: Add compile definitions to a target.

  ```
  target_compile_definitions(<target> <INTERFACE|PUBLIC|PRIVATE> [items1...]
    [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
  ```

  Specify compile definitions to use when compiling a given target. The named <target> must have been created by a command such as add_executable or add_library and must not be an IMPORTED target. The INTERFACE, PUBLIC and PRIVATE keywords are required to specify the scope of the following arguments. PRIVATE and PUBLIC items will populate the COMPILE_DEFINITIONS property of <target>. PUBLIC and INTERFACE items will populate the INTERFACE_COMPILE_DEFINITIONS property of <target>. The following arguments specify compile definitions. Repeated calls for the same <target> append items in the order called.

  Arguments to target_compile_definitions may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

  ```
  $<0:...>                  = empty string (ignores "...")
  $<1:...>                  = content of "..."
  $<CONFIG:cfg>             = '1' if config is "cfg", else '0'
  $<CONFIGURATION>          = configuration name
  $<BOOL:...>               = '1' if the '...' is true, else '0'
  $<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
  $<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
  $<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
  $<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
  $<JOIN:list,...>          = joins the list with the content of "..."
  $<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
  $<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
  $<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
  $<C_COMPILER_ID>          = The CMake-id of the C compiler used.
  $<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
  $<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
  $<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
  $<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
  $<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
  $<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
  ```

```
  $<C_COMPILER_VERSION>     = The version of the C compiler used.
  $<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
  $<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
  $<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
  $<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>



  $<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
  $<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
  $<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
  $<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
  $<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
  $<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
  $<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **`target_compile_options`**: Add compile options to a target.

```
  target_compile_options(<target> [BEFORE] <INTERFACE|PUBLIC|PRIVATE> [items1...]
    [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

Specify compile options to use when compiling a given target. The named <target> must have been created by a command such as add_executable or add_library and must not be an IMPORTED target. If BEFORE is specified, the content will be prepended to the property instead of being appended.

The INTERFACE, PUBLIC and PRIVATE keywords are required to specify the scope of the following arguments. PRIVATE and PUBLIC items will populate the COMPILE_OPTIONS property of <target>. PUBLIC and INTERFACE items will populate the INTERFACE_COMPILE_OPTIONS property of <target>. The following arguments specify compile opitions. Repeated calls for the same <target> append items in the order called.

Arguments to target_compile_options may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
  $<0:...>                  = empty string (ignores "...")
  $<1:...>                  = content of "..."
  $<CONFIG:cfg>             = '1' if config is "cfg", else '0'
  $<CONFIGURATION>          = configuration name
  $<BOOL:...>               = '1' if the '...' is true, else '0'
  $<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
  $<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
  $<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
  $<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
  $<JOIN:list,...>          = joins the list with the content of "..."
  $<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent export
  $<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
  $<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
  $<C_COMPILER_ID>          = The CMake-id of the C compiler used.
  $<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
  $<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
  $<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
  $<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
  $<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
  $<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
  $<C_COMPILER_VERSION>     = The version of the C compiler used.
  $<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
  $<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
  $<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
  $<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>
```

```
$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **`target_include_directories`**: Add include directories to a target.

```
target_include_directories(<target> [SYSTEM] [BEFORE] <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

Specify include directories or targets to use when compiling a given target. The named <target> must have been created by a command such as add_executable or add_library and must not be an IMPORTED target.

If BEFORE is specified, the content will be prepended to the property instead of being appended.

The INTERFACE, PUBLIC and PRIVATE keywords are required to specify the scope of the following arguments. PRIVATE and PUBLIC items will populate the INCLUDE_DIRECTORIES property of <target>. PUBLIC and INTERFACE items will populate the INTERFACE_INCLUDE_DIRECTORIES property of <target>. The following arguments specify include directories. Specified include directories may be absolute paths or relative paths. Repeated calls for the same <target> append items in the order called.If SYSTEM is specified, the compiler will be told the directories are meant as system include directories on some platforms (signalling this setting might achieve effects such as the compiler skipping warnings, or these fixed-install system files not being considered in dependency calculations - see compiler docs). If SYSTEM is used together with PUBLIC or INTERFACE, the INTERFACE_SYSTEM_INCLUDE_DIRECTORIES target property will be populated with the specified directories.

Arguments to target_include_directories may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
$<0:...>                  = empty string (ignores "...")
$<1:...>                  = content of "..."
$<CONFIG:cfg>             = '1' if config is "cfg", else '0'
$<CONFIGURATION>          = configuration name
$<BOOL:...>               = '1' if the '...' is true, else '0'
$<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
$<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
$<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>          = joins the list with the content of "..."
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **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. The named <target> must have been created in the current directory by a command such as add_executable or add_library. The remaining arguments specify library names or flags. Repeated calls for the same <target> append items in the order called.

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 INTERFACE_LINK_LIBRARIES target property to override the set of transitive link dependencies for a target. Calls to other signatures of this command may set the property making any libraries linked exclusively by this signature private.

CMake will also propagate "usage requirements" from linked library targets. Usage requirements affect compilation of sources in the <target>. They are specified by properties defined on linked targets. During generation of the build system, CMake integrates usage requirement property values with the corresponding build properties for <target>:

```
INTERFACE_COMPILE_DEFINITONS: Appends to COMPILE_DEFINITONS
INTERFACE_INCLUDE_DIRECTORIES: Appends to INCLUDE_DIRECTORIES
INTERFACE_POSITION_INDEPENDENT_CODE: Sets POSITION_INDEPENDENT_CODE
  or checked for consistency with existing value
```

If an <item> is a library in a Mac OX framework, the Headers directory of the framework will also be processed as a "usage requirement". This has the same effect as passing the framework directory as an include directory.
target_link_libraries(<target>

```
                      <PRIVATE|PUBLIC|INTERFACE> <lib> ...
                      [<PRIVATE|PUBLIC|INTERFACE> <lib> ... ] ...])
```

The PUBLIC, PRIVATE and INTERFACE keywords can be used to specify both the link dependencies and the link interface in one command. Libraries and targets following PUBLIC are linked to, and are made part of the link interface. Libraries and targets following PRIVATE are linked to, but are not made part of the link interface. Libraries following INTERFACE are appended to the link interface and are not used for linking <target>.

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

The LINK_INTERFACE_LIBRARIES mode appends the libraries to the INTERFACE_LINK_LIBRARIES target property instead of

using them for linking. If policy CMP0022 is not NEW, then this mode also appends libraries to the LINK_INTERFACE_LIBRARIES and its per-configuration equivalent. This signature is for compatibility only. Prefer the INTERFACE mode instead. Libraries specified as "debug" are wrapped in a generator expression to correspond to debug builds. If policy CMP0022 is not NEW, the libraries are also appended to the LINK_INTERFACE_LIBRARIES_DEBUG property (or to the properties corresponding to configurations listed in the DEBUG_CONFIGURATIONS global property if it is set). Libraries specified as "optimized" are appended to the INTERFACE_LINK_LIBRARIES property. If policy CMP0022 is not NEW, they are also appended to the LINK_INTERFACE_LIBRARIES property. Libraries specified as "general" (or without any keyword) are treated as if specified for both "debug" and "optimized".

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

The LINK_PUBLIC and LINK_PRIVATE modes can be used to specify both the link dependencies and the link interface in one command. This signature is for compatibility only. Prefer the PUBLIC or PRIVATE keywords instead. Libraries and targets following LINK_PUBLIC are linked to, and are made part of the INTERFACE_LINK_LIBRARIES. If policy CMP0022 is not NEW, they are also made part of the LINK_INTERFACE_LIBRARIES. Libraries and targets following LINK_PRIVATE are linked to, but are not made part of the INTERFACE_LINK_LIBRARIES (or LINK_INTERFACE_LIBRARIES).

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.)

Arguments to target_link_libraries may use "generator expressions" with the syntax "$<...>". Note however, that generator expressions will not be used in OLD handling of CMP0003 or CMP0004.

Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
$<0:...>                  = empty string (ignores "...")
$<1:...>                  = content of "..."
$<CONFIG:cfg>             = '1' if config is "cfg", else '0'
$<CONFIGURATION>          = configuration name
$<BOOL:...>               = '1' if the '...' is true, else '0'
$<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
$<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
$<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>          = joins the list with the content of "..."
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>



$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **`try_compile`**: Try building some code.

  ```
  try_compile(RESULT_VAR <bindir> <srcdir>
              <projectName> [targetName] [CMAKE_FLAGS flags...]
              [OUTPUT_VARIABLE <var>])
  ```

  Try building a project. In this form, srcdir should contain a complete CMake project with a CMakeLists.txt file and all sources. The bindir and srcdir will not be deleted after this command is run. Specify targetName to build a specific target instead of the 'all' or 'ALL_BUILD' target.

  ```
  try_compile(RESULT_VAR <bindir> <srcfile|SOURCES srcfile...>
              [CMAKE_FLAGS flags...]
              [COMPILE_DEFINITIONS flags...]
              [LINK_LIBRARIES libs...]
              [OUTPUT_VARIABLE <var>]
              [COPY_FILE <fileName> [COPY_FILE_ERROR <var>]])
  ```

  Try building an executable from one or more source files. In this form the user need only supply one or more source files that include a definition for 'main'. CMake will create a CMakeLists.txt file to build the source(s) as an executable. Specify COPY_FILE to get a copy of the linked executable at the given fileName and optionally COPY_FILE_ERROR to capture any error.

  In this version all files in bindir/CMakeFiles/CMakeTmp will be cleaned automatically. For debugging, --debug-trycompile can be passed to cmake to avoid this clean. However, multiple sequential try_compile operations reuse this single output directory. If you use --debug-trycompile, you can only debug one try_compile call at a time. The recommended procedure is to configure with cmake all the way through once, then delete the cache entry associated with the try_compile call of interest, and then re-run cmake again with --debug-trycompile.

  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.

  The srcfile signature also accepts a LINK_LIBRARIES argument which may contain a list of libraries or IMPORTED targets which will be linked to in the generated project. If LINK_LIBRARIES is specified as a parameter to try_compile, then any LINK_LIBRARIES passed as CMAKE_FLAGS will be ignored.

  try_compile creates a CMakeList.txt file on the fly that looks like this:

  ```
  add_definitions( <expanded COMPILE_DEFINITIONS from calling 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. The success or failure of the try_compile, i.e. TRUE or FALSE respectively, is returned in RESULT_VAR. CMAKE_FLAGS can be used to pass -DVAR:TYPE=VALUE flags to the cmake that is run during the build. Set variable CMAKE_TRY_COMPILE_CONFIGURATION to choose a build configuration.

- **`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 had 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.

Set variable CMAKE_TRY_COMPILE_CONFIGURATION to choose a build configuration.

- **`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.

Properties

```
CMake Properties - Properties supported by CMake, the Cross-Platform Makefile Generator.
```

This is the documentation for the properties supported by CMake. Properties can have different scopes. They can either be assigned to a source file, a directory, a target or globally to CMake. By modifying the values of properties the behaviour of the build system can be customized.

Properties of Global Scope

- **`ALLOW_DUPLICATE_CUSTOM_TARGETS`**
- **`AUTOMOC_TARGETS_FOLDER`**
- **`DEBUG_CONFIGURATIONS`**
- **`DISABLED_FEATURES`**
- **`ENABLED_FEATURES`**
- **`ENABLED_LANGUAGES`**
- **`FIND_LIBRARY_USE_LIB64_PATHS`**
- **`FIND_LIBRARY_USE_OPENBSD_VERSIONING`**
- **`GLOBAL_DEPENDS_DEBUG_MODE`**
- **`GLOBAL_DEPENDS_NO_CYCLES`**
- **`IN_TRY_COMPILE`**
- **`PACKAGES_FOUND`**
- **`PACKAGES_NOT_FOUND`**

- **PREDEFINED_TARGETS_FOLDER**
- **REPORT_UNDEFINED_PROPERTIES**
- **RULE_LAUNCH_COMPILE**
- **RULE_LAUNCH_CUSTOM**
- **RULE_LAUNCH_LINK**
- **RULE_MESSAGES**
- **TARGET_ARCHIVES_MAY_BE_SHARED_LIBS**
- **TARGET_SUPPORTS_SHARED_LIBS**
- **USE_FOLDERS**
- **__CMAKE_DELETE_CACHE_CHANGE_VARS_**

- **ALLOW_DUPLICATE_CUSTOM_TARGETS**: Allow duplicate custom targets to be created.

  Normally CMake requires that all targets built in a project have globally unique logical names (see policy CMP0002). This is necessary to generate meaningful project file names in Xcode and VS IDE generators. It also allows the target names to be referenced unambiguously.

  Makefile generators are capable of supporting duplicate custom target names. For projects that care only about Makefile generators and do not wish to support Xcode or VS IDE generators, one may set this property to true to allow duplicate custom targets. The property allows multiple add_custom_target command calls in different directories to specify the same target name. However, setting this property will cause non-Makefile generators to produce an error and refuse to generate the project.

- **AUTOMOC_TARGETS_FOLDER**: Name of FOLDER for *_automoc targets that are added automatically by CMake for targets for which AUTOMOC is enabled.

  If not set, CMake uses the FOLDER property of the parent target as a default value for this property. See also the documentation for the FOLDER target property and the AUTOMOC target property.

- **DEBUG_CONFIGURATIONS**: Specify which configurations are for debugging.

  The value must be a semi-colon separated list of configuration names. Currently this property is used only by the target_link_libraries command (see its documentation for details). Additional uses may be defined in the future.

  This property must be set at the top level of the project and before the first target_link_libraries command invocation. If any entry in the list does not match a valid configuration for the project the behavior is undefined.

- **DISABLED_FEATURES**: List of features which are disabled during the CMake run.

  List of features which are disabled during the CMake run. By default it contains the names of all packages which were not found. This is determined using the <NAME>_FOUND variables. Packages which are searched QUIET are not listed. A project can add its own features to this list. This property is used by the macros in FeatureSummary.cmake.

- **ENABLED_FEATURES**: List of features which are enabled during the CMake run.

  List of features which are enabled during the CMake run. By default it contains the names of all packages which were found. This is determined using the <NAME>_FOUND variables. Packages which are searched QUIET are not listed. A project can add its own features to this list. This property is used by the macros in FeatureSummary.cmake.

- **ENABLED_LANGUAGES**: Read-only property that contains the list of currently enabled languages

  Set to list of currently enabled languages.

- **FIND_LIBRARY_USE_LIB64_PATHS**: Whether FIND_LIBRARY should automatically search lib64 directories.

  FIND_LIBRARY_USE_LIB64_PATHS is a boolean specifying whether the FIND_LIBRARY command should automatically search the lib64 variant of directories called lib in the search path when building 64-bit binaries.

- **FIND_LIBRARY_USE_OPENBSD_VERSIONING**: Whether FIND_LIBRARY should find OpenBSD-style shared libraries.

  This property is a boolean specifying whether the FIND_LIBRARY command should find shared libraries with OpenBSD-style versioned extension: ".so.<major>.<minor>". The property is set to true on OpenBSD and false on other platforms.

- **GLOBAL_DEPENDS_DEBUG_MODE**: Enable global target dependency graph debug mode.

  CMake automatically analyzes the global inter-target dependency graph at the beginning of native build system generation. This property causes it to display details of its analysis to stderr.

- **GLOBAL_DEPENDS_NO_CYCLES**: Disallow global target dependency graph cycles.

  CMake automatically analyzes the global inter-target dependency graph at the beginning of native build system generation. It reports an error if the dependency graph contains a cycle that does not consist of all STATIC library targets. This property tells CMake to disallow all cycles completely, even among static libraries.

- **IN_TRY_COMPILE**: Read-only property that is true during a try-compile configuration.

  True when building a project inside a TRY_COMPILE or TRY_RUN command.

- **PACKAGES_FOUND**: List of packages which were found during the CMake run.

  List of packages which were found during the CMake run. Whether a package has been found is determined using the <NAME>_FOUND variables.

- **PACKAGES_NOT_FOUND**: List of packages which were not found during the CMake run.

  List of packages which were not found during the CMake run. Whether a package has been found is determined using the <NAME>_FOUND variables.

- **PREDEFINED_TARGETS_FOLDER**: Name of FOLDER for targets that are added automatically by CMake.

  If not set, CMake uses "CMakePredefinedTargets" as a default value for this property. Targets such as INSTALL, PACKAGE and RUN_TESTS will be organized into this FOLDER. See also the documentation for the FOLDER target property.

- **REPORT_UNDEFINED_PROPERTIES**: If set, report any undefined properties to this file.

  If this property is set to a filename then when CMake runs it will report any properties or variables that were accessed but not defined into the filename specified in this property.

- **RULE_LAUNCH_COMPILE**: Specify a launcher for compile rules.

  Makefile generators prefix compiler commands with the given launcher command line. This is intended to allow launchers to intercept build problems with high granularity. Non-Makefile generators currently ignore this property.

- **RULE_LAUNCH_CUSTOM**: Specify a launcher for custom rules.

  Makefile generators prefix custom commands with the given launcher command line. This is intended to allow launchers to intercept build problems with high granularity. Non-Makefile generators currently ignore this property.

- **RULE_LAUNCH_LINK**: Specify a launcher for link rules.

  Makefile generators prefix link and archive commands with the given launcher command line. This is intended to allow launchers to intercept build problems with high granularity. Non-Makefile generators currently ignore this property.

- **RULE_MESSAGES**: Specify whether to report a message for each make rule.

  This property specifies whether Makefile generators should add a progress message describing what each build rule does. If the property is not set the default is ON. Set the property to OFF to disable granular messages and report only as each target completes. This is intended to allow scripted builds to avoid the build time cost of detailed reports. If a CMAKE_RULE_MESSAGES cache entry exists its value initializes the value of this property. Non-Makefile generators currently ignore this property.

- **TARGET_ARCHIVES_MAY_BE_SHARED_LIBS**: Set if shared libraries may be named like archives.

  On AIX shared libraries may be named "lib<name>.a". This property is set to true on such platforms.

- **TARGET_SUPPORTS_SHARED_LIBS**: Does the target platform support shared libraries.

  TARGET_SUPPORTS_SHARED_LIBS is a boolean specifying whether the target platform supports shared libraries. Basically all current general general purpose OS do so, the exception are usually embedded systems with no or special OSs.

- **USE_FOLDERS**: Use the FOLDER target property to organize targets into folders.

  If not set, CMake treats this property as OFF by default. CMake generators that are capable of organizing into a hierarchy of folders use the values of the FOLDER target property to name those folders. See also the documentation for the FOLDER target property.

- **__CMAKE_DELETE_CACHE_CHANGE_VARS_**: Internal property

  Used to detect compiler changes, Do not set.

Properties on Directories

- ADDITIONAL_MAKE_CLEAN_FILES
- CACHE_VARIABLES
- CLEAN_NO_CUSTOM
- COMPILE_DEFINITIONS
- COMPILE_DEFINITIONS_<CONFIG>
- COMPILE_DEFINITIONS_<CONFIG>
- COMPILE_OPTIONS
- DEFINITIONS
- EXCLUDE_FROM_ALL
- IMPLICIT_DEPENDS_INCLUDE_TRANSFORM
- INCLUDE_DIRECTORIES
- INCLUDE_REGULAR_EXPRESSION
- INTERPROCEDURAL_OPTIMIZATION
- INTERPROCEDURAL_OPTIMIZATION_<CONFIG>
- LINK_DIRECTORIES
- LISTFILE_STACK
- MACROS
- PARENT_DIRECTORY
- RULE_LAUNCH_COMPILE
- RULE_LAUNCH_CUSTOM
- RULE_LAUNCH_LINK
- TEST_INCLUDE_FILE
- VARIABLES

- **VS_GLOBAL_SECTION_POST_<section>**
- **VS_GLOBAL_SECTION_PRE_<section>**

- **ADDITIONAL_MAKE_CLEAN_FILES**: Additional files to clean during the make clean stage.

  A list of files that will be cleaned as a part of the "make clean" stage.

- **CACHE_VARIABLES**: List of cache variables available in the current directory.

  This read-only property specifies the list of CMake cache variables currently defined. It is intended for debugging purposes.

- **CLEAN_NO_CUSTOM**: Should the output of custom commands be left.

  If this is true then the outputs of custom commands for this directory will not be removed during the "make clean" stage.

- **COMPILE_DEFINITIONS**: Preprocessor definitions for compiling a directory's sources.

  The COMPILE_DEFINITIONS property may be set to a semicolon-separated list of preprocessor definitions using the syntax VAR or VAR=value. Function-style definitions are not supported. CMake will automatically escape the value correctly for the native build system (note that CMake language syntax may require escapes to specify some values). This property may be set on a per-configuration basis using the name COMPILE_DEFINITIONS_<CONFIG> where <CONFIG> is an upper-case name (ex. "COMPILE_DEFINITIONS_DEBUG"). This property will be initialized in each directory by its value in the directory's parent.

  CMake will automatically drop some definitions that are not supported by the native build tool. The VS6 IDE does not support definition values with spaces (but NMake does).

  Disclaimer: Most native build tools have poor support for escaping certain values. CMake has work-arounds for many cases but some values may just not be possible to pass correctly. If a value does not seem to be escaped correctly, do not attempt to work-around the problem by adding escape sequences to the value. Your work-around may break in a future version of CMake that has improved escape support. Instead consider defining the macro in a (configured) header file. Then report the limitation. Known limitations include:

  ```
  #          - broken almost everywhere
  ;          - broken in VS IDE 7.0 and Borland Makefiles
  ,          - broken in VS IDE
  %          - broken in some cases in NMake
  & |        - broken in some cases on MinGW
  ^ < > \"   - broken in most Make tools on Windows
  ```

  CMake does not reject these values outright because they do work in some cases. Use with caution.

- **COMPILE_DEFINITIONS_<CONFIG>**: Per-configuration preprocessor definitions in a directory.

  This is the configuration-specific version of COMPILE_DEFINITIONS. This property will be initialized in each directory by its value in the directory's parent.

- **COMPILE_OPTIONS**: List of options to pass to the compiler.

  This property specifies the list of directories given so far for this property. This property exists on directories and targets.

  The target property values are used by the generators to set the options for the compiler.

  Contents of COMPILE_OPTIONS may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

  ```
  $<0:...>                  = empty string (ignores "...")
  $<1:...>                  = content of "..."
  $<CONFIG:cfg>             = '1' if config is "cfg", else '0'
  $<CONFIGURATION>          = configuration name
  $<BOOL:...>               = '1' if the '...' is true, else '0'
  $<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
  $<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
  $<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
  $<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
  $<JOIN:list,...>          = joins the list with the content of "..."
  $<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
  $<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
  $<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
  $<C_COMPILER_ID>          = The CMake-id of the C compiler used.
  $<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
  $<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
  $<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
  $<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
  $<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
  $<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
  $<C_COMPILER_VERSION>     = The version of the C compiler used.
  $<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
  $<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
  $<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
  ```

```
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>    = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>           = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>          = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>            = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>             = '0' if all '?' are '0', else '1'
$<NOT:?>                   = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>    = The value of the property prop on the target on which the generator expression is evaluated.
```

- **DEFINITIONS**: For CMake 2.4 compatibility only. Use COMPILE_DEFINITIONS instead.

  This read-only property specifies the list of flags given so far to the add_definitions command. It is intended for debugging purposes. Use the COMPILE_DEFINITIONS instead.

- **EXCLUDE_FROM_ALL**: Exclude the directory from the all target of its parent.

  A property on a directory that indicates if its targets are excluded from the default build target. If it is not, then with a Makefile for example typing make will cause the targets to be built. The same concept applies to the default build of other generators.

- **IMPLICIT_DEPENDS_INCLUDE_TRANSFORM**: Specify #include line transforms for dependencies in a directory.

  This property specifies rules to transform macro-like #include lines during implicit dependency scanning of C and C++ source files. The list of rules must be semicolon-separated with each entry of the form "A_MACRO(%)=value-with-%" (the % must be literal). During dependency scanning occurrences of A_MACRO(...) on #include lines will be replaced by the value given with the macro argument substituted for '%'. For example, the entry

  ```
  MYDIR(%)=<mydir/%>
  ```

  will convert lines of the form

  ```
  #include MYDIR(myheader.h)
  ```

  to

  ```
  #include <mydir/myheader.h>
  ```

  allowing the dependency to be followed.

  This property applies to sources in all targets within a directory. The property value is initialized in each directory by its value in the directory's parent.

- **INCLUDE_DIRECTORIES**: List of preprocessor include file search directories.

  This property specifies the list of directories given so far to the include_directories command. This property exists on directories and targets. In addition to accepting values from the include_directories command, values may be set directly on any directory or any target using the set_property command. A target gets its initial value for this property from the value of the directory property. A directory gets its initial value from its parent directory if it has one. Both directory and target property values are adjusted by calls to the include_directories command.

  The target property values are used by the generators to set the include paths for the compiler. See also the include_directories command.

- **INCLUDE_REGULAR_EXPRESSION**: Include file scanning regular expression.

  This read-only property specifies the regular expression used during dependency scanning to match include files that should be followed. See the include_regular_expression command.

- **INTERPROCEDURAL_OPTIMIZATION**: Enable interprocedural optimization for targets in a directory.

  If set to true, enables interprocedural optimizations if they are known to be supported by the compiler.

- **INTERPROCEDURAL_OPTIMIZATION_<CONFIG>**: Per-configuration interprocedural optimization for a directory.

This is a per-configuration version of INTERPROCEDURAL_OPTIMIZATION. If set, this property overrides the generic property for the named configuration.

- **LINK_DIRECTORIES**: List of linker search directories.

  This read-only property specifies the list of directories given so far to the link_directories command. It is intended for debugging purposes.

- **LISTFILE_STACK**: The current stack of listfiles being processed.

  This property is mainly useful when trying to debug errors in your CMake scripts. It returns a list of what list files are currently being processed, in order. So if one listfile does an INCLUDE command then that is effectively pushing the included listfile onto the stack.

- **MACROS**: List of macro commands available in the current directory.

  This read-only property specifies the list of CMake macros currently defined. It is intended for debugging purposes. See the macro command.

- **PARENT_DIRECTORY**: Source directory that added current subdirectory.

  This read-only property specifies the source directory that added the current source directory as a subdirectory of the build. In the top-level directory the value is the empty-string.

- **RULE_LAUNCH_COMPILE**: Specify a launcher for compile rules.

  See the global property of the same name for details. This overrides the global property for a directory.

- **RULE_LAUNCH_CUSTOM**: Specify a launcher for custom rules.

  See the global property of the same name for details. This overrides the global property for a directory.

- **RULE_LAUNCH_LINK**: Specify a launcher for link rules.

  See the global property of the same name for details. This overrides the global property for a directory.

- **TEST_INCLUDE_FILE**: A cmake file that will be included when ctest is run.

  If you specify TEST_INCLUDE_FILE, that file will be included and processed when ctest is run on the directory.

- **VARIABLES**: List of variables defined in the current directory.

  This read-only property specifies the list of CMake variables currently defined. It is intended for debugging purposes.

- **VS_GLOBAL_SECTION_POST_<section>**: Specify a postSolution global section in Visual Studio.

  Setting a property like this generates an entry of the following form in the solution file:

  ```
  GlobalSection(<section>) = postSolution
    <contents based on property value>
  EndGlobalSection
  ```

  The property must be set to a semicolon-separated list of key=value pairs. Each such pair will be transformed into an entry in the solution global section. Whitespace around key and value is ignored. List elements which do not contain an equal sign are skipped.

  This property only works for Visual Studio 7 and above; it is ignored on other generators. The property only applies when set on a directory whose CMakeLists.txt contains a project() command.

  Note that CMake generates postSolution sections ExtensibilityGlobals and ExtensibilityAddIns by default. If you set the corresponding property, it will override the default section. For example, setting VS_GLOBAL_SECTION_POST_ExtensibilityGlobals will override the default contents of the ExtensibilityGlobals section, while keeping ExtensibilityAddIns on its default.

- **VS_GLOBAL_SECTION_PRE_<section>**: Specify a preSolution global section in Visual Studio.

  Setting a property like this generates an entry of the following form in the solution file:

  ```
  GlobalSection(<section>) = preSolution
    <contents based on property value>
  EndGlobalSection
  ```

  The property must be set to a semicolon-separated list of key=value pairs. Each such pair will be transformed into an entry in the solution global section. Whitespace around key and value is ignored. List elements which do not contain an equal sign are skipped.

  This property only works for Visual Studio 7 and above; it is ignored on other generators. The property only applies when set on a directory whose CMakeLists.txt contains a project() command.

Properties on Targets

- **<CONFIG>_OUTPUT_NAME**
- **<CONFIG>_POSTFIX**
- **<LANG>_VISIBILITY_PRESET**

- ALIASED_TARGET
- ARCHIVE_OUTPUT_DIRECTORY
- ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>
- ARCHIVE_OUTPUT_NAME
- ARCHIVE_OUTPUT_NAME_<CONFIG>
- AUTOMOC
- AUTOMOC_MOC_OPTIONS
- BUILD_WITH_INSTALL_RPATH
- BUNDLE
- BUNDLE_EXTENSION
- COMPATIBLE_INTERFACE_BOOL
- COMPATIBLE_INTERFACE_STRING
- COMPILE_DEFINITIONS
- COMPILE_DEFINITIONS_<CONFIG>
- COMPILE_FLAGS
- COMPILE_OPTIONS
- DEBUG_POSTFIX
- DEFINE_SYMBOL
- ENABLE_EXPORTS
- EXCLUDE_FROM_ALL
- EXCLUDE_FROM_DEFAULT_BUILD
- EXCLUDE_FROM_DEFAULT_BUILD_<CONFIG>
- EXPORT_NAME
- EchoString
- FOLDER
- FRAMEWORK
- Fortran_FORMAT
- Fortran_MODULE_DIRECTORY
- GENERATOR_FILE_NAME
- GNUtoMS
- HAS_CXX
- IMPLICIT_DEPENDS_INCLUDE_TRANSFORM
- IMPORTED
- IMPORTED_CONFIGURATIONS
- IMPORTED_IMPLIB
- IMPORTED_IMPLIB_<CONFIG>
- IMPORTED_LINK_DEPENDENT_LIBRARIES
- IMPORTED_LINK_DEPENDENT_LIBRARIES_<CONFIG>
- IMPORTED_LINK_INTERFACE_LANGUAGES
- IMPORTED_LINK_INTERFACE_LANGUAGES_<CONFIG>
- IMPORTED_LINK_INTERFACE_LIBRARIES
- IMPORTED_LINK_INTERFACE_LIBRARIES_<CONFIG>
- IMPORTED_LINK_INTERFACE_MULTIPLICITY
- IMPORTED_LINK_INTERFACE_MULTIPLICITY_<CONFIG>
- IMPORTED_LOCATION
- IMPORTED_LOCATION_<CONFIG>
- IMPORTED_NO_SONAME
- IMPORTED_NO_SONAME_<CONFIG>
- IMPORTED_SONAME
- IMPORTED_SONAME_<CONFIG>
- IMPORT_PREFIX
- IMPORT_SUFFIX
- INCLUDE_DIRECTORIES
- INSTALL_NAME_DIR
- INSTALL_RPATH
- INSTALL_RPATH_USE_LINK_PATH
- INTERFACE_COMPILE_DEFINITIONS
- INTERFACE_COMPILE_OPTIONS
- INTERFACE_INCLUDE_DIRECTORIES
- INTERFACE_LINK_LIBRARIES
- INTERFACE_POSITION_INDEPENDENT_CODE
- INTERFACE_SYSTEM_INCLUDE_DIRECTORIES
- INTERPROCEDURAL_OPTIMIZATION
- INTERPROCEDURAL_OPTIMIZATION_<CONFIG>
- LABELS
- LIBRARY_OUTPUT_DIRECTORY
- LIBRARY_OUTPUT_DIRECTORY_<CONFIG>
- LIBRARY_OUTPUT_NAME
- LIBRARY_OUTPUT_NAME_<CONFIG>
- LINKER_LANGUAGE
- LINK_DEPENDS

- LINK_DEPENDS_NO_SHARED
- LINK_FLAGS
- LINK_FLAGS_<CONFIG>
- LINK_INTERFACE_LIBRARIES
- LINK_INTERFACE_LIBRARIES_<CONFIG>
- LINK_INTERFACE_MULTIPLICITY
- LINK_INTERFACE_MULTIPLICITY_<CONFIG>
- LINK_LIBRARIES
- LINK_SEARCH_END_STATIC
- LINK_SEARCH_START_STATIC
- LOCATION
- LOCATION_<CONFIG>
- MACOSX_BUNDLE
- MACOSX_BUNDLE_INFO_PLIST
- MACOSX_FRAMEWORK_INFO_PLIST
- MACOSX_RPATH
- MAP_IMPORTED_CONFIG_<CONFIG>
- NAME
- NO_SONAME
- OSX_ARCHITECTURES
- OSX_ARCHITECTURES_<CONFIG>
- OUTPUT_NAME
- OUTPUT_NAME_<CONFIG>
- PDB_NAME
- PDB_NAME_<CONFIG>
- PDB_OUTPUT_DIRECTORY
- PDB_OUTPUT_DIRECTORY_<CONFIG>
- POSITION_INDEPENDENT_CODE
- POST_INSTALL_SCRIPT
- PREFIX
- PRE_INSTALL_SCRIPT
- PRIVATE_HEADER
- PROJECT_LABEL
- PUBLIC_HEADER
- RESOURCE
- RULE_LAUNCH_COMPILE
- RULE_LAUNCH_CUSTOM
- RULE_LAUNCH_LINK
- RUNTIME_OUTPUT_DIRECTORY
- RUNTIME_OUTPUT_DIRECTORY_<CONFIG>
- RUNTIME_OUTPUT_NAME
- RUNTIME_OUTPUT_NAME_<CONFIG>
- SKIP_BUILD_RPATH
- SOURCES
- SOVERSION
- STATIC_LIBRARY_FLAGS
- STATIC_LIBRARY_FLAGS_<CONFIG>
- SUFFIX
- TYPE
- VERSION
- VISIBILITY_INLINES_HIDDEN
- VS_DOTNET_REFERENCES
- VS_DOTNET_TARGET_FRAMEWORK_VERSION
- VS_GLOBAL_<variable>
- VS_GLOBAL_KEYWORD
- VS_GLOBAL_PROJECT_TYPES
- VS_GLOBAL_ROOTNAMESPACE
- VS_KEYWORD
- VS_SCC_AUXPATH
- VS_SCC_LOCALPATH
- VS_SCC_PROJECTNAME
- VS_SCC_PROVIDER
- VS_WINRT_EXTENSIONS
- VS_WINRT_REFERENCES
- WIN32_EXECUTABLE
- XCODE_ATTRIBUTE_<an-attribute>

- **<CONFIG>_OUTPUT_NAME**: Old per-configuration target file base name.

  This is a configuration-specific version of OUTPUT_NAME. Use OUTPUT_NAME_<CONFIG> instead.

- **<CONFIG>_POSTFIX**: Postfix to append to the target file name for configuration <CONFIG>.

When building with configuration <CONFIG> the value of this property is appended to the target file name built on disk. For non-executable targets, this property is initialized by the value of the variable CMAKE_<CONFIG>_POSTFIX if it is set when a target is created. This property is ignored on the Mac for Frameworks and App Bundles.

- **<LANG>_VISIBILITY_PRESET**: Value for symbol visibility compile flags

  The <LANG>_VISIBILITY_PRESET property determines the value passed in a visibility related compile option, such as -fvisibility= for <LANG>. This property only has an affect for libraries and executables with exports. This property is initialized by the value of the variable CMAKE_<LANG>_VISIBILITY_PRESET if it is set when a target is created.

- **ALIASED_TARGET**: Name of target aliased by this target.

  If this is an ALIAS target, this property contains the name of the target aliased.

- **ARCHIVE_OUTPUT_DIRECTORY**: Output directory in which to build ARCHIVE target files.

  This property specifies the directory into which archive target files should be built. Multi-configuration generators (VS, Xcode) append a per-configuration subdirectory to the specified directory. There are three kinds of target files that may be built: archive, library, and runtime. Executables are always treated as runtime targets. 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. 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. This property is initialized by the value of the variable CMAKE_ARCHIVE_OUTPUT_DIRECTORY if it is set when a target is created.

- **ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>**: Per-configuration output directory for ARCHIVE target files.

  This is a per-configuration version of ARCHIVE_OUTPUT_DIRECTORY, but multi-configuration generators (VS, Xcode) do NOT append a per-configuration subdirectory to the specified directory. This property is initialized by the value of the variable CMAKE_ARCHIVE_OUTPUT_DIRECTORY_<CONFIG> if it is set when a target is created.

- **ARCHIVE_OUTPUT_NAME**: Output name for ARCHIVE target files.

  This property specifies the base name for archive target files. It overrides OUTPUT_NAME and OUTPUT_NAME_<CONFIG> properties. There are three kinds of target files that may be built: archive, library, and runtime. Executables are always treated as runtime targets. 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. 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.

- **ARCHIVE_OUTPUT_NAME_<CONFIG>**: Per-configuration output name for ARCHIVE target files.

  This is the configuration-specific version of ARCHIVE_OUTPUT_NAME.

- **AUTOMOC**: Should the target be processed with automoc (for Qt projects).

  AUTOMOC is a boolean specifying whether CMake will handle the Qt moc preprocessor automatically, i.e. without having to use the QT4_WRAP_CPP() or QT5_WRAP_CPP() macro. Currently Qt4 and Qt5 are supported. When this property is set to TRUE, CMake will scan the source files at build time and invoke moc accordingly. If an #include statement like #include "moc_foo.cpp" is found, the Q_OBJECT class declaration is expected in the header, and moc is run on the header file. If an #include statement like #include "foo.moc" is found, then a Q_OBJECT is expected in the current source file and moc is run on the file itself. Additionally, all header files are parsed for Q_OBJECT macros, and if found, moc is also executed on those files. The resulting moc files, which are not included as shown above in any of the source files are included in a generated <targetname>_automoc.cpp file, which is compiled as part of the target.This property is initialized by the value of the variable CMAKE_AUTOMOC if it is set when a target is created.

  Additional command line options for moc can be set via the AUTOMOC_MOC_OPTIONS property.

  By setting the CMAKE_AUTOMOC_RELAXED_MODE variable to TRUE the rules for searching the files which will be processed by moc can be relaxed. See the documentation for this variable for more details.

  The global property AUTOMOC_TARGETS_FOLDER can be used to group the automoc targets together in an IDE, e.g. in MSVS.

- **AUTOMOC_MOC_OPTIONS**: Additional options for moc when using automoc (see the AUTOMOC property)

  This property is only used if the AUTOMOC property is set to TRUE for this target. In this case, it holds additional command line options which will be used when moc is executed during the build, i.e. it is equivalent to the optional OPTIONS argument of the qt4_wrap_cpp() macro.

  By default it is empty.

- **BUILD_WITH_INSTALL_RPATH**: Should build tree targets have install tree rpaths.

  BUILD_WITH_INSTALL_RPATH is a boolean specifying whether to link the target in the build tree with the INSTALL_RPATH. This takes precedence over SKIP_BUILD_RPATH and avoids the need for relinking before installation. This property is initialized by the value of the variable CMAKE_BUILD_WITH_INSTALL_RPATH if it is set when a target is created.

- **BUNDLE**: This target is a CFBundle on the Mac.

  If a module library target has this property set to true it will be built as a CFBundle when built on the mac. It will have the directory structure required for a CFBundle and will be suitable to be used for creating Browser Plugins or other application resources.

- **`BUNDLE_EXTENSION`**: The file extension used to name a BUNDLE target on the Mac.

  The default value is "bundle" - you can also use "plugin" or whatever file extension is required by the host app for your bundle.

- **`COMPATIBLE_INTERFACE_BOOL`**: Properties which must be compatible with their link interface

  The COMPATIBLE_INTERFACE_BOOL property may contain a list of propertiesfor this target which must be consistent when evaluated as a boolean in the INTERFACE of all linked dependees. For example, if a property "FOO" appears in the list, then for each dependee, the "INTERFACE_FOO" property content in all of its dependencies must be consistent with each other, and with the "FOO" property in the dependee. Consistency in this sense has the meaning that if the property is set, then it must have the same boolean value as all others, and if the property is not set, then it is ignored. Note that for each dependee, the set of properties from this property must not intersect with the set of properties from the COMPATIBLE_INTERFACE_STRING property.

- **`COMPATIBLE_INTERFACE_STRING`**: Properties which must be string-compatible with their link interface

  The COMPATIBLE_INTERFACE_STRING property may contain a list of properties for this target which must be the same when evaluated as a string in the INTERFACE of all linked dependees. For example, if a property "FOO" appears in the list, then for each dependee, the "INTERFACE_FOO" property content in all of its dependencies must be equal with each other, and with the "FOO" property in the dependee. If the property is not set, then it is ignored. Note that for each dependee, the set of properties from this property must not intersect with the set of properties from the COMPATIBLE_INTERFACE_BOOL property.

- **`COMPILE_DEFINITIONS`**: Preprocessor definitions for compiling a target's sources.

  The COMPILE_DEFINITIONS property may be set to a semicolon-separated list of preprocessor definitions using the syntax VAR or VAR=value. Function-style definitions are not supported. CMake will automatically escape the value correctly for the native build system (note that CMake language syntax may require escapes to specify some values). This property may be set on a per-configuration basis using the name COMPILE_DEFINITIONS_<CONFIG> where <CONFIG> is an upper-case name (ex. "COMPILE_DEFINITIONS_DEBUG").

  CMake will automatically drop some definitions that are not supported by the native build tool. The VS6 IDE does not support definition values with spaces (but NMake does).

  Contents of COMPILE_DEFINITIONS may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
$<0:...>                  = empty string (ignores "...")
$<1:...>                  = content of "..."
$<CONFIG:cfg>             = '1' if config is "cfg", else '0'
$<CONFIGURATION>          = configuration name
$<BOOL:...>               = '1' if the '...' is true, else '0'
$<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
$<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
$<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>          = joins the list with the content of "..."
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

  Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
```

```
  $<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
  $<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
  $<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
  $<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
  $<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

Disclaimer: Most native build tools have poor support for escaping certain values. CMake has work-arounds for many cases but some values may just not be possible to pass correctly. If a value does not seem to be escaped correctly, do not attempt to work-around the problem by adding escape sequences to the value. Your work-around may break in a future version of CMake that has improved escape support. Instead consider defining the macro in a (configured) header file. Then report the limitation. Known limitations include:

```
  #           - broken almost everywhere
  ;           - broken in VS IDE 7.0 and Borland Makefiles
  ,           - broken in VS IDE
  %           - broken in some cases in NMake
  & |         - broken in some cases on MinGW
  ^ < > \"    - broken in most Make tools on Windows
```

CMake does not reject these values outright because they do work in some cases. Use with caution.

- **COMPILE_DEFINITIONS_<CONFIG>**: Per-configuration preprocessor definitions on a target.

  This is the configuration-specific version of COMPILE_DEFINITIONS.

- **COMPILE_FLAGS**: Additional flags to use when compiling this target's sources.

  The COMPILE_FLAGS property sets additional compiler flags used to build sources within the target. Use COMPILE_DEFINITIONS to pass additional preprocessor definitions.

- **COMPILE_OPTIONS**: List of options to pass to the compiler.

  This property specifies the list of options specified so far for this property. This property exists on directories and targets.

  The target property values are used by the generators to set the options for the compiler.

  Contents of COMPILE_OPTIONS may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
  $<0:...>                  = empty string (ignores "...")
  $<1:...>                  = content of "..."
  $<CONFIG:cfg>             = '1' if config is "cfg", else '0'
  $<CONFIGURATION>          = configuration name
  $<BOOL:...>               = '1' if the '...' is true, else '0'
  $<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
  $<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
  $<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
  $<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
  $<JOIN:list,...>          = joins the list with the content of "..."
  $<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
  $<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
  $<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
  $<C_COMPILER_ID>          = The CMake-id of the C compiler used.
  $<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
  $<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
  $<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
  $<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
  $<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
  $<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
  $<C_COMPILER_VERSION>     = The version of the C compiler used.
  $<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
  $<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
  $<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
  $<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>



$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **`DEBUG_POSTFIX`**: See target property <CONFIG>_POSTFIX.

  This property is a special case of the more-general <CONFIG>_POSTFIX property for the DEBUG configuration.

- **`DEFINE_SYMBOL`**: Define a symbol when compiling this target's sources.

  DEFINE_SYMBOL sets the name of the preprocessor symbol defined when compiling sources in a shared library. If not set here then it is set to target_EXPORTS by default (with some substitutions if the target is not a valid C identifier). This is useful for headers to know whether they are being included from inside their library or outside to properly setup dllexport/dllimport decorations.

- **`ENABLE_EXPORTS`**: Specify whether an executable exports symbols for loadable modules.

  Normally an executable does not export any symbols because it is the final program. It is possible for an executable to export symbols to be used by loadable modules. When this property is set to true CMake will allow other targets to "link" to the executable with the TARGET_LINK_LIBRARIES command. On all platforms a target-level dependency on the executable is created for targets that link to it. For DLL platforms an import library will be created for the exported symbols and then used for linking. All Windows-based systems including Cygwin are DLL platforms. For non-DLL platforms that require all symbols to be resolved at link time, such as Mac OS X, the module will "link" to the executable using a flag like "-bundle_loader". For other non-DLL platforms the link rule is simply ignored since the dynamic loader will automatically bind symbols when the module is loaded.

- **`EXCLUDE_FROM_ALL`**: Exclude the target from the all target.

  A property on a target that indicates if the target is excluded from the default build target. If it is not, then with a Makefile for example typing make will cause this target to be built. The same concept applies to the default build of other generators. Installing a target with EXCLUDE_FROM_ALL set to true has undefined behavior.

- **`EXCLUDE_FROM_DEFAULT_BUILD`**: Exclude target from "Build Solution".

  This property is only used by Visual Studio generators 7 and above. When set to TRUE, the target will not be built when you press "Build Solution".

- **`EXCLUDE_FROM_DEFAULT_BUILD_<CONFIG>`**: Per-configuration version of target exclusion from "Build Solution".

  This is the configuration-specific version of EXCLUDE_FROM_DEFAULT_BUILD. If the generic EXCLUDE_FROM_DEFAULT_BUILD is also set on a target, EXCLUDE_FROM_DEFAULT_BUILD_<CONFIG> takes precedence in configurations for which it has a value.

- **`EXPORT_NAME`**: Exported name for target files.

  This sets the name for the IMPORTED target generated when it this target is is exported. If not set, the logical target name is used by default.

- **`EchoString`**: A message to be displayed when the target is built.

  A message to display on some generators (such as makefiles) when the target is built.

- **`FOLDER`**: Set the folder name. Use to organize targets in an IDE.

  Targets with no FOLDER property will appear as top level entities in IDEs like Visual Studio. Targets with the same FOLDER property value will appear next to each other in a folder of that name. To nest folders, use FOLDER values such as 'GUI/Dialogs' with '/' characters separating folder levels.

- **`FRAMEWORK`**: This target is a framework on the Mac.

  If a shared library target has this property set to true it will be built as a framework when built on the mac. It will have the directory structure required for a framework and will be suitable to be used with the -framework option

- **`Fortran_FORMAT`**: Set to FIXED or FREE to indicate the Fortran source layout.

This property tells CMake whether the Fortran source files in a target use fixed-format or free-format. CMake will pass the corresponding format flag to the compiler. Use the source-specific Fortran_FORMAT property to change the format of a specific source file. If the variable CMAKE_Fortran_FORMAT is set when a target is created its value is used to initialize this property.

- **Fortran_MODULE_DIRECTORY**: Specify output directory for Fortran modules provided by the target.

  If the target contains Fortran source files that provide modules and the compiler supports a module output directory this specifies the directory in which the modules will be placed. When this property is not set the modules will be placed in the build directory corresponding to the target's source directory. If the variable CMAKE_Fortran_MODULE_DIRECTORY is set when a target is created its value is used to initialize this property.

  Note that some compilers will automatically search the module output directory for modules USEd during compilation but others will not. If your sources USE modules their location must be specified by INCLUDE_DIRECTORIES regardless of this property.

- **GENERATOR_FILE_NAME**: Generator's file for this target.

  An internal property used by some generators to record the name of the project or dsp file associated with this target. Note that at configure time, this property is only set for targets created by include_external_msproject().

- **GNUtoMS**: Convert GNU import library (.dll.a) to MS format (.lib).

  When linking a shared library or executable that exports symbols using GNU tools on Windows (MinGW/MSYS) with Visual Studio installed convert the import library (.dll.a) from GNU to MS format (.lib). Both import libraries will be installed by install(TARGETS) and exported by install(EXPORT) and export() to be linked by applications with either GNU- or MS-compatible tools.

  If the variable CMAKE_GNUtoMS is set when a target is created its value is used to initialize this property. The variable must be set prior to the first command that enables a language such as project() or enable_language(). CMake provides the variable as an option to the user automatically when configuring on Windows with GNU tools.

- **HAS_CXX**: Link the target using the C++ linker tool (obsolete).

  This is equivalent to setting the LINKER_LANGUAGE property to CXX. See that property's documentation for details.

- **IMPLICIT_DEPENDS_INCLUDE_TRANSFORM**: Specify #include line transforms for dependencies in a target.

  This property specifies rules to transform macro-like #include lines during implicit dependency scanning of C and C++ source files. The list of rules must be semicolon-separated with each entry of the form "A_MACRO(%)=value-with-%" (the % must be literal). During dependency scanning occurrences of A_MACRO(...) on #include lines will be replaced by the value given with the macro argument substituted for '%'. For example, the entry

  ```
  MYDIR(%)=<mydir/%>
  ```

  will convert lines of the form

  ```
  #include MYDIR(myheader.h)
  ```

  to

  ```
  #include <mydir/myheader.h>
  ```

  allowing the dependency to be followed.

  This property applies to sources in the target on which it is set.

- **IMPORTED**: Read-only indication of whether a target is IMPORTED.

  The boolean value of this property is true for targets created with the IMPORTED option to add_executable or add_library. It is false for targets built within the project.

- **IMPORTED_CONFIGURATIONS**: Configurations provided for an IMPORTED target.

  Set this to the list of configuration names available for an IMPORTED target. The names correspond to configurations defined in the project from which the target is imported. If the importing project uses a different set of configurations the names may be mapped using the MAP_IMPORTED_CONFIG_<CONFIG> property. Ignored for non-imported targets.

- **IMPORTED_IMPLIB**: Full path to the import library for an IMPORTED target.

  Set this to the location of the ".lib" part of a windows DLL. Ignored for non-imported targets.

- **IMPORTED_IMPLIB_<CONFIG>**: <CONFIG>-specific version of IMPORTED_IMPLIB property.

  Configuration names correspond to those provided by the project from which the target is imported.

- **IMPORTED_LINK_DEPENDENT_LIBRARIES**: Dependent shared libraries of an imported shared library.

  Shared libraries may be linked to other shared libraries as part of their implementation. On some platforms the linker searches for the dependent libraries of shared libraries they are including in the link. Set this property to the list of dependent shared libraries of an imported library. The list should be disjoint from the list of interface libraries in the INTERFACE_LINK_LIBRARIES property. On platforms requiring dependent shared libraries to be found at link time CMake uses this list to add appropriate files or paths to the link command line. Ignored for non-imported targets.

- **IMPORTED_LINK_DEPENDENT_LIBRARIES_<CONFIG>**: <CONFIG>-specific version of IMPORTED_LINK_DEPENDENT_LIBRARIES.

Configuration names correspond to those provided by the project from which the target is imported. If set, this property completely overrides the generic property for the named configuration.

- **IMPORTED_LINK_INTERFACE_LANGUAGES**: Languages compiled into an IMPORTED static library.

  Set this to the list of languages of source files compiled to produce a STATIC IMPORTED library (such as "C" or "CXX"). CMake accounts for these languages when computing how to link a target to the imported library. For example, when a C executable links to an imported C++ static library CMake chooses the C++ linker to satisfy language runtime dependencies of the static library.

  This property is ignored for targets that are not STATIC libraries. This property is ignored for non-imported targets.

- **IMPORTED_LINK_INTERFACE_LANGUAGES_<CONFIG>**: <CONFIG>-specific version of IMPORTED_LINK_INTERFACE_LANGUAGES.

  Configuration names correspond to those provided by the project from which the target is imported. If set, this property completely overrides the generic property for the named configuration.

- **IMPORTED_LINK_INTERFACE_LIBRARIES**: Transitive link interface of an IMPORTED target.

  Set this to the list of libraries whose interface is included when an IMPORTED library target is linked to another target. The libraries will be included on the link line for the target. Unlike the LINK_INTERFACE_LIBRARIES property, this property applies to all imported target types, including STATIC libraries. This property is ignored for non-imported targets.

  This property is ignored if the target also has a non-empty INTERFACE_LINK_LIBRARIES property.

  This property is deprecated. Use INTERFACE_LINK_LIBRARIES instead.

- **IMPORTED_LINK_INTERFACE_LIBRARIES_<CONFIG>**: <CONFIG>-specific version of IMPORTED_LINK_INTERFACE_LIBRARIES.

  Configuration names correspond to those provided by the project from which the target is imported. If set, this property completely overrides the generic property for the named configuration.

  This property is ignored if the target also has a non-empty INTERFACE_LINK_LIBRARIES property.

  This property is deprecated. Use INTERFACE_LINK_LIBRARIES instead.

- **IMPORTED_LINK_INTERFACE_MULTIPLICITY**: Repetition count for cycles of IMPORTED static libraries.

  This is LINK_INTERFACE_MULTIPLICITY for IMPORTED targets.

- **IMPORTED_LINK_INTERFACE_MULTIPLICITY_<CONFIG>**: <CONFIG>-specific version of IMPORTED_LINK_INTERFACE_MULTIPLICITY.

  If set, this property completely overrides the generic property for the named configuration.

- **IMPORTED_LOCATION**: Full path to the main file on disk for an IMPORTED target.

  Set this to the location of an IMPORTED target file on disk. For executables this is the location of the executable file. For bundles on OS X this is the location of the executable file inside Contents/MacOS under the application bundle folder. For static libraries and modules this is the location of the library or module. For shared libraries on non-DLL platforms this is the location of the shared library. For frameworks on OS X this is the location of the library file symlink just inside the framework folder. For DLLs this is the location of the ".dll" part of the library. For UNKNOWN libraries this is the location of the file to be linked. Ignored for non-imported targets.

  Projects may skip IMPORTED_LOCATION if the configuration-specific property IMPORTED_LOCATION_<CONFIG> is set. To get the location of an imported target read one of the LOCATION or LOCATION_<CONFIG> properties.

- **IMPORTED_LOCATION_<CONFIG>**: <CONFIG>-specific version of IMPORTED_LOCATION property.

  Configuration names correspond to those provided by the project from which the target is imported.

- **IMPORTED_NO_SONAME**: Specifies that an IMPORTED shared library target has no "soname".

  Set this property to true for an imported shared library file that has no "soname" field. CMake may adjust generated link commands for some platforms to prevent the linker from using the path to the library in place of its missing soname. Ignored for non-imported targets.

- **IMPORTED_NO_SONAME_<CONFIG>**: <CONFIG>-specific version of IMPORTED_NO_SONAME property.

  Configuration names correspond to those provided by the project from which the target is imported.

- **IMPORTED_SONAME**: The "soname" of an IMPORTED target of shared library type.

  Set this to the "soname" embedded in an imported shared library. This is meaningful only on platforms supporting the feature. Ignored for non-imported targets.

- **IMPORTED_SONAME_<CONFIG>**: <CONFIG>-specific version of IMPORTED_SONAME property.

  Configuration names correspond to those provided by the project from which the target is imported.

- **IMPORT_PREFIX**: What comes before the import library name.

  Similar to the target property PREFIX, but used for import libraries (typically corresponding to a DLL) instead of regular libraries. A target property that can be set to override the prefix (such as "lib") on an import library name.

- **IMPORT_SUFFIX**: What comes after the import library name.

Similar to the target property SUFFIX, but used for import libraries (typically corresponding to a DLL) instead of regular libraries. A target property that can be set to override the suffix (such as ".lib") on an import library name.

- **INCLUDE_DIRECTORIES**: List of preprocessor include file search directories.

  This property specifies the list of directories given so far to the include_directories command. This property exists on directories and targets. In addition to accepting values from the include_directories command, values may be set directly on any directory or any target using the set_property command. A target gets its initial value for this property from the value of the directory property. A directory gets its initial value from its parent directory if it has one. Both directory and target property values are adjusted by calls to the include_directories command.

  The target property values are used by the generators to set the include paths for the compiler. See also the include_directories command.

  Contents of INCLUDE_DIRECTORIES may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
$<0:...>                  = empty string (ignores "...")
$<1:...>                  = content of "..."
$<CONFIG:cfg>             = '1' if config is "cfg", else '0'
$<CONFIGURATION>          = configuration name
$<BOOL:...>               = '1' if the '...' is true, else '0'
$<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
$<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
$<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>          = joins the list with the content of "..."
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

  Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

  Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

  where '?' is always either '0' or '1'.

  Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **INSTALL_NAME_DIR**: Mac OSX directory name for installed targets.

  INSTALL_NAME_DIR is a string specifying the directory portion of the "install_name" field of shared libraries on Mac OSX to use in the installed targets.

- **INSTALL_RPATH**: The rpath to use for installed targets.

A semicolon-separated list specifying the rpath to use in installed targets (for platforms that support it). This property is initialized by the value of the variable CMAKE_INSTALL_RPATH if it is set when a target is created.

- **`INSTALL_RPATH_USE_LINK_PATH`**: Add paths to linker search and installed rpath.

  INSTALL_RPATH_USE_LINK_PATH is a boolean that if set to true will append directories in the linker search path and outside the project to the INSTALL_RPATH. This property is initialized by the value of the variable CMAKE_INSTALL_RPATH_USE_LINK_PATH if it is set when a target is created.

- **`INTERFACE_COMPILE_DEFINITIONS`**: List of public compile definitions for a library.

  Targets may populate this property to publish the compile definitions required to compile against the headers for the target. Consuming targets can add entries to their own COMPILE_DEFINITIONS property such as $<TARGET_PROPERTY:foo,INTERFACE_COMPILE_DEFINITIONS> to use the compile definitions specified in the interface of 'foo'.

  Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
$<0:...>                  = empty string (ignores "...")
$<1:...>                  = content of "..."
$<CONFIG:cfg>             = '1' if config is "cfg", else '0'
$<CONFIGURATION>          = configuration name
$<BOOL:...>               = '1' if the '...' is true, else '0'
$<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
$<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
$<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>          = joins the list with the content of "..."
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

  Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

  Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

  where '?' is always either '0' or '1'.

  Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **`INTERFACE_COMPILE_OPTIONS`**: List of interface options to pass to the compiler.

  Targets may populate this property to publish the compile options required to compile against the headers for the target.
  Consuming targets can add entries to their own COMPILE_OPTIONS property such as

$<TARGET_PROPERTY:foo,INTERFACE_COMPILE_OPTIONS> to use the compile options specified in the interface of 'foo'.

Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
$<0:...>                  = empty string (ignores "...")
$<1:...>                  = content of "..."
$<CONFIG:cfg>             = '1' if config is "cfg", else '0'
$<CONFIGURATION>          = configuration name
$<BOOL:...>               = '1' if the '...' is true, else '0'
$<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
$<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
$<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>          = joins the list with the content of "..."
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>


$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **`INTERFACE_INCLUDE_DIRECTORIES`**: List of public include directories for a library.

  Targets may populate this property to publish the include directories required to compile against the headers for the target. Consuming targets can add entries to their own INCLUDE_DIRECTORIES property such as $<TARGET_PROPERTY:foo,INTERFACE_INCLUDE_DIRECTORIES> to use the include directories specified in the interface of 'foo'.

  Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

  ```
  $<0:...>                  = empty string (ignores "...")
  $<1:...>                  = content of "..."
  $<CONFIG:cfg>             = '1' if config is "cfg", else '0'
  $<CONFIGURATION>          = configuration name
  $<BOOL:...>               = '1' if the '...' is true, else '0'
  $<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
  $<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
  ```

```
$<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>          = joins the list with the content of "..."
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **`INTERFACE_LINK_LIBRARIES`**: List public interface libraries for a shared library or executable.

  This property contains the list of transitive link dependencies. When the target is linked into another target the libraries listed (and recursively their link interface libraries) will be provided to the other target also. This property is overriden by the LINK_INTERFACE_LIBRARIES or LINK_INTERFACE_LIBRARIES_<CONFIG> property if policy CMP0022 is OLD or unset.

  Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
$<0:...>                  = empty string (ignores "...")
$<1:...>                  = content of "..."
$<CONFIG:cfg>             = '1' if config is "cfg", else '0'
$<CONFIGURATION>          = configuration name
$<BOOL:...>               = '1' if the '...' is true, else '0'
$<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
$<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
$<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>          = joins the list with the content of "..."
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
```

```
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **`INTERFACE_POSITION_INDEPENDENT_CODE`**: Whether consumers need to create a position-independent target

  The INTERFACE_POSITION_INDEPENDENT_CODE property informs consumers of this target whether they must set their POSITION_INDEPENDENT_CODE property to ON. If this property is set to ON, then the POSITION_INDEPENDENT_CODE property on all consumers will be set to ON. Similarly, if this property is set to OFF, then the POSITION_INDEPENDENT_CODE property on all consumers will be set to OFF. If this property is undefined, then consumers will determine their POSITION_INDEPENDENT_CODE property by other means. Consumers must ensure that the targets that they link to have a consistent requirement for their INTERFACE_POSITION_INDEPENDENT_CODE property.

- **`INTERFACE_SYSTEM_INCLUDE_DIRECTORIES`**: List of public system include directories for a library.

  Targets may populate this property to publish the include directories which contain system headers, and therefore should not result in compiler warnings. Consuming targets will then mark the same include directories as system headers.

  Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

  ```
  $<0:...>                  = empty string (ignores "...")
  $<1:...>                  = content of "..."
  $<CONFIG:cfg>             = '1' if config is "cfg", else '0'
  $<CONFIGURATION>          = configuration name
  $<BOOL:...>               = '1' if the '...' is true, else '0'
  $<STREQUAL:a,b>           = '1' if a is STREQUAL b, else '0'
  $<ANGLE-R>                = A literal '>'. Used to compare strings which contain a '>' for example.
  $<COMMA>                  = A literal ','. Used to compare strings which contain a ',' for example.
  $<SEMICOLON>              = A literal ';'. Used to prevent list expansion on an argument with ';'.
  $<JOIN:list,...>          = joins the list with the content of "..."
  $<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
  $<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
  $<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
  $<C_COMPILER_ID>          = The CMake-id of the C compiler used.
  $<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
  $<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
  $<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
  $<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
  $<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
  $<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
  $<C_COMPILER_VERSION>     = The version of the C compiler used.
  $<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
  $<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
  ```

```
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>    = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>           = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>          = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>            = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>             = '0' if all '?' are '0', else '1'
$<NOT:?>                   = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>    = The value of the property prop on the target on which the generator expression is evaluated.
```

- **`INTERPROCEDURAL_OPTIMIZATION`**: Enable interprocedural optimization for a target.

  If set to true, enables interprocedural optimizations if they are known to be supported by the compiler.

- **`INTERPROCEDURAL_OPTIMIZATION_<CONFIG>`**: Per-configuration interprocedural optimization for a target.

  This is a per-configuration version of INTERPROCEDURAL_OPTIMIZATION. If set, this property overrides the generic property for the named configuration.

- **`LABELS`**: Specify a list of text labels associated with a target.

  Target label semantics are currently unspecified.

- **`LIBRARY_OUTPUT_DIRECTORY`**: Output directory in which to build LIBRARY target files.

  This property specifies the directory into which library target files should be built. Multi-configuration generators (VS, Xcode) append a per-configuration subdirectory to the specified directory. There are three kinds of target files that may be built: archive, library, and runtime. Executables are always treated as runtime targets. 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. 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. This property is initialized by the value of the variable CMAKE_LIBRARY_OUTPUT_DIRECTORY if it is set when a target is created.

- **`LIBRARY_OUTPUT_DIRECTORY_<CONFIG>`**: Per-configuration output directory for LIBRARY target files.

  This is a per-configuration version of LIBRARY_OUTPUT_DIRECTORY, but multi-configuration generators (VS, Xcode) do NOT append a per-configuration subdirectory to the specified directory. This property is initialized by the value of the variable CMAKE_LIBRARY_OUTPUT_DIRECTORY_<CONFIG> if it is set when a target is created.

- **`LIBRARY_OUTPUT_NAME`**: Output name for LIBRARY target files.

  This property specifies the base name for library target files. It overrides OUTPUT_NAME and OUTPUT_NAME_<CONFIG> properties. There are three kinds of target files that may be built: archive, library, and runtime. Executables are always treated as runtime targets. 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. 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.

- **`LIBRARY_OUTPUT_NAME_<CONFIG>`**: Per-configuration output name for LIBRARY target files.

  This is the configuration-specific version of LIBRARY_OUTPUT_NAME.

- **`LINKER_LANGUAGE`**: Specifies language whose compiler will invoke the linker.

  For executables, shared libraries, and modules, this sets the language whose compiler is used to link the target (such as "C" or "CXX"). A typical value for an executable is the language of the source file providing the program entry point (main). If not set, the language with the highest linker preference value is the default. See documentation of CMAKE_<LANG>_LINKER_PREFERENCE variables.

  If this property is not set by the user, it will be calculated at generate-time by CMake.

- **LINK_DEPENDS**: Additional files on which a target binary depends for linking.

  Specifies a semicolon-separated list of full-paths to files on which the link rule for this target depends. The target binary will be linked if any of the named files is newer than it.

  This property is ignored by non-Makefile generators. It is intended to specify dependencies on "linker scripts" for custom Makefile link rules.

- **LINK_DEPENDS_NO_SHARED**: Do not depend on linked shared library files.

  Set this property to true to tell CMake generators not to add file-level dependencies on the shared library files linked by this target. Modification to the shared libraries will not be sufficient to re-link this target. Logical target-level dependencies will not be affected so the linked shared libraries will still be brought up to date before this target is built.

  This property is initialized by the value of the variable CMAKE_LINK_DEPENDS_NO_SHARED if it is set when a target is created.

- **LINK_FLAGS**: Additional flags to use when linking this target.

  The LINK_FLAGS property can be used to add extra flags to the link step of a target. LINK_FLAGS_<CONFIG> will add to the configuration <CONFIG>, for example, DEBUG, RELEASE, MINSIZEREL, RELWITHDEBINFO.

- **LINK_FLAGS_<CONFIG>**: Per-configuration linker flags for a target.

  This is the configuration-specific version of LINK_FLAGS.

- **LINK_INTERFACE_LIBRARIES**: List public interface libraries for a shared library or executable.

  By default linking to a shared library target transitively links to targets with which the library itself was linked. For an executable with exports (see the ENABLE_EXPORTS property) no default transitive link dependencies are used. This property replaces the default transitive link dependencies with an explicit list. When the target is linked into another target the libraries listed (and recursively their link interface libraries) will be provided to the other target also. If the list is empty then no transitive link dependencies will be incorporated when this target is linked into another target even if the default set is non-empty. This property is initialized by the value of the variable CMAKE_LINK_INTERFACE_LIBRARIES if it is set when a target is created. This property is ignored for STATIC libraries.

  This property is overriden by the INTERFACE_LINK_LIBRARIES property if policy CMP0022 is NEW.

  This property is deprecated. Use INTERFACE_LINK_LIBRARIES instead.

- **LINK_INTERFACE_LIBRARIES_<CONFIG>**: Per-configuration list of public interface libraries for a target.

  This is the configuration-specific version of LINK_INTERFACE_LIBRARIES. If set, this property completely overrides the generic property for the named configuration.

  This property is overriden by the INTERFACE_LINK_LIBRARIES property if policy CMP0022 is NEW.

  This property is deprecated. Use INTERFACE_LINK_LIBRARIES instead.

- **LINK_INTERFACE_MULTIPLICITY**: Repetition count for STATIC libraries with cyclic dependencies.

  When linking to a STATIC library target with cyclic dependencies the linker may need to scan more than once through the archives in the strongly connected component of the dependency graph. CMake by default constructs the link line so that the linker will scan through the component at least twice. This property specifies the minimum number of scans if it is larger than the default. CMake uses the largest value specified by any target in a component.

- **LINK_INTERFACE_MULTIPLICITY_<CONFIG>**: Per-configuration repetition count for cycles of STATIC libraries.

  This is the configuration-specific version of LINK_INTERFACE_MULTIPLICITY. If set, this property completely overrides the generic property for the named configuration.

- **LINK_LIBRARIES**: List of direct link dependencies.

  This property specifies the list of libraries or targets which will be used for linking. In addition to accepting values from the target_link_libraries command, values may be set directly on any target using the set_property command.

The target property values are used by the generators to set the link libraries for the compiler. See also the target_link_libraries command.

Contents of LINK_LIBRARIES may use "generator expressions" with the syntax "$<...>". Generator expressions are evaluated during build system generation to produce information specific to each build configuration. Valid expressions are:

```
$<0:...>                = empty string (ignores "...")
$<1:...>                = content of "..."
$<CONFIG:cfg>           = '1' if config is "cfg", else '0'
$<CONFIGURATION>        = configuration name
$<BOOL:...>             = '1' if the '...' is true, else '0'
$<STREQUAL:a,b>         = '1' if a is STREQUAL b, else '0'
$<ANGLE-R>              = A literal '>'. Used to compare strings which contain a '>' for example.
$<COMMA>                = A literal ','. Used to compare strings which contain a ',' for example.
$<SEMICOLON>            = A literal ';'. Used to prevent list expansion on an argument with ';'.
$<JOIN:list,...>        = joins the list with the content of "..."
```

```
$<TARGET_NAME:...>        = Marks ... as being the name of a target.  This is required if exporting targets to multiple dependent expor
$<INSTALL_INTERFACE:...>  = content of "..." when the property is exported using install(EXPORT), and empty otherwise.
$<BUILD_INTERFACE:...>    = content of "..." when the property is exported using export(), or when the target is used by another target
$<C_COMPILER_ID>          = The CMake-id of the C compiler used.
$<C_COMPILER_ID:comp>     = '1' if the CMake-id of the C compiler matches comp, otherwise '0'.
$<CXX_COMPILER_ID>        = The CMake-id of the CXX compiler used.
$<CXX_COMPILER_ID:comp>   = '1' if the CMake-id of the CXX compiler matches comp, otherwise '0'.
$<VERSION_GREATER:v1,v2>  = '1' if v1 is a version greater than v2, else '0'.
$<VERSION_LESS:v1,v2>     = '1' if v1 is a version less than v2, else '0'.
$<VERSION_EQUAL:v1,v2>    = '1' if v1 is the same version as v2, else '0'.
$<C_COMPILER_VERSION>     = The version of the C compiler used.
$<C_COMPILER_VERSION:ver> = '1' if the version of the C compiler matches ver, otherwise '0'.
$<CXX_COMPILER_VERSION>   = The version of the CXX compiler used.
$<CXX_COMPILER_VERSION:ver> = '1' if the version of the CXX compiler matches ver, otherwise '0'.
$<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>
```

```
$<TARGET_PROPERTY:tgt,prop>   = The value of the property prop on the target tgt.
```

Note that tgt is not added as a dependency of the target this expression is evaluated on.

```
$<TARGET_POLICY:pol>          = '1' if the policy was NEW when the 'head' target was created, else '0'.  If the policy was not set, the
$<INSTALL_PREFIX>         = Content of the install prefix when the target is exported via INSTALL(EXPORT) and empty otherwise.
```

Boolean expressions:

```
$<AND:?[,?]...>           = '1' if all '?' are '1', else '0'
$<OR:?[,?]...>            = '0' if all '?' are '0', else '1'
$<NOT:?>                  = '0' if '?' is '1', else '1'
```

where '?' is always either '0' or '1'.

Expressions with an implicit 'this' target:

```
$<TARGET_PROPERTY:prop>   = The value of the property prop on the target on which the generator expression is evaluated.
```

- **`LINK_SEARCH_END_STATIC`**: End a link line such that static system libraries are used.

  Some linkers support switches such as -Bstatic and -Bdynamic to determine whether to use static or shared libraries for -lXXX options. CMake uses these options to set the link type for libraries whose full paths are not known or (in some cases) are in implicit link directories for the platform. By default CMake adds an option at the end of the library list (if necessary) to set the linker search type back to its starting type. This property switches the final linker search type to -Bstatic regardless of how it started. See also LINK_SEARCH_START_STATIC.

- **`LINK_SEARCH_START_STATIC`**: Assume the linker looks for static libraries by default.

  Some linkers support switches such as -Bstatic and -Bdynamic to determine whether to use static or shared libraries for -lXXX options. CMake uses these options to set the link type for libraries whose full paths are not known or (in some cases) are in implicit link directories for the platform. By default the linker search type is assumed to be -Bdynamic at the beginning of the library list. This property switches the assumption to -Bstatic. It is intended for use when linking an executable statically (e.g. with the GNU -static option). See also LINK_SEARCH_END_STATIC.

- **`LOCATION`**: Read-only location of a target on disk.

  For an imported target, this read-only property returns the value of the LOCATION_<CONFIG> property for an unspecified configuration <CONFIG> provided by the target.

  For a non-imported target, this property is provided for compatibility with CMake 2.4 and below. It was meant to get the location of an executable target's output file for use in add_custom_command. The path may contain a build-system-specific portion that is replaced at build time with the configuration getting built (such as "$(ConfigurationName)" in VS). In CMake 2.6 and above add_custom_command automatically recognizes a target name in its COMMAND and DEPENDS options and computes the target location. In CMake 2.8.4 and above add_custom_command recognizes generator expressions to refer to target locations anywhere in the command. Therefore this property is not needed for creating custom commands.

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

- **`LOCATION_<CONFIG>`**: Read-only property providing a target location on disk.

A read-only property that indicates where a target's main file is located on disk for the configuration <CONFIG>. The property is defined only for library and executable targets. An imported target may provide a set of configurations different from that of the importing project. By default CMake looks for an exact-match but otherwise uses an arbitrary available configuration. Use the MAP_IMPORTED_CONFIG_<CONFIG> property to map imported configurations explicitly.

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

- **MACOSX_BUNDLE**: Build an executable as an application bundle on Mac OS X.

  When this property is set to true the executable when built on Mac OS X will be created as an application bundle. This makes it a GUI executable that can be launched from the Finder. See the MACOSX_BUNDLE_INFO_PLIST target property for information about creation of the Info.plist file for the application bundle. This property is initialized by the value of the variable CMAKE_MACOSX_BUNDLE if it is set when a target is created.

- **MACOSX_BUNDLE_INFO_PLIST**: Specify a custom Info.plist template for a Mac OS X App Bundle.

  An executable target with MACOSX_BUNDLE enabled will be built as an application bundle on Mac OS X. By default its Info.plist file is created by configuring a template called MacOSXBundleInfo.plist.in located in the CMAKE_MODULE_PATH. This property specifies an alternative template file name which may be a full path.

  The following target properties may be set to specify content to be configured into the file:

  ```
  MACOSX_BUNDLE_INFO_STRING
  MACOSX_BUNDLE_ICON_FILE
  MACOSX_BUNDLE_GUI_IDENTIFIER
  MACOSX_BUNDLE_LONG_VERSION_STRING
  MACOSX_BUNDLE_BUNDLE_NAME
  MACOSX_BUNDLE_SHORT_VERSION_STRING
  MACOSX_BUNDLE_BUNDLE_VERSION
  MACOSX_BUNDLE_COPYRIGHT
  ```

  CMake variables of the same name may be set to affect all targets in a directory that do not have each specific property set. If a custom Info.plist is specified by this property it may of course hard-code all the settings instead of using the target properties.

- **MACOSX_FRAMEWORK_INFO_PLIST**: Specify a custom Info.plist template for a Mac OS X Framework.

  A library target with FRAMEWORK enabled will be built as a framework on Mac OS X. By default its Info.plist file is created by configuring a template called MacOSXFrameworkInfo.plist.in located in the CMAKE_MODULE_PATH. This property specifies an alternative template file name which may be a full path.

  The following target properties may be set to specify content to be configured into the file:

  ```
  MACOSX_FRAMEWORK_ICON_FILE
  MACOSX_FRAMEWORK_IDENTIFIER
  MACOSX_FRAMEWORK_SHORT_VERSION_STRING
  MACOSX_FRAMEWORK_BUNDLE_VERSION
  ```

  CMake variables of the same name may be set to affect all targets in a directory that do not have each specific property set. If a custom Info.plist is specified by this property it may of course hard-code all the settings instead of using the target properties.

- **MACOSX_RPATH**: Whether to use rpaths on Mac OS X.

  When this property is set to true, the directory portion of the"install_name" field of shared libraries will default to "@rpath".Runtime paths will also be embedded in binaries using this target.This property is initialized by the value of the variable CMAKE_MACOSX_RPATH if it is set when a target is created.

- **MAP_IMPORTED_CONFIG_<CONFIG>**: Map from project configuration to IMPORTED target's configuration.

  Set this to the list of configurations of an imported target that may be used for the current project's <CONFIG> configuration. Targets imported from another project may not provide the same set of configuration names available in the current project. Setting this property tells CMake what imported configurations are suitable for use when building the <CONFIG> configuration. The first configuration in the list found to be provided by the imported target is selected. If this property is set and no matching configurations are available, then the imported target is considered to be not found. This property is ignored for non-imported targets.

- **NAME**: Logical name for the target.

  Read-only logical name for the target as used by CMake.

- **NO_SONAME**: Whether to set "soname" when linking a shared library or module.

  Enable this boolean property if a generated shared library or module should not have "soname" set. Default is to set "soname" on all shared libraries and modules as long as the platform supports it. Generally, use this property only for leaf private libraries or plugins. If you use it on normal shared libraries which other targets link against, on some platforms a linker will insert a full path to the library (as specified at link time) into the dynamic section of the dependent binary. Therefore, once installed, dynamic loader may eventually fail to locate the library for the binary.

- **OSX_ARCHITECTURES**: Target specific architectures for OS X.

  The OSX_ARCHITECTURES property sets the target binary architecture for targets on OS X. This property is initialized by the value of the variable CMAKE_OSX_ARCHITECTURES if it is set when a target is created. Use OSX_ARCHITECTURES_<CONFIG> to set the binary architectures on a per-configuration basis. <CONFIG> is an upper-case name (ex: "OSX_ARCHITECTURES_DEBUG").

- **OSX_ARCHITECTURES_<CONFIG>**: Per-configuration OS X binary architectures for a target.

  This property is the configuration-specific version of OSX_ARCHITECTURES.

- **OUTPUT_NAME**: Output name for target files.

  This sets the base name for output files created for an executable or library target. If not set, the logical target name is used by default.

- **OUTPUT_NAME_<CONFIG>**: Per-configuration target file base name.

  This is the configuration-specific version of OUTPUT_NAME.

- **PDB_NAME**: Output name for MS debug symbols .pdb file from linker.

  Set the base name for debug symbols file created for an executable or shared library target. If not set, the logical target name is used by default.

  This property is not implemented by the Visual Studio 6 generator.

- **PDB_NAME_<CONFIG>**: Per-configuration name for MS debug symbols .pdb file.

  This is the configuration-specific version of PDB_NAME.

  This property is not implemented by the Visual Studio 6 generator.

- **PDB_OUTPUT_DIRECTORY**: Output directory for MS debug symbols .pdb file from linker.

  This property specifies the directory into which the MS debug symbols will be placed by the linker. This property is initialized by the value of the variable CMAKE_PDB_OUTPUT_DIRECTORY if it is set when a target is created.

  This property is not implemented by the Visual Studio 6 generator.

- **PDB_OUTPUT_DIRECTORY_<CONFIG>**: Per-configuration output directory for MS debug symbols .pdb files.

  This is a per-configuration version of PDB_OUTPUT_DIRECTORY, but multi-configuration generators (VS, Xcode) do NOT append a per-configuration subdirectory to the specified directory. This property is initialized by the value of the variable CMAKE_PDB_OUTPUT_DIRECTORY_<CONFIG> if it is set when a target is created.

  This property is not implemented by the Visual Studio 6 generator.

- **POSITION_INDEPENDENT_CODE**: Whether to create a position-independent target

  The POSITION_INDEPENDENT_CODE property determines whether position independent executables or shared libraries will be created. This property is true by default for SHARED and MODULE library targets and false otherwise. This property is initialized by the value of the variable CMAKE_POSITION_INDEPENDENT_CODE if it is set when a target is created.

- **POST_INSTALL_SCRIPT**: Deprecated install support.

  The PRE_INSTALL_SCRIPT and POST_INSTALL_SCRIPT properties are the old way to specify CMake scripts to run before and after installing a target. They are used only when the old INSTALL_TARGETS command is used to install the target. Use the INSTALL command instead.

- **PREFIX**: What comes before the library name.

  A target property that can be set to override the prefix (such as "lib") on a library name.

- **PRE_INSTALL_SCRIPT**: Deprecated install support.

  The PRE_INSTALL_SCRIPT and POST_INSTALL_SCRIPT properties are the old way to specify CMake scripts to run before and after installing a target. They are used only when the old INSTALL_TARGETS command is used to install the target. Use the INSTALL command instead.

- **PRIVATE_HEADER**: Specify private header files in a FRAMEWORK shared library target.

  Shared library targets marked with the FRAMEWORK property generate frameworks on OS X and normal shared libraries on other platforms. This property may be set to a list of header files to be placed in the PrivateHeaders directory inside the framework folder. On non-Apple platforms these headers may be installed using the PRIVATE_HEADER option to the install(TARGETS) command.

- **PROJECT_LABEL**: Change the name of a target in an IDE.

  Can be used to change the name of the target in an IDE like Visual Studio.

- **PUBLIC_HEADER**: Specify public header files in a FRAMEWORK shared library target.

  Shared library targets marked with the FRAMEWORK property generate frameworks on OS X and normal shared libraries on other platforms. This property may be set to a list of header files to be placed in the Headers directory inside the framework folder. On non-Apple platforms these headers may be installed using the PUBLIC_HEADER option to the install(TARGETS)

command.

- **RESOURCE**: Specify resource files in a FRAMEWORK shared library target.

  Shared library targets marked with the FRAMEWORK property generate frameworks on OS X and normal shared libraries on other platforms. This property may be set to a list of files to be placed in the Resources directory inside the framework folder. On non-Apple platforms these files may be installed using the RESOURCE option to the install(TARGETS) command.

- **RULE_LAUNCH_COMPILE**: Specify a launcher for compile rules.

  See the global property of the same name for details. This overrides the global and directory property for a target.

- **RULE_LAUNCH_CUSTOM**: Specify a launcher for custom rules.

  See the global property of the same name for details. This overrides the global and directory property for a target.

- **RULE_LAUNCH_LINK**: Specify a launcher for link rules.

  See the global property of the same name for details. This overrides the global and directory property for a target.

- **RUNTIME_OUTPUT_DIRECTORY**: Output directory in which to build RUNTIME target files.

  This property specifies the directory into which runtime target files should be built. Multi-configuration generators (VS, Xcode) append a per-configuration subdirectory to the specified directory. There are three kinds of target files that may be built: archive, library, and runtime. Executables are always treated as runtime targets. 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. 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. This property is initialized by the value of the variable CMAKE_RUNTIME_OUTPUT_DIRECTORY if it is set when a target is created.

- **RUNTIME_OUTPUT_DIRECTORY_<CONFIG>**: Per-configuration output directory for RUNTIME target files.

  This is a per-configuration version of RUNTIME_OUTPUT_DIRECTORY, but multi-configuration generators (VS, Xcode) do NOT append a per-configuration subdirectory to the specified directory. This property is initialized by the value of the variable CMAKE_RUNTIME_OUTPUT_DIRECTORY_<CONFIG> if it is set when a target is created.

- **RUNTIME_OUTPUT_NAME**: Output name for RUNTIME target files.

  This property specifies the base name for runtime target files. It overrides OUTPUT_NAME and OUTPUT_NAME_<CONFIG> properties. There are three kinds of target files that may be built: archive, library, and runtime. Executables are always treated as runtime targets. 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. 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.

- **RUNTIME_OUTPUT_NAME_<CONFIG>**: Per-configuration output name for RUNTIME target files.

  This is the configuration-specific version of RUNTIME_OUTPUT_NAME.

- **SKIP_BUILD_RPATH**: Should rpaths be used for the build tree.

  SKIP_BUILD_RPATH is a boolean specifying whether to skip automatic generation of an rpath allowing the target to run from the build tree. This property is initialized by the value of the variable CMAKE_SKIP_BUILD_RPATH if it is set when a target is created.

- **SOURCES**: Source names specified for a target.

  Read-only list of sources specified for a target. The names returned are suitable for passing to the set_source_files_properties command.

- **SOVERSION**: What version number is this target.

  For shared libraries VERSION and SOVERSION can be used to specify the build version and API version respectively. When building or installing appropriate symlinks are created if the platform supports symlinks and the linker supports so-names. If only one of both is specified the missing is assumed to have the same version number. SOVERSION is ignored if NO_SONAME property is set. For shared libraries and executables on Windows the VERSION attribute is parsed to extract a "major.minor" version number. These numbers are used as the image version of the binary.

- **STATIC_LIBRARY_FLAGS**: Extra flags to use when linking static libraries.

  Extra flags to use when linking a static library.

- **STATIC_LIBRARY_FLAGS_<CONFIG>**: Per-configuration flags for creating a static library.

  This is the configuration-specific version of STATIC_LIBRARY_FLAGS.

- **SUFFIX**: What comes after the target name.

  A target property that can be set to override the suffix (such as ".so" or ".exe") on the name of a library, module or executable.

- **TYPE**: The type of the target.

  This read-only property can be used to test the type of the given target. It will be one of STATIC_LIBRARY, MODULE_LIBRARY,

SHARED_LIBRARY, EXECUTABLE or one of the internal target types.

- **VERSION**: What version number is this target.

  For shared libraries VERSION and SOVERSION can be used to specify the build version and API version respectively. When building or installing appropriate symlinks are created if the platform supports symlinks and the linker supports so-names. If only one of both is specified the missing is assumed to have the same version number. For executables VERSION can be used to specify the build version. When building or installing appropriate symlinks are created if the platform supports symlinks. For shared libraries and executables on Windows the VERSION attribute is parsed to extract a "major.minor" version number. These numbers are used as the image version of the binary.

- **VISIBILITY_INLINES_HIDDEN**: Whether to add a compile flag to hide symbols of inline functions

  The VISIBILITY_INLINES_HIDDEN property determines whether a flag for hiding symbols for inline functions. the value passed used in a visibility related compile option, such as -fvisibility=. This property only has an affect for libraries and executables with exports. This property is initialized by the value of the variable CMAKE_VISIBILITY_INLINES_HIDDEN if it is set when a target is created.

- **VS_DOTNET_REFERENCES**: Visual Studio managed project .NET references

  Adds one or more semicolon-delimited .NET references to a generated Visual Studio project. For example, "System;System.Windows.Forms".

- **VS_DOTNET_TARGET_FRAMEWORK_VERSION**: Specify the .NET target framework version.

  Used to specify the .NET target framework version for C++/CLI. For example, "v4.5".

- **VS_GLOBAL_<variable>**: Visual Studio project-specific global variable.

  Tell the Visual Studio generator to set the global variable '<variable>' to a given value in the generated Visual Studio project. Ignored on other generators. Qt integration works better if VS_GLOBAL_QtVersion is set to the version FindQt4.cmake found. For example, "4.7.3"

- **VS_GLOBAL_KEYWORD**: Visual Studio project keyword.

  Sets the "keyword" attribute for a generated Visual Studio project. Defaults to "Win32Proj". You may wish to override this value with "ManagedCProj", for example, in a Visual Studio managed C++ unit test project.

- **VS_GLOBAL_PROJECT_TYPES**: Visual Studio project type(s).

  Can be set to one or more UUIDs recognized by Visual Studio to indicate the type of project. This value is copied verbatim into the generated project file. Example for a managed C++ unit testing project:

  ```
  {3AC096D0-A1C2-E12C-1390-A8335801FDAB};{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}
  ```

  UUIDs are semicolon-delimited.

- **VS_GLOBAL_ROOTNAMESPACE**: Visual Studio project root namespace.

  Sets the "RootNamespace" attribute for a generated Visual Studio project. The attribute will be generated only if this is set.

- **VS_KEYWORD**: Visual Studio project keyword.

  Can be set to change the visual studio keyword, for example Qt integration works better if this is set to Qt4VSv1.0.

- **VS_SCC_AUXPATH**: Visual Studio Source Code Control Aux Path.

  Can be set to change the visual studio source code control auxpath property.

- **VS_SCC_LOCALPATH**: Visual Studio Source Code Control Local Path.

  Can be set to change the visual studio source code control local path property.

- **VS_SCC_PROJECTNAME**: Visual Studio Source Code Control Project.

  Can be set to change the visual studio source code control project name property.

- **VS_SCC_PROVIDER**: Visual Studio Source Code Control Provider.

  Can be set to change the visual studio source code control provider property.

- **VS_WINRT_EXTENSIONS**: Visual Studio project C++/CX language extensions for Windows Runtime

  Can be set to enable C++/CX language extensions.

- **VS_WINRT_REFERENCES**: Visual Studio project Windows Runtime Metadata references

  Adds one or more semicolon-delimited WinRT references to a generated Visual Studio project. For example, "Windows;Windows.UI.Core".

- **WIN32_EXECUTABLE**: Build an executable with a WinMain entry point on windows.

  When this property is set to true the executable when linked on Windows will be created with a WinMain() entry point instead of just main(). This makes it a GUI executable instead of a console application. See the CMAKE_MFC_FLAG variable documentation to configure use of MFC for WinMain executables. This property is initialized by the value of the variable CMAKE_WIN32_EXECUTABLE if it is set when a target is created.

- **XCODE_ATTRIBUTE_<an-attribute>**: Set Xcode target attributes directly.

  Tell the Xcode generator to set '<an-attribute>' to a given value in the generated Xcode project. Ignored on other generators.

Properties on Tests

- ATTACHED_FILES
- ATTACHED_FILES_ON_FAIL
- COST
- DEPENDS
- ENVIRONMENT
- FAIL_REGULAR_EXPRESSION
- LABELS
- MEASUREMENT
- PASS_REGULAR_EXPRESSION
- PROCESSORS
- REQUIRED_FILES
- RESOURCE_LOCK
- RUN_SERIAL
- TIMEOUT
- WILL_FAIL
- WORKING_DIRECTORY

- **ATTACHED_FILES**: Attach a list of files to a dashboard submission.

  Set this property to a list of files that will be encoded and submitted to the dashboard as an addition to the test result.

- **ATTACHED_FILES_ON_FAIL**: Attach a list of files to a dashboard submission if the test fails.

  Same as ATTACHED_FILES, but these files will only be included if the test does not pass.

- **COST**: Set this to a floating point value. Tests in a test set will be run in descending order of cost.

  This property describes the cost of a test. You can explicitly set this value; tests with higher COST values will run first.

- **DEPENDS**: Specifies that this test should only be run after the specified list of tests.

  Set this to a list of tests that must finish before this test is run.

- **ENVIRONMENT**: Specify environment variables that should be defined for running a test.

  If set to a list of environment variables and values of the form MYVAR=value those environment variables will be defined while running the test. The environment is restored to its previous state after the test is done.

- **FAIL_REGULAR_EXPRESSION**: If the output matches this regular expression the test will fail.

  If set, if the output matches one of specified regular expressions, the test will fail.For example: FAIL_REGULAR_EXPRESSION "[^a-z]Error;ERROR;Failed"

- **LABELS**: Specify a list of text labels associated with a test.

  The list is reported in dashboard submissions.

- **MEASUREMENT**: Specify a CDASH measurement and value to be reported for a test.

  If set to a name then that name will be reported to CDASH as a named measurement with a value of 1. You may also specify a value by setting MEASUREMENT to "measurement=value".

- **PASS_REGULAR_EXPRESSION**: The output must match this regular expression for the test to pass.

  If set, the test output will be checked against the specified regular expressions and at least one of the regular expressions has to match, otherwise the test will fail.

- **PROCESSORS**: How many process slots this test requires

  Denotes the number of processors that this test will require. This is typically used for MPI tests, and should be used in conjunction with the ctest_test PARALLEL_LEVEL option.

- **REQUIRED_FILES**: List of files required to run the test.

  If set to a list of files, the test will not be run unless all of the files exist.

- **RESOURCE_LOCK**: Specify a list of resources that are locked by this test.

  If multiple tests specify the same resource lock, they are guaranteed not to run concurrently.

- **RUN_SERIAL**: Do not run this test in parallel with any other test.

  Use this option in conjunction with the ctest_test PARALLEL_LEVEL option to specify that this test should not be run in parallel with any other tests.

- **TIMEOUT**: How many seconds to allow for this test.

  This property if set will limit a test to not take more than the specified number of seconds to run. If it exceeds that the test

process will be killed and ctest will move to the next test. This setting takes precedence over CTEST_TESTING_TIMEOUT.

- **WILL_FAIL**: If set to true, this will invert the pass/fail flag of the test.

  This property can be used for tests that are expected to fail and return a non zero return code.

- **WORKING_DIRECTORY**: The directory from which the test executable will be called.

  If this is not set it is called from the directory the test executable is located in.

Properties on Source Files

- ABSTRACT
- COMPILE_DEFINITIONS
- COMPILE_DEFINITIONS_<CONFIG>
- COMPILE_FLAGS
- EXTERNAL_OBJECT
- Fortran_FORMAT
- GENERATED
- HEADER_FILE_ONLY
- KEEP_EXTENSION
- LABELS
- LANGUAGE
- LOCATION
- MACOSX_PACKAGE_LOCATION
- OBJECT_DEPENDS
- OBJECT_OUTPUTS
- SYMBOLIC
- WRAP_EXCLUDE

- **ABSTRACT**: Is this source file an abstract class.

  A property on a source file that indicates if the source file represents a class that is abstract. This only makes sense for languages that have a notion of an abstract class and it is only used by some tools that wrap classes into other languages.

- **COMPILE_DEFINITIONS**: Preprocessor definitions for compiling a source file.

  The COMPILE_DEFINITIONS property may be set to a semicolon-separated list of preprocessor definitions using the syntax VAR or VAR=value. Function-style definitions are not supported. CMake will automatically escape the value correctly for the native build system (note that CMake language syntax may require escapes to specify some values). This property may be set on a per-configuration basis using the name COMPILE_DEFINITIONS_<CONFIG> where <CONFIG> is an upper-case name (ex. "COMPILE_DEFINITIONS_DEBUG").

  CMake will automatically drop some definitions that are not supported by the native build tool. The VS6 IDE does not support definition values with spaces (but NMake does). Xcode does not support per-configuration definitions on source files.

  Disclaimer: Most native build tools have poor support for escaping certain values. CMake has work-arounds for many cases but some values may just not be possible to pass correctly. If a value does not seem to be escaped correctly, do not attempt to work-around the problem by adding escape sequences to the value. Your work-around may break in a future version of CMake that has improved escape support. Instead consider defining the macro in a (configured) header file. Then report the limitation. Known limitations include:

  ```
  #          - broken almost everywhere
  ;          - broken in VS IDE 7.0 and Borland Makefiles
  ,          - broken in VS IDE
  %          - broken in some cases in NMake
  & |        - broken in some cases on MinGW
  ^ < > \"   - broken in most Make tools on Windows
  ```

  CMake does not reject these values outright because they do work in some cases. Use with caution.

- **COMPILE_DEFINITIONS_<CONFIG>**: Per-configuration preprocessor definitions on a source file.

  This is the configuration-specific version of COMPILE_DEFINITIONS. Note that Xcode does not support per-configuration source file flags so this property will be ignored by the Xcode generator.

- **COMPILE_FLAGS**: Additional flags to be added when compiling this source file.

  These flags will be added to the list of compile flags when this source file builds. Use COMPILE_DEFINITIONS to pass additional preprocessor definitions.

- **EXTERNAL_OBJECT**: If set to true then this is an object file.

  If this property is set to true then the source file is really an object file and should not be compiled. It will still be linked into the target though.

- **Fortran_FORMAT**: Set to FIXED or FREE to indicate the Fortran source layout.

  This property tells CMake whether a given Fortran source file uses fixed-format or free-format. CMake will pass the corresponding format flag to the compiler. Consider using the target-wide Fortran_FORMAT property if all source files in a

target share the same format.

- **GENERATED**: Is this source file generated as part of the build process.

  If a source file is generated by the build process CMake will handle it differently in terms of dependency checking etc. Otherwise having a non-existent source file could create problems.

- **HEADER_FILE_ONLY**: Is this source file only a header file.

  A property on a source file that indicates if the source file is a header file with no associated implementation. This is set automatically based on the file extension and is used by CMake to determine if certain dependency information should be computed.

- **KEEP_EXTENSION**: Make the output file have the same extension as the source file.

  If this property is set then the file extension of the output file will be the same as that of the source file. Normally the output file extension is computed based on the language of the source file, for example .cxx will go to a .o extension.

- **LABELS**: Specify a list of text labels associated with a source file.

  This property has meaning only when the source file is listed in a target whose LABELS property is also set. No other semantics are currently specified.

- **LANGUAGE**: What programming language is the file.

  A property that can be set to indicate what programming language the source file is. If it is not set the language is determined based on the file extension. Typical values are CXX C etc. Setting this property for a file means this file will be compiled. Do not set this for headers or files that should not be compiled.

- **LOCATION**: The full path to a source file.

  A read only property on a SOURCE FILE that contains the full path to the source file.

- **MACOSX_PACKAGE_LOCATION**: Place a source file inside a Mac OS X bundle, CFBundle, or framework.

  Executable targets with the MACOSX_BUNDLE property set are built as Mac OS X application bundles on Apple platforms. Shared library targets with the FRAMEWORK property set are built as Mac OS X frameworks on Apple platforms. Module library targets with the BUNDLE property set are built as Mac OS X CFBundle bundles on Apple platforms. Source files listed in the target with this property set will be copied to a directory inside the bundle or framework content folder specified by the property value. For bundles the content folder is "<name>.app/Contents". For frameworks the content folder is "<name>.framework/Versions/<version>". For cfbundles the content folder is "<name>.bundle/Contents" (unless the extension is changed). See the PUBLIC_HEADER, PRIVATE_HEADER, and RESOURCE target properties for specifying files meant for Headers, PrivateHeaders, or Resources directories.

- **OBJECT_DEPENDS**: Additional files on which a compiled object file depends.

  Specifies a semicolon-separated list of full-paths to files on which any object files compiled from this source file depend. An object file will be recompiled if any of the named files is newer than it.

  This property need not be used to specify the dependency of a source file on a generated header file that it includes. Although the property was originally introduced for this purpose, it is no longer necessary. If the generated header file is created by a custom command in the same target as the source file, the automatic dependency scanning process will recognize the dependency. If the generated header file is created by another target, an inter-target dependency should be created with the add_dependencies command (if one does not already exist due to linking relationships).

- **OBJECT_OUTPUTS**: Additional outputs for a Makefile rule.

  Additional outputs created by compilation of this source file. If any of these outputs is missing the object will be recompiled. This is supported only on Makefile generators and will be ignored on other generators.

- **SYMBOLIC**: Is this just a name for a rule.

  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.

- **WRAP_EXCLUDE**: Exclude this source file from any code wrapping techniques.

  Some packages can wrap source files into alternate languages to provide additional functionality. For example, C++ code can be wrapped into Java or Python etc using SWIG etc. If WRAP_EXCLUDE is set to true (1 etc) that indicates that this source file should not be wrapped.

Properties on Cache Entries

- **ADVANCED**
- **HELPSTRING**
- **MODIFIED**
- **STRINGS**
- **TYPE**
- **VALUE**

- **ADVANCED**: True if entry should be hidden by default in GUIs.

  This is a boolean value indicating whether the entry is considered interesting only for advanced configuration. The

mark_as_advanced() command modifies this property.

- **HELPSTRING**: Help associated with entry in GUIs.

  This string summarizes the purpose of an entry to help users set it through a CMake GUI.

- **MODIFIED**: Internal management property. Do not set or get.

  This is an internal cache entry property managed by CMake to track interactive user modification of entries. Ignore it.

- **STRINGS**: Enumerate possible STRING entry values for GUI selection.

  For cache entries with type STRING, this enumerates a set of values. CMake GUIs may use this to provide a selection widget instead of a generic string entry field. This is for convenience only. CMake does not enforce that the value matches one of those listed.

- **TYPE**: Widget type for entry in GUIs.

  Cache entry values are always strings, but CMake GUIs present widgets to help users set values. The GUIs use this property as a hint to determine the widget type. Valid TYPE values are:

  ```
  BOOL          = Boolean ON/OFF value.
  PATH          = Path to a directory.
  FILEPATH      = Path to a file.
  STRING        = Generic string value.
  INTERNAL      = Do not present in GUI at all.
  STATIC        = Value managed by CMake, do not change.
  UNINITIALIZED = Type not yet specified.
  ```

  Generally the TYPE of a cache entry should be set by the command which creates it (set, option, find_library, etc.).

- **VALUE**: Value of a cache entry.

  This property maps to the actual value of a cache entry. Setting this property always sets the value without checking, so use with care.

Compatibility Commands

- **build_name**
- **exec_program**
- **export_library_dependencies**
- **install_files**
- **install_programs**
- **install_targets**
- **link_libraries**
- **make_directory**
- **output_required_files**
- **remove**
- **subdir_depends**
- **subdirs**
- **use_mangled_mesa**
- **utility_source**
- **variable_requires**
- **write_file**

```
CMake Compatibility Listfile Commands - Obsolete commands supported by CMake for compatibility.
```

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, faster and shinier new 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.

- **output_required_files**: Deprecated. Approximate C preprocessor dependency scanning.

  This command exists only because ancient CMake versions provided it. CMake handles preprocessor dependency scanning automatically using a more advanced scanner.

  ```
  output_required_files(srcfile outputfile)
  ```

  Outputs a list of all the source files that are required by the specified srcfile. This list is written into outputfile. This is similar to writing out the dependencies for srcfile except that it jumps from .h files into .cxx, .c and .cpp files if possible.

- **remove**: Deprecated. Use the list(REMOVE_ITEM) command instead.

  ```
  remove(VAR VALUE VALUE ...)
  ```

  Removes VALUE from the variable VAR. This is typically used to remove entries from a vector (e.g. semicolon separated list). VALUE is expanded.

- **subdir_depends**: 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 exclude_dir2 ...]
          [PREORDER] )
  ```

  Add a list of subdirectories to the build. The add_subdirectory command should be used instead of subdirs although subdirs will still work. This will cause any CMakeLists.txt files in the sub directories to be processed by CMake. Any directories after the PREORDER flag are traversed first by makefile builds, the PREORDER flag has no effect on IDE projects. Any directories after the EXCLUDE_FROM_ALL marker will not be included in the top level makefile or project file. This is useful for having CMake create makefiles or projects for a set of examples in a project. You would want CMake to generate makefiles or project files for all the examples at the same time, but you would not want them to show up in the top level project or be built each time make is run from the top.

- **use_mangled_mesa**: 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.

Standard CMake Modules

- AddFileDependencies
- BundleUtilities
- CMakeAddFortranSubdirectory
- CMakeBackwardCompatibilityCXX
- CMakeDependentOption
- CMakeDetermineVSServicePack
- CMakeExpandImportedTargets
- CMakeFindFrameworks
- CMakeFindPackageMode
- CMakeForceCompiler
- CMakeGraphVizOptions
- CMakePackageConfigHelpers
- CMakeParseArguments
- CMakePrintHelpers
- CMakePrintSystemInformation
- CMakePushCheckState
- CMakeVerifyManifest
- CPack
- CPackBundle
- CPackComponent
- CPackCygwin
- CPackDMG
- CPackDeb
- CPackNSIS
- CPackPackageMaker
- CPackRPM
- CPackWIX
- CTest
- CTestScriptMode
- CTestUseLaunchers
- CheckCCompilerFlag
- CheckCSourceCompiles
- CheckCSourceRuns
- CheckCXXCompilerFlag
- CheckCXXSourceCompiles
- CheckCXXSourceRuns
- CheckCXXSymbolExists
- CheckFortranFunctionExists
- CheckFunctionExists
- CheckIncludeFile
- CheckIncludeFileCXX
- CheckIncludeFiles
- CheckLanguage
- CheckLibraryExists
- CheckPrototypeDefinition
- CheckStructHasMember
- CheckSymbolExists
- CheckTypeSize
- CheckVariableExists
- Dart
- DeployQt4
- Documentation
- ExternalData
- ExternalProject
- FeatureSummary
- FindALSA
- FindASPELL
- FindAVIFile
- FindArmadillo
- FindBISON
- FindBLAS
- FindBZip2
- FindBoost
- FindBullet
- FindCABLE
- FindCUDA
- FindCURL
- FindCVS

- FindCoin3D
- FindCups
- FindCurses
- FindCxxTest
- FindCygwin
- FindDCMTK
- FindDart
- FindDevIL
- FindDoxygen
- FindEXPAT
- FindFLEX
- FindFLTK
- FindFLTK2
- FindFreetype
- FindGCCXML
- FindGDAL
- FindGIF
- FindGLEW
- FindGLUT
- FindGTK
- FindGTK2
- FindGTest
- FindGettext
- FindGit
- FindGnuTLS
- FindGnuplot
- FindHDF5
- FindHSPELL
- FindHTMLHelp
- FindHg
- FindITK
- FindIcotool
- FindImageMagick
- FindJNI
- FindJPEG
- FindJasper
- FindJava
- FindKDE3
- FindKDE4
- FindLAPACK
- FindLATEX
- FindLibArchive
- FindLibLZMA
- FindLibXml2
- FindLibXslt
- FindLua50
- FindLua51
- FindMFC
- FindMPEG
- FindMPEG2
- FindMPI
- FindMatlab
- FindMotif
- FindOpenAL
- FindOpenGL
- FindOpenMP
- FindOpenSSL
- FindOpenSceneGraph
- FindOpenThreads
- FindPHP4
- FindPNG
- FindPackageHandleStandardArgs
- FindPackageMessage
- FindPerl
- FindPerlLibs
- FindPhysFS
- FindPike
- FindPkgConfig
- FindPostgreSQL
- FindProducer
- FindProtobuf

- FindPythonInterp
- FindPythonLibs
- FindQt
- FindQt3
- FindQt4
- FindQuickTime
- FindRTI
- FindRuby
- FindSDL
- FindSDL_image
- FindSDL_mixer
- FindSDL_net
- FindSDL_sound
- FindSDL_ttf
- FindSWIG
- FindSelfPackers
- FindSquish
- FindSubversion
- FindTCL
- FindTIFF
- FindTclStub
- FindTclsh
- FindThreads
- FindUnixCommands
- FindVTK
- FindWget
- FindWish
- FindX11
- FindXMLRPC
- FindZLIB
- Findosg
- FindosgAnimation
- FindosgDB
- FindosgFX
- FindosgGA
- FindosgIntrospection
- FindosgManipulator
- FindosgParticle
- FindosgPresentation
- FindosgProducer
- FindosgQt
- FindosgShadow
- FindosgSim
- FindosgTerrain
- FindosgText
- FindosgUtil
- FindosgViewer
- FindosgVolume
- FindosgWidget
- Findosg_functions
- FindwxWidgets
- FindwxWindows
- FortranCInterface
- GNUInstallDirs
- GenerateExportHeader
- GetPrerequisites
- InstallRequiredSystemLibraries
- MacroAddFileDependencies
- ProcessorCount
- Qt4ConfigDependentSettings
- Qt4Macros
- SelectLibraryConfigurations
- SquishTestScript
- TestBigEndian
- TestCXXAcceptsFlag
- TestForANSIForScope
- TestForANSIStreamHeaders
- TestForSSTREAM
- TestForSTDNamespace
- UseEcos
- UseJava

- **UseJavaClassFilelist**
- **UseJavaSymlinks**
- **UsePkgConfig**
- **UseQt4**
- **UseSWIG**
- **Use_wxWindows**
- **UsewxWidgets**
- **WriteBasicConfigVersionFile**

The following modules are provided with CMake. They can be used with INCLUDE(ModuleName).

```
CMake Modules - Modules coming with CMake, the Cross-Platform Makefile Generator.
```

This is the documentation for the modules and scripts coming with CMake. Using these modules you can check the computer system for installed software packages, features of the compiler and the existence of headers to name just a few.

- **AddFileDependencies**: ADD_FILE_DEPENDENCIES(source_file depend_files...)

  Adds the given files as dependencies to source_file

- **BundleUtilities**: Functions to help assemble a standalone bundle application.

  A collection of CMake utility functions useful for dealing with .app bundles on the Mac and bundle-like directories on any OS.

  The following functions are provided by this module:

  ```
  fixup_bundle
  copy_and_fixup_bundle
  verify_app
  get_bundle_main_executable
  get_dotapp_dir
  get_bundle_and_executable
  get_bundle_all_executables
  get_item_key
  clear_bundle_keys
  set_bundle_key_values
  get_bundle_keys
  copy_resolved_item_into_bundle
  copy_resolved_framework_into_bundle
  fixup_bundle_item
  verify_bundle_prerequisites
  verify_bundle_symlinks
  ```

  Requires CMake 2.6 or greater because it uses function, break and PARENT_SCOPE. Also depends on GetPrerequisites.cmake.

  ```
  FIXUP_BUNDLE(<app> <libs> <dirs>)
  ```

  Fix up a bundle in-place and make it standalone, such that it can be drag-n-drop copied to another machine and run on that machine as long as all of the system libraries are compatible.

  If you pass plugins to fixup_bundle as the libs parameter, you should install them or copy them into the bundle before calling fixup_bundle. The "libs" parameter is a list of libraries that must be fixed up, but that cannot be determined by otool output analysis. (i.e., plugins)

  Gather all the keys for all the executables and libraries in a bundle, and then, for each key, copy each prerequisite into the bundle. Then fix each one up according to its own list of prerequisites.

  Then clear all the keys and call verify_app on the final bundle to ensure that it is truly standalone.

  ```
  COPY_AND_FIXUP_BUNDLE(<src> <dst> <libs> <dirs>)
  ```

  Makes a copy of the bundle <src> at location <dst> and then fixes up the new copied bundle in-place at <dst>...

  ```
  VERIFY_APP(<app>)
  ```

  Verifies that an application <app> appears valid based on running analysis tools on it. Calls "message(FATAL_ERROR" if the application is not verified.

  ```
  GET_BUNDLE_MAIN_EXECUTABLE(<bundle> <result_var>)
  ```

  The result will be the full path name of the bundle's main executable file or an "error:" prefixed string if it could not be determined.

  ```
  GET_DOTAPP_DIR(<exe> <dotapp_dir_var>)
  ```

  Returns the nearest parent dir whose name ends with ".app" given the full path to an executable. If there is no such parent dir, then simply return the dir containing the executable.

  The returned directory may or may not exist.

  ```
  GET_BUNDLE_AND_EXECUTABLE(<app> <bundle_var> <executable_var> <valid_var>)
  ```

Takes either a ".app" directory name or the name of an executable nested inside a ".app" directory and returns the path to the ".app" directory in <bundle_var> and the path to its main executable in <executable_var>

```
GET_BUNDLE_ALL_EXECUTABLES(<bundle> <exes_var>)
```

Scans the given bundle recursively for all executable files and accumulates them into a variable.

```
GET_ITEM_KEY(<item> <key_var>)
```

Given a file (item) name, generate a key that should be unique considering the set of libraries that need copying or fixing up to make a bundle standalone. This is essentially the file name including extension with "." replaced by "_"

This key is used as a prefix for CMake variables so that we can associate a set of variables with a given item based on its key.

```
CLEAR_BUNDLE_KEYS(<keys_var>)
```

Loop over the list of keys, clearing all the variables associated with each key. After the loop, clear the list of keys itself.

Caller of get_bundle_keys should call clear_bundle_keys when done with list of keys.

```
SET_BUNDLE_KEY_VALUES(<keys_var> <context> <item> <exepath> <dirs>
                      <copyflag>)
```

Add a key to the list (if necessary) for the given item. If added, also set all the variables associated with that key.

```
GET_BUNDLE_KEYS(<app> <libs> <dirs> <keys_var>)
```

Loop over all the executable and library files within the bundle (and given as extra <libs>) and accumulate a list of keys representing them. Set values associated with each key such that we can loop over all of them and copy prerequisite libs into the bundle and then do appropriate install_name_tool fixups.

```
COPY_RESOLVED_ITEM_INTO_BUNDLE(<resolved_item> <resolved_embedded_item>)
```

Copy a resolved item into the bundle if necessary. Copy is not necessary if the resolved_item is "the same as" the resolved_embedded_item.

```
COPY_RESOLVED_FRAMEWORK_INTO_BUNDLE(<resolved_item> <resolved_embedded_item>)
```

Copy a resolved framework into the bundle if necessary. Copy is not necessary if the resolved_item is "the same as" the resolved_embedded_item.

By default, BU_COPY_FULL_FRAMEWORK_CONTENTS is not set. If you want full frameworks embedded in your bundles, set BU_COPY_FULL_FRAMEWORK_CONTENTS to ON before calling fixup_bundle. By default, COPY_RESOLVED_FRAMEWORK_INTO_BUNDLE copies the framework dylib itself plus the framework Resources directory.

```
FIXUP_BUNDLE_ITEM(<resolved_embedded_item> <exepath> <dirs>)
```

Get the direct/non-system prerequisites of the resolved embedded item. For each prerequisite, change the way it is referenced to the value of the _EMBEDDED_ITEM keyed variable for that prerequisite. (Most likely changing to an "@executable_path" style reference.)

This function requires that the resolved_embedded_item be "inside" the bundle already. In other words, if you pass plugins to fixup_bundle as the libs parameter, you should install them or copy them into the bundle before calling fixup_bundle. The "libs" parameter is a list of libraries that must be fixed up, but that cannot be determined by otool output analysis. (i.e., plugins)

Also, change the id of the item being fixed up to its own _EMBEDDED_ITEM value.

Accumulate changes in a local variable and make *one* call to install_name_tool at the end of the function with all the changes at once.

If the BU_CHMOD_BUNDLE_ITEMS variable is set then bundle items will be marked writable before install_name_tool tries to change them.

```
VERIFY_BUNDLE_PREREQUISITES(<bundle> <result_var> <info_var>)
```

Verifies that the sum of all prerequisites of all files inside the bundle are contained within the bundle or are "system" libraries, presumed to exist everywhere.

```
VERIFY_BUNDLE_SYMLINKS(<bundle> <result_var> <info_var>)
```

Verifies that any symlinks found in the bundle point to other files that are already also in the bundle... Anything that points to an external file causes this function to fail the verification.

- **`CMakeAddFortranSubdirectory`**: Use MinGW gfortran from VS if a fortran compiler is not found.

  The 'add_fortran_subdirectory' function adds a subdirectory to a project that contains a fortran only sub-project. The module will check the current compiler and see if it can support fortran. If no fortran compiler is found and the compiler is MSVC, then this module will find the MinGW gfortran. It will then use an external project to build with the MinGW tools. It will also create imported targets for the libraries created. This will only work if the fortran code is built into a dll, so BUILD_SHARED_LIBS is turned on in the project. In addition the CMAKE_GNUtoMS option is set to on, so that the MS .lib files are created. Usage is as follows:

  ```
  cmake_add_fortran_subdirectory(
   <subdir>                # name of subdirectory
  ```

```
  PROJECT <project_name>  # project name in subdir top CMakeLists.txt
  ARCHIVE_DIR <dir>       # dir where project places .lib files
  RUNTIME_DIR <dir>       # dir where project places .dll files
  LIBRARIES <lib>...      # names of library targets to import
  LINK_LIBRARIES          # link interface libraries for LIBRARIES
   [LINK_LIBS <lib> <dep>...]...
  CMAKE_COMMAND_LINE ...  # extra command line flags to pass to cmake
  NO_EXTERNAL_INSTALL     # skip installation of external project
  )
```

Relative paths in ARCHIVE_DIR and RUNTIME_DIR are interpreted with respect to the build directory corresponding to the source directory in which the function is invoked.

Limitations:

NO_EXTERNAL_INSTALL is required for forward compatibility with a future version that supports installation of the external project binaries during "make install".

- **CMakeBackwardCompatibilityCXX**: define a bunch of backwards compatibility variables

```
CMAKE_ANSI_CXXFLAGS - flag for ansi c++
CMAKE_HAS_ANSI_STRING_STREAM - has <strstream>
include(TestForANSIStreamHeaders)
include(CheckIncludeFileCXX)
include(TestForSTDNamespace)
include(TestForANSIForScope)
```

- **CMakeDependentOption**: Macro to provide an option dependent on other options.

  This macro presents an option to the user only if a set of other conditions are true. When the option is not presented a default value is used, but any value set by the user is preserved for when the option is presented again. Example invocation:

```
CMAKE_DEPENDENT_OPTION(USE_FOO "Use Foo" ON
                       "USE_BAR;NOT USE_ZOT" OFF)
```

  If USE_BAR is true and USE_ZOT is false, this provides an option called USE_FOO that defaults to ON. Otherwise, it sets USE_FOO to OFF. If the status of USE_BAR or USE_ZOT ever changes, any value for the USE_FOO option is saved so that when the option is re-enabled it retains its old value.

- **CMakeDetermineVSServicePack**: Determine the Visual Studio service pack of the 'cl' in use.

  The functionality of this module has been superseded by the platform variable CMAKE_<LANG>_COMPILER_VERSION that contains the compiler version number.

  Usage:

```
if(MSVC)
  include(CMakeDetermineVSServicePack)
  DetermineVSServicePack( my_service_pack )
  if( my_service_pack )
    message(STATUS "Detected: ${my_service_pack}")
  endif()
endif()
```

  Function DetermineVSServicePack sets the given variable to one of the following values or an empty string if unknown:

```
vc80, vc80sp1
vc90, vc90sp1
vc100, vc100sp1
vc110, vc110sp1, vc110sp2
```

- **CMakeExpandImportedTargets**:

  CMAKE_EXPAND_IMPORTED_TARGETS(<var> LIBRARIES lib1 lib2...libN

  [CONFIGURATION <config>])

  CMAKE_EXPAND_IMPORTED_TARGETS() takes a list of libraries and replaces all imported targets contained in this list with their actual file paths of the referenced libraries on disk, including the libraries from their link interfaces. If a CONFIGURATION is given, it uses the respective configuration of the imported targets if it exists. If no CONFIGURATION is given, it uses the first configuration from ${CMAKE_CONFIGURATION_TYPES} if set, otherwise ${CMAKE_BUILD_TYPE}. This macro is used by all Check*.cmake files which use try_compile() or try_run() and support CMAKE_REQUIRED_LIBRARIES , so that these checks support imported targets in CMAKE_REQUIRED_LIBRARIES:

```
cmake_expand_imported_targets(expandedLibs LIBRARIES ${CMAKE_REQUIRED_LIBRARIES}
                                           CONFIGURATION "${CMAKE_TRY_COMPILE_CONFIGURATION}" )
```

- **CMakeFindFrameworks**: helper module to find OSX frameworks
- **CMakeFindPackageMode**:

This file is executed by cmake when invoked with --find-package. It expects that the following variables are set using -D:

```
NAME = name of the package
COMPILER_ID = the CMake compiler ID for which the result is, i.e. GNU/Intel/Clang/MSVC, etc.
LANGUAGE = language for which the result will be used, i.e. C/CXX/Fortan/ASM
MODE = EXIST : only check for existence of the given package
       COMPILE : print the flags needed for compiling an object file which uses the given package
       LINK : print the flags needed for linking when using the given package
QUIET = if TRUE, don't print anything
```

- **CMakeForceCompiler**:

  This module defines macros intended for use by cross-compiling toolchain files when CMake is not able to automatically detect the compiler identification.

  Macro CMAKE_FORCE_C_COMPILER has the following signature:

  ```
  CMAKE_FORCE_C_COMPILER(<compiler> <compiler-id>)
  ```

  It sets CMAKE_C_COMPILER to the given compiler and the cmake internal variable CMAKE_C_COMPILER_ID to the given compiler-id. It also bypasses the check for working compiler and basic compiler information tests.

  Macro CMAKE_FORCE_CXX_COMPILER has the following signature:

  ```
  CMAKE_FORCE_CXX_COMPILER(<compiler> <compiler-id>)
  ```

  It sets CMAKE_CXX_COMPILER to the given compiler and the cmake internal variable CMAKE_CXX_COMPILER_ID to the given compiler-id. It also bypasses the check for working compiler and basic compiler information tests.

  Macro CMAKE_FORCE_Fortran_COMPILER has the following signature:

  ```
  CMAKE_FORCE_Fortran_COMPILER(<compiler> <compiler-id>)
  ```

  It sets CMAKE_Fortran_COMPILER to the given compiler and the cmake internal variable CMAKE_Fortran_COMPILER_ID to the given compiler-id. It also bypasses the check for working compiler and basic compiler information tests.

  So a simple toolchain file could look like this:

  ```
  include (CMakeForceCompiler)
  set(CMAKE_SYSTEM_NAME Generic)
  CMAKE_FORCE_C_COMPILER   (chc12 MetrowerksHicross)
  CMAKE_FORCE_CXX_COMPILER (chc12 MetrowerksHicross)
  ```

- **CMakeGraphVizOptions**: The builtin graphviz support of CMake.

  CMake can generate graphviz files, showing the dependencies between the targets in a project and also external libraries which are linked against. When CMake is run with the --graphiz=foo option, it will produce

  ```
  * a foo.dot file showing all dependencies in the project
  * a foo.dot.<target> file for each target, file showing on which other targets the respective target depends
  * a foo.dot.<target>.dependers file, showing which other targets depend on the respective target
  ```

  This can result in huge graphs. Using the file CMakeGraphVizOptions.cmake the look and content of the generated graphs can be influenced. This file is searched first in ${CMAKE_BINARY_DIR} and then in ${CMAKE_SOURCE_DIR}. If found, it is read and the variables set in it are used to adjust options for the generated graphviz files.

  ```
  GRAPHVIZ_GRAPH_TYPE - The graph type
     Mandatory : NO
     Default   : "digraph"
  GRAPHVIZ_GRAPH_NAME - The graph name.
     Mandatory : NO
     Default   : "GG"
  GRAPHVIZ_GRAPH_HEADER - The header written at the top of the graphviz file.
     Mandatory : NO
     Default   : "node [n  fontsize = "12"];"
  GRAPHVIZ_NODE_PREFIX - The prefix for each node in the graphviz file.
     Mandatory : NO
     Default   : "node"
  GRAPHVIZ_EXECUTABLES - Set this to FALSE to exclude executables from the generated graphs.
     Mandatory : NO
     Default   : TRUE
  GRAPHVIZ_STATIC_LIBS - Set this to FALSE to exclude static libraries from the generated graphs.
     Mandatory : NO
     Default   : TRUE
  GRAPHVIZ_SHARED_LIBS - Set this to FALSE to exclude shared libraries from the generated graphs.
     Mandatory : NO
     Default   : TRUE
  GRAPHVIZ_MODULE_LIBS - Set this to FALSE to exclude static libraries from the generated graphs.
  ```

```
   Mandatory : NO
   Default   : TRUE
GRAPHVIZ_EXTERNAL_LIBS - Set this to FALSE to exclude external libraries from the generated graphs.
   Mandatory : NO
   Default   : TRUE
GRAPHVIZ_IGNORE_TARGETS - A list of regular expressions for ignoring targets.
   Mandatory : NO
   Default   : empty
```

- **CMakePackageConfigHelpers**: CONFIGURE_PACKAGE_CONFIG_FILE(), WRITE_BASIC_PACKAGE_VERSION_FILE()

```
CONFIGURE_PACKAGE_CONFIG_FILE(<input> <output> INSTALL_DESTINATION <path>
                                               [PATH_VARS <var1> <var2> ... <varN>]
                                               [NO_SET_AND_CHECK_MACRO]
                                               [NO_CHECK_REQUIRED_COMPONENTS_MACRO])
```

CONFIGURE_PACKAGE_CONFIG_FILE() should be used instead of the plain configure_file() command when creating the <Name>Config.cmake or <Name>-config.cmake file for installing a project or library. It helps making the resulting package relocatable by avoiding hardcoded paths in the installed Config.cmake file.

In a FooConfig.cmake file there may be code like this to make the install destinations know to the using project:

```
set(FOO_INCLUDE_DIR   "@CMAKE_INSTALL_FULL_INCLUDEDIR@" )
set(FOO_DATA_DIR   "@CMAKE_INSTALL_PREFIX@/@RELATIVE_DATA_INSTALL_DIR@" )
set(FOO_ICONS_DIR   "@CMAKE_INSTALL_PREFIX@/share/icons" )
...logic to determine installedPrefix from the own location...
set(FOO_CONFIG_DIR  "${installedPrefix}/@CONFIG_INSTALL_DIR@" )
```

All 4 options shown above are not sufficient, since the first 3 hardcode the absolute directory locations, and the 4th case works only if the logic to determine the installedPrefix is correct, and if CONFIG_INSTALL_DIR contains a relative path, which in general cannot be guaranteed. This has the effect that the resulting FooConfig.cmake file would work poorly under Windows and OSX, where users are used to choose the install location of a binary package at install time, independent from how CMAKE_INSTALL_PREFIX was set at build/cmake time.

Using CONFIGURE_PACKAGE_CONFIG_FILE() helps. If used correctly, it makes the resulting FooConfig.cmake file relocatable. Usage:

1. write a FooConfig.cmake.in file as you are used to
2. insert a line containing only the string "@PACKAGE_INIT@"
3. instead of set(FOO_DIR "@SOME_INSTALL_DIR@"), use set(FOO_DIR "@PACKAGE_SOME_INSTALL_DIR@")
   (this must be after the @PACKAGE_INIT@ line)
4. instead of using the normal configure_file(), use CONFIGURE_PACKAGE_CONFIG_FILE()

The <input> and <output> arguments are the input and output file, the same way as in configure_file().

The <path> given to INSTALL_DESTINATION must be the destination where the FooConfig.cmake file will be installed to. This can either be a relative or absolute path, both work.

The variables <var1> to <varN> given as PATH_VARS are the variables which contain install destinations. For each of them the macro will create a helper variable PACKAGE_<var...>. These helper variables must be used in the FooConfig.cmake.in file for setting the installed location. They are calculated by CONFIGURE_PACKAGE_CONFIG_FILE() so that they are always relative to the installed location of the package. This works both for relative and also for absolute locations. For absolute locations it works only if the absolute location is a subdirectory of CMAKE_INSTALL_PREFIX.

By default configure_package_config_file() also generates two helper macros, set_and_check() and check_required_components() into the FooConfig.cmake file.

set_and_check() should be used instead of the normal set() command for setting directories and file locations. Additionally to setting the variable it also checks that the referenced file or directory actually exists and fails with a FATAL_ERROR otherwise. This makes sure that the created FooConfig.cmake file does not contain wrong references. When using the NO_SET_AND_CHECK_MACRO, this macro is not generated into the FooConfig.cmake file.

check_required_components(<package_name>) should be called at the end of the FooConfig.cmake file if the package supports components. This macro checks whether all requested, non-optional components have been found, and if this is not the case, sets the Foo_FOUND variable to FALSE, so that the package is considered to be not found. It does that by testing the Foo_<Component>_FOUND variables for all requested required components. When using the NO_CHECK_REQUIRED_COMPONENTS option, this macro is not generated into the FooConfig.cmake file.

For an example see below the documentation for WRITE_BASIC_PACKAGE_VERSION_FILE().

```
WRITE_BASIC_PACKAGE_VERSION_FILE( filename VERSION major.minor.patch COMPATIBILITY (AnyNewerVersion|SameMajorVersion|ExactVersion) )
```

Writes a file for use as <package>ConfigVersion.cmake file to <filename>. See the documentation of find_package() for details on this.

```
filename is the output filename, it should be in the build tree.
major.minor.patch is the version number of the project to be installed
```

The COMPATIBILITY mode AnyNewerVersion means that the installed package version will be considered compatible if it is newer or exactly the same as the requested version. This mode should be used for packages which are fully backward compatible, also across major versions. If SameMajorVersion is used instead, then the behaviour differs from AnyNewerVersion in that the major version number must be the same as requested, e.g. version 2.0 will not be considered compatible if 1.0 is requested. This mode should be used for packages which guarantee backward compatibility within the same major version. If ExactVersion is used, then the package is only considered compatible if the requested version matches exactly its own version number (not considering the tweak version). For example, version 1.2.3 of a package is only considered compatible to requested version 1.2.3. This mode is for packages without compatibility guarantees. If your project has more elaborated version matching rules, you will need to write your own custom ConfigVersion.cmake file instead of using this macro.

Internally, this macro executes configure_file() to create the resulting version file. Depending on the COMPATIBLITY, either the file BasicConfigVersion-SameMajorVersion.cmake.in or BasicConfigVersion-AnyNewerVersion.cmake.in is used. Please note that these two files are internal to CMake and you should not call configure_file() on them yourself, but they can be used as starting point to create more sophisticted custom ConfigVersion.cmake files.

Example using both configure_package_config_file() and write_basic_package_version_file(): CMakeLists.txt:

```
set(INCLUDE_INSTALL_DIR include/ ... CACHE )
set(LIB_INSTALL_DIR lib/ ... CACHE )
set(SYSCONFIG_INSTALL_DIR etc/foo/ ... CACHE )
...
include(CMakePackageConfigHelpers)
configure_package_config_file(FooConfig.cmake.in ${CMAKE_CURRENT_BINARY_DIR}/FooConfig.cmake
                              INSTALL_DESTINATION ${LIB_INSTALL_DIR}/Foo/cmake
                              PATH_VARS INCLUDE_INSTALL_DIR SYSCONFIG_INSTALL_DIR)
write_basic_package_version_file(${CMAKE_CURRENT_BINARY_DIR}/FooConfigVersion.cmake
                                 VERSION 1.2.3
                                 COMPATIBILITY SameMajorVersion )
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/FooConfig.cmake ${CMAKE_CURRENT_BINARY_DIR}/FooConfigVersion.cmake
        DESTINATION ${LIB_INSTALL_DIR}/Foo/cmake )
```

With a FooConfig.cmake.in:

```
set(FOO_VERSION x.y.z)
...
@PACKAGE_INIT@
...
set_and_check(FOO_INCLUDE_DIR "@PACKAGE_INCLUDE_INSTALL_DIR@")
set_and_check(FOO_SYSCONFIG_DIR "@PACKAGE_SYSCONFIG_INSTALL_DIR@")


check_required_components(Foo)
```

- **CMakeParseArguments**:

  CMAKE_PARSE_ARGUMENTS(<prefix> <options> <one_value_keywords> <multi_value_keywords> args...)

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

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

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

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

  When done, CMAKE_PARSE_ARGUMENTS() will have defined for each of the keywords listed in <options>, <one_value_keywords> and <multi_value_keywords> a variable composed of the given <prefix> followed by "_" and the name of the respective keyword. These variables will then hold the respective value from the argument list. For the <options> keywords this will be TRUE or FALSE.

  All remaining arguments are collected in a variable <prefix>_UNPARSED_ARGUMENTS, this can be checked afterwards to see whether your macro was called with unrecognized parameters.

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

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

Assume my_install() has been called like this:

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

After the cmake_parse_arguments() call the macro will have set the following variables:

```
MY_INSTALL_OPTIONAL = TRUE
MY_INSTALL_FAST = FALSE (this option was not used when calling my_install()
MY_INSTALL_DESTINATION = "bin"
MY_INSTALL_RENAME = "" (was not used)
MY_INSTALL_TARGETS = "foo;bar"
MY_INSTALL_CONFIGURATIONS = "" (was not used)
MY_INSTALL_UNPARSED_ARGUMENTS = "blub" (no value expected after "OPTIONAL"
```

You can then continue and process these variables.

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

- **`CMakePrintHelpers`**: Convenience macros for printing properties and variables, useful e.g. for debugging.

CMAKE_PRINT_PROPERTIES([TARGETS target1 .. targetN]

```
                        [SOURCES source1 .. sourceN]
                        [DIRECTORIES dir1 .. dirN]
                        [TESTS test1 .. testN]
                        [CACHE_ENTRIES entry1 .. entryN]
                        PROPERTIES prop1 .. propN )
```

This macro prints the values of the properties of the given targets, source files, directories, tests or cache entries. Exactly one of the scope keywords must be used. Example:

```
cmake_print_properties(TARGETS foo bar PROPERTIES LOCATION INTERFACE_INCLUDE_DIRS)
```

This will print the LOCATION and INTERFACE_INCLUDE_DIRS properties for both targets foo and bar.

CMAKE_PRINT_VARIABLES(var1 var2 .. varN)

This macro will print the name of each variable followed by its value. Example:

```
cmake_print_variables(CMAKE_C_COMPILER CMAKE_MAJOR_VERSION THIS_ONE_DOES_NOT_EXIST)
```

Gives:

```
-- CMAKE_C_COMPILER="/usr/bin/gcc" ; CMAKE_MAJOR_VERSION="2" ; THIS_ONE_DOES_NOT_EXIST=""
```

- **`CMakePrintSystemInformation`**: print system information

  This file can be used for diagnostic purposes just include it in a project to see various internal CMake variables.

- **`CMakePushCheckState`**:

  This module defines three macros: CMAKE_PUSH_CHECK_STATE() CMAKE_POP_CHECK_STATE() and CMAKE_RESET_CHECK_STATE() These macros can be used to save, restore and reset (i.e., clear contents) the state of the variables CMAKE_REQUIRED_FLAGS, CMAKE_REQUIRED_DEFINITIONS, CMAKE_REQUIRED_LIBRARIES and CMAKE_REQUIRED_INCLUDES used by the various Check-files coming with CMake, like e.g. check_function_exists() etc. The variable contents are pushed on a stack, pushing multiple times is supported. This is useful e.g. when executing such tests in a Find-module, where they have to be set, but after the Find-module has been executed they should have the same value as they had before.

CMAKE_PUSH_CHECK_STATE() macro receives optional argument RESET. Whether it's specified, CMAKE_PUSH_CHECK_STATE() will set all CMAKE_REQUIRED_* variables to empty values, same as CMAKE_RESET_CHECK_STATE() call will do.

Usage:

```
cmake_push_check_state(RESET)
set(CMAKE_REQUIRED_DEFINITIONS -DSOME_MORE_DEF)
check_function_exists(...)
cmake_reset_check_state()
set(CMAKE_REQUIRED_DEFINITIONS -DANOTHER_DEF)
check_function_exists(...)
cmake_pop_check_state()
```

- **`CMakeVerifyManifest`**:

  CMakeVerifyManifest.cmake

  This script is used to verify that embeded manifests and side by side manifests for a project match. To run this script, cd to a directory and run the script with cmake -P. On the command line you can pass in versions that are OK even if not found in the .manifest files. For example, cmake -Dallow_versions=8.0.50608.0 -PCmakeVerifyManifest.cmake could be used to allow an embeded manifest of 8.0.50608.0 to be used in a project even if that version was not found in the .manifest file.

- **`CPack`**: Build binary and source package installers.

  The CPack module generates binary and source installers in a variety of formats using the cpack program. Inclusion of the CPack module adds two new targets to the resulting makefiles, package and package_source, which build the binary and source installers, respectively. The generated binary installers contain everything installed via CMake's INSTALL command (and the deprecated INSTALL_FILES, INSTALL_PROGRAMS, and INSTALL_TARGETS commands).

  For certain kinds of binary installers (including the graphical installers on Mac OS X and Windows), CPack generates installers that allow users to select individual application components to install. See CPackComponent module for that.

  The CPACK_GENERATOR variable has different meanings in different contexts. In your CMakeLists.txt file, CPACK_GENERATOR is a *list of generators*: when run with no other arguments, CPack will iterate over that list and produce one package for each generator. In a CPACK_PROJECT_CONFIG_FILE, though, CPACK_GENERATOR is a *string naming a single generator*. If you need per-cpack- generator logic to control *other* cpack settings, then you need a CPACK_PROJECT_CONFIG_FILE.

  The CMake source tree itself contains a CPACK_PROJECT_CONFIG_FILE. See the top level file CMakeCPackOptions.cmake.in for an example.

  If set, the CPACK_PROJECT_CONFIG_FILE is included automatically on a per-generator basis. It only need contain overrides.

  Here's how it works:

  ```
  - cpack runs
  - it includes CPackConfig.cmake
  - it iterates over the generators listed in that file's
    CPACK_GENERATOR list variable (unless told to use just a
    specific one via -G on the command line...)


  - foreach generator, it then
    - sets CPACK_GENERATOR to the one currently being iterated
    - includes the CPACK_PROJECT_CONFIG_FILE
    - produces the package for that generator
  ```

  This is the key: For each generator listed in CPACK_GENERATOR in CPackConfig.cmake, cpack will *reset* CPACK_GENERATOR internally to *the one currently being used* and then include the CPACK_PROJECT_CONFIG_FILE.

  Before including this CPack module in your CMakeLists.txt file, there are a variety of variables that can be set to customize the resulting installers. The most commonly-used variables are:

  ```
  CPACK_PACKAGE_NAME - The name of the package (or application). If
  not specified, defaults to the project name.


  CPACK_PACKAGE_VENDOR - The name of the package vendor. (e.g.,
  "Kitware").


  CPACK_PACKAGE_DIRECTORY - The directory in which CPack is doing its
  packaging. If it is not set then this will default (internally) to the
  build dir. This variable may be defined in CPack config file or from
  the cpack command line option "-B". If set the command line option
  override the value found in the config file.
  ```

CPACK_PACKAGE_VERSION_MAJOR - Package major Version

CPACK_PACKAGE_VERSION_MINOR - Package minor Version

CPACK_PACKAGE_VERSION_PATCH - Package patch Version

CPACK_PACKAGE_DESCRIPTION_FILE - A text file used to describe the
project. Used, for example, the introduction screen of a
CPack-generated Windows installer to describe the project.

CPACK_PACKAGE_DESCRIPTION_SUMMARY - Short description of the
project (only a few words).

CPACK_PACKAGE_FILE_NAME - The name of the package file to generate,
not including the extension. For example, cmake-2.6.1-Linux-i686.
The default value is

${CPACK_PACKAGE_NAME}-${CPACK_PACKAGE_VERSION}-${CPACK_SYSTEM_NAME}.

CPACK_PACKAGE_INSTALL_DIRECTORY - Installation directory on the
target system. This may be used by some CPack generators
like NSIS to create an installation directory e.g., "CMake 2.5"
below the installation prefix. All installed element will be
put inside this directory.

CPACK_PACKAGE_ICON - A branding image that will be displayed inside
the installer (used by GUI installers).

CPACK_PROJECT_CONFIG_FILE - CPack-time project CPack configuration
file. This file included at cpack time, once per
generator after CPack has set CPACK_GENERATOR to the actual generator
being used. It allows per-generator setting of CPACK_* variables at
cpack time.

CPACK_RESOURCE_FILE_LICENSE - License to be embedded in the installer. It
will typically be displayed to the user by the produced installer
(often with an explicit "Accept" button, for graphical installers)
prior to installation. This license file is NOT added to installed
file but is used by some CPack generators like NSIS. If you want
to install a license file (may be the same as this one)
along with your project you must add an appropriate CMake INSTALL
command in your CMakeLists.txt.

CPACK_RESOURCE_FILE_README - ReadMe file to be embedded in the installer. It
typically describes in some detail the purpose of the project
during the installation. Not all CPack generators uses
this file.

CPACK_RESOURCE_FILE_WELCOME - Welcome file to be embedded in the
installer. It welcomes users to this installer.
Typically used in the graphical installers on Windows and Mac OS X.

CPACK_MONOLITHIC_INSTALL - Disables the component-based
installation mechanism. When set the component specification is ignored
and all installed items are put in a single "MONOLITHIC" package.

```
Some CPack generators do monolithic packaging by default and
may be asked to do component packaging by setting
CPACK_<GENNAME>_COMPONENT_INSTALL to 1/TRUE.


CPACK_GENERATOR - List of CPack generators to use. If not
specified, CPack will create a set of options CPACK_BINARY_<GENNAME> (e.g.,
CPACK_BINARY_NSIS) allowing the user to enable/disable individual
generators. This variable may be used on the command line
as well as in:


  cpack -D CPACK_GENERATOR="ZIP;TGZ" /path/to/build/tree


CPACK_OUTPUT_CONFIG_FILE - The name of the CPack binary configuration
file. This file is the CPack configuration generated by the CPack module
for binary installers. Defaults to CPackConfig.cmake.


  CPACK_PACKAGE_EXECUTABLES - Lists each of the executables and associated
  text label to be used to create Start Menu shortcuts. For example,
  setting this to the list ccmake;CMake will
  create a shortcut named "CMake" that will execute the installed
  executable ccmake. Not all CPack generators use it (at least NSIS and
  OSXX11 do).


  CPACK_STRIP_FILES - List of files to be stripped. Starting with
  CMake 2.6.0 CPACK_STRIP_FILES will be a boolean variable which
  enables stripping of all files (a list of files evaluates to TRUE
  in CMake, so this change is compatible).
```

The following CPack variables are specific to source packages, and will not affect binary packages:

```
  CPACK_SOURCE_PACKAGE_FILE_NAME - The name of the source package. For
  example cmake-2.6.1.


  CPACK_SOURCE_STRIP_FILES - List of files in the source tree that
  will be stripped. Starting with CMake 2.6.0
  CPACK_SOURCE_STRIP_FILES will be a boolean variable which enables
  stripping of all files (a list of files evaluates to TRUE in CMake,
  so this change is compatible).


  CPACK_SOURCE_GENERATOR - List of generators used for the source
  packages. As with CPACK_GENERATOR, if this is not specified then
  CPack will create a set of options (e.g., CPACK_SOURCE_ZIP)
  allowing users to select which packages will be generated.


  CPACK_SOURCE_OUTPUT_CONFIG_FILE - The name of the CPack source
  configuration file. This file is the CPack configuration generated by the
  CPack module for source installers. Defaults to CPackSourceConfig.cmake.


  CPACK_SOURCE_IGNORE_FILES - Pattern of files in the source tree
  that won't be packaged when building a source package. This is a
  list of regular expression patterns (that must be properly escaped),
  e.g., /CVS/;/\\.svn/;\\.swp$;\\.#;/#;.*~;cscope.*
```

The following variables are for advanced uses of CPack:

```
  CPACK_CMAKE_GENERATOR - What CMake generator should be used if the
  project is CMake project. Defaults to the value of CMAKE_GENERATOR
  few users will want to change this setting.
```

```
CPACK_INSTALL_CMAKE_PROJECTS - List of four values that specify
what project to install. The four values are: Build directory,
Project Name, Project Component, Directory. If omitted, CPack will
build an installer that installers everything.


CPACK_SYSTEM_NAME - System name, defaults to the value of
${CMAKE_SYSTEM_NAME}.


CPACK_PACKAGE_VERSION - Package full version, used internally. By
default, this is built from CPACK_PACKAGE_VERSION_MAJOR,
CPACK_PACKAGE_VERSION_MINOR, and CPACK_PACKAGE_VERSION_PATCH.


CPACK_TOPLEVEL_TAG - Directory for the installed files.


  CPACK_INSTALL_COMMANDS - Extra commands to install components.


  CPACK_INSTALLED_DIRECTORIES - Extra directories to install.


   CPACK_PACKAGE_INSTALL_REGISTRY_KEY - Registry key used when
   installing this project. This is only used by installer for Windows.
   The default value is based on the installation directory.
   CPACK_CREATE_DESKTOP_LINKS - List of desktop links to create.
```

- **`CPackBundle`**: CPack Bundle generator (Mac OS X) specific options

  Installers built on Mac OS X using the Bundle generator use the aforementioned DragNDrop (CPACK_DMG_xxx) variables, plus the following Bundle-specific parameters (CPACK_BUNDLE_xxx).

```
  CPACK_BUNDLE_NAME - The name of the generated bundle. This
  appears in the OSX finder as the bundle name. Required.


  CPACK_BUNDLE_PLIST - Path to an OSX plist file that will be used
  for the generated bundle. This assumes that the caller has generated
  or specified their own Info.plist file. Required.


  CPACK_BUNDLE_ICON - Path to an OSX icon file that will be used as
  the icon for the generated bundle. This is the icon that appears in the
  OSX finder for the bundle, and in the OSX dock when the bundle is opened.
  Required.


  CPACK_BUNDLE_STARTUP_COMMAND - Path to a startup script. This is a path to
  an executable or script that will be run whenever an end-user double-clicks
  the generated bundle in the OSX Finder. Optional.
```

- **`CPackComponent`**: Build binary and source package installers

  The CPackComponent module is the module which handles the component part of CPack. See CPack module for general information about CPack.

  For certain kinds of binary installers (including the graphical installers on Mac OS X and Windows), CPack generates installers that allow users to select individual application components to install. The contents of each of the components are identified by the COMPONENT argument of CMake's INSTALL command. These components can be annotated with user-friendly names and descriptions, inter-component dependencies, etc., and grouped in various ways to customize the resulting installer. See the cpack_add_* commands, described below, for more information about component-specific installations.

  Component-specific installation allows users to select specific sets of components to install during the install process. Installation components are identified by the COMPONENT argument of CMake's INSTALL commands, and should be further described by the following CPack commands:

```
  CPACK_COMPONENTS_ALL - The list of component to install.
```

The default value of this variable is computed by CPack and contains all components defined by the project. The user may set it to only include the specified components.

```
CPACK_<GENNAME>_COMPONENT_INSTALL - Enable/Disable component install for
CPack generator <GENNAME>.
```

Each CPack Generator (RPM, DEB, ARCHIVE, NSIS, DMG, etc...) has a legacy default behavior. e.g. RPM builds monolithic whereas NSIS builds component. One can change the default behavior by setting this variable to 0/1 or OFF/ON.

```
CPACK_COMPONENTS_GROUPING - Specify how components are grouped for multi-package
component-aware CPack generators.
```

Some generators like RPM or ARCHIVE family (TGZ, ZIP, ...) generates several packages files when asked for component packaging. They group the component differently depending on the value of this variable:

```
- ONE_PER_GROUP (default): creates one package file per component group
- ALL_COMPONENTS_IN_ONE : creates a single package with all (requested) component
- IGNORE : creates one package per component, i.e. IGNORE component group
```

One can specify different grouping for different CPack generator by using a CPACK_PROJECT_CONFIG_FILE.

```
CPACK_COMPONENT_<compName>_DISPLAY_NAME - The name to be displayed for a component.
CPACK_COMPONENT_<compName>_DESCRIPTION - The description of a component.
CPACK_COMPONENT_<compName>_GROUP - The group of a component.
CPACK_COMPONENT_<compName>_DEPENDS - The dependencies (list of components)
on which this component depends.
CPACK_COMPONENT_<compName>_REQUIRED - True is this component is required.
```

cpack_add_component - Describes a CPack installation component named by the COMPONENT argument to a CMake INSTALL command.

```
cpack_add_component(compname
                    [DISPLAY_NAME name]
                    [DESCRIPTION description]
                    [HIDDEN | REQUIRED | DISABLED ]
                    [GROUP group]
                    [DEPENDS comp1 comp2 ... ]
                    [INSTALL_TYPES type1 type2 ... ]
                    [DOWNLOADED]
                    [ARCHIVE_FILE filename])
```

The cmake_add_component command describes an installation component, which the user can opt to install or remove as part of the graphical installation process. compname is the name of the component, as provided to the COMPONENT argument of one or more CMake INSTALL commands.

DISPLAY_NAME is the displayed name of the component, used in graphical installers to display the component name. This value can be any string.

DESCRIPTION is an extended description of the component, used in graphical installers to give the user additional information about the component. Descriptions can span multiple lines using "\n" as the line separator. Typically, these descriptions should be no more than a few lines long.

HIDDEN indicates that this component will be hidden in the graphical installer, so that the user cannot directly change whether it is installed or not.

REQUIRED indicates that this component is required, and therefore will always be installed. It will be visible in the graphical installer, but it cannot be unselected. (Typically, required components are shown greyed out).

DISABLED indicates that this component should be disabled (unselected) by default. The user is free to select this component for installation, unless it is also HIDDEN.

DEPENDS lists the components on which this component depends. If this component is selected, then each of the components listed must also be selected. The dependency information is encoded within the installer itself, so that users cannot install inconsistent sets of components.

GROUP names the component group of which this component is a part. If not provided, the component will be a standalone component, not part of any component group. Component groups are described with the cpack_add_component_group command, detailed below.

INSTALL_TYPES lists the installation types of which this component is a part. When one of these installations types is selected, this component will automatically be selected. Installation types are described with the cpack_add_install_type command, detailed below.

DOWNLOADED indicates that this component should be downloaded on-the-fly by the installer, rather than packaged in with the installer itself. For more information, see the cpack_configure_downloads command.

ARCHIVE_FILE provides a name for the archive file created by CPack to be used for downloaded components. If not supplied, CPack will create a file with some name based on CPACK_PACKAGE_FILE_NAME and the name of the component. See cpack_configure_downloads for more information.

cpack_add_component_group - Describes a group of related CPack installation components.

```
cpack_add_component_group(groupname
                          [DISPLAY_NAME name]
                          [DESCRIPTION description]
                          [PARENT_GROUP parent]
                          [EXPANDED]
                          [BOLD_TITLE])
```

The cpack_add_component_group describes a group of installation components, which will be placed together within the listing of options. Typically, component groups allow the user to select/deselect all of the components within a single group via a single group-level option. Use component groups to reduce the complexity of installers with many options. groupname is an arbitrary name used to identify the group in the GROUP argument of the cpack_add_component command, which is used to place a component in a group. The name of the group must not conflict with the name of any component.

DISPLAY_NAME is the displayed name of the component group, used in graphical installers to display the component group name. This value can be any string.

DESCRIPTION is an extended description of the component group, used in graphical installers to give the user additional information about the components within that group. Descriptions can span multiple lines using "\n" as the line separator. Typically, these descriptions should be no more than a few lines long.

PARENT_GROUP, if supplied, names the parent group of this group. Parent groups are used to establish a hierarchy of groups, providing an arbitrary hierarchy of groups.

EXPANDED indicates that, by default, the group should show up as "expanded", so that the user immediately sees all of the components within the group. Otherwise, the group will initially show up as a single entry.

BOLD_TITLE indicates that the group title should appear in bold, to call the user's attention to the group.

cpack_add_install_type - Add a new installation type containing a set of predefined component selections to the graphical installer.

```
cpack_add_install_type(typename
                       [DISPLAY_NAME name])
```

The cpack_add_install_type command identifies a set of preselected components that represents a common use case for an application. For example, a "Developer" install type might include an application along with its header and library files, while an "End user" install type might just include the application's executable. Each component identifies itself with one or more install types via the INSTALL_TYPES argument to cpack_add_component.

DISPLAY_NAME is the displayed name of the install type, which will typically show up in a drop-down box within a graphical installer. This value can be any string.

cpack_configure_downloads - Configure CPack to download selected components on-the-fly as part of the installation process.

```
cpack_configure_downloads(site
                          [UPLOAD_DIRECTORY dirname]
                          [ALL]
                          [ADD_REMOVE|NO_ADD_REMOVE])
```

The cpack_configure_downloads command configures installation-time downloads of selected components. For each downloadable component, CPack will create an archive containing the contents of that component, which should be uploaded to the given site. When the user selects that component for installation, the installer will download and extract the component in place. This feature is useful for creating small installers that only download the requested components, saving bandwidth. Additionally, the installers are small enough that they will be installed as part of the normal installation process, and the "Change" button in Windows Add/Remove Programs control panel will allow one to add or remove parts of the application after the original installation. On Windows, the downloaded-components functionality requires the ZipDLL plug-in for NSIS, available at:

http://nsis.sourceforge.net/ZipDLL_plug-in

On Mac OS X, installers that download components on-the-fly can only be built and installed on system using Mac OS X 10.5 or later.

The site argument is a URL where the archives for downloadable components will reside, e.g.,

http://www.cmake.org/files/2.6.1/installer/ All of the archives produced by CPack should be uploaded to that location.

UPLOAD_DIRECTORY is the local directory where CPack will create the various archives for each of the components. The contents of this directory should be uploaded to a location accessible by the URL given in the site argument. If omitted, CPack will use the directory CPackUploads inside the CMake binary directory to store the generated archives.

The ALL flag indicates that all components be downloaded. Otherwise, only those components explicitly marked as DOWNLOADED or that have a specified ARCHIVE_FILE will be downloaded. Additionally, the ALL option implies ADD_REMOVE (unless NO_ADD_REMOVE is specified).

ADD_REMOVE indicates that CPack should install a copy of the installer that can be called from Windows' Add/Remove Programs dialog (via the "Modify" button) to change the set of installed components. NO_ADD_REMOVE turns off this behavior. This option is ignored on Mac OS X.

- **`CPackCygwin`**: Cygwin CPack generator (Cygwin).

  The following variable is specific to installers build on and/or for Cygwin:

  ```
  CPACK_CYGWIN_PATCH_NUMBER - The Cygwin patch number.
  FIXME: This documentation is incomplete.
  CPACK_CYGWIN_PATCH_FILE - The Cygwin patch file.
  FIXME: This documentation is incomplete.
  CPACK_CYGWIN_BUILD_SCRIPT - The Cygwin build script.
  FIXME: This documentation is incomplete.
  ```

- **`CPackDMG`**: DragNDrop CPack generator (Mac OS X).

  The following variables are specific to the DragNDrop installers built on Mac OS X:

  ```
  CPACK_DMG_VOLUME_NAME - The volume name of the generated disk
  image. Defaults to CPACK_PACKAGE_FILE_NAME.


  CPACK_DMG_FORMAT - The disk image format. Common values are UDRO
  (UDIF read-only), UDZO (UDIF zlib-compressed) or UDBZ (UDIF
  bzip2-compressed). Refer to hdiutil(1) for more information on
  other available formats.


  CPACK_DMG_DS_STORE - Path to a custom DS_Store file. This .DS_Store
  file e.g. can be used to specify the Finder window
  position/geometry and layout (such as hidden toolbars, placement of the
  icons etc.). This file has to be generated by the Finder (either manually or
  through OSA-script) using a normal folder from which the .DS_Store
  file can then be extracted.


  CPACK_DMG_BACKGROUND_IMAGE - Path to a background image file. This
  file will be used as the background for the Finder Window when the disk
  image is opened.  By default no background image is set. The background
  image is applied after applying the custom .DS_Store file.


  CPACK_COMMAND_HDIUTIL - Path to the hdiutil(1) command used to
  operate on disk image files on Mac OS X. This variable can be used
  to override the automatically detected command (or specify its
  location if the auto-detection fails to find it.)


  CPACK_COMMAND_SETFILE - Path to the SetFile(1) command used to set
  extended attributes on files and directories on Mac OS X. This
  variable can be used to override the automatically detected
  command (or specify its location if the auto-detection fails to
  find it.)


  CPACK_COMMAND_REZ - Path to the Rez(1) command used to compile
  resources on Mac OS X. This variable can be used to override the
  automatically detected command (or specify its location if the
  auto-detection fails to find it.)
  ```

- **`CPackDeb`**: The builtin (binary) CPack Deb generator (Unix only)

  CPackDeb may be used to create Deb package using CPack. CPackDeb is a CPack generator thus it uses the CPACK_XXX variables used by CPack : http://www.cmake.org/Wiki/CMake:CPackConfiguration. CPackDeb generator should work on any

linux host but it will produce better deb package when Debian specific tools 'dpkg-xxx' are usable on the build system.

CPackDeb has specific features which are controlled by the specifics CPACK_DEBIAN_XXX variables.You'll find a detailed usage on the wiki:

http://www.cmake.org/Wiki/CMake:CPackPackageGenerators#DEB_.28UNIX_only.29

However as a handy reminder here comes the list of specific variables:

CPACK_DEBIAN_PACKAGE_NAME

```
Mandatory : YES
Default   : CPACK_PACKAGE_NAME (lower case)
The debian package summary
```

CPACK_DEBIAN_PACKAGE_VERSION

```
Mandatory : YES
Default   : CPACK_PACKAGE_VERSION
The debian package version
```

CPACK_DEBIAN_PACKAGE_ARCHITECTURE

```
Mandatory : YES
Default   : Output of dpkg --print-architecture (or i386 if dpkg is not found)
The debian package architecture
```

CPACK_DEBIAN_PACKAGE_DEPENDS

```
Mandatory : NO
Default   : -
May be used to set deb dependencies.
```

CPACK_DEBIAN_PACKAGE_MAINTAINER

```
Mandatory : YES
Default   : CPACK_PACKAGE_CONTACT
The debian package maintainer
```

CPACK_DEBIAN_PACKAGE_DESCRIPTION

```
Mandatory : YES
Default   : CPACK_PACKAGE_DESCRIPTION_SUMMARY
The debian package description
```

CPACK_DEBIAN_PACKAGE_SECTION

```
Mandatory : YES
Default   : 'devel'
The debian package section
```

CPACK_DEBIAN_PACKAGE_PRIORITY

```
Mandatory : YES
Default   : 'optional'
The debian package priority
```

CPACK_DEBIAN_PACKAGE_HOMEPAGE

```
Mandatory : NO
Default   : -
The URL of the web site for this package, preferably (when applicable) the
site from which the original source can be obtained and any additional
upstream documentation or information may be found.
The content of this field is a simple URL without any surrounding
characters such as <>.
```

CPACK_DEBIAN_PACKAGE_SHLIBDEPS

```
Mandatory : NO
Default   : OFF
May be set to ON in order to use dpkg-shlibdeps to generate
better package dependency list.
You may need set CMAKE_INSTALL_RPATH toi appropriate value
if you use this feature, because if you don't dpkg-shlibdeps
may fail to find your own shared libs.
See http://www.cmake.org/Wiki/CMake_RPATH_handling.
```

CPACK_DEBIAN_PACKAGE_DEBUG

```
Mandatory : NO
Default   : -
```

May be set when invoking cpack in order to trace debug information
during CPackDeb run.

CPACK_DEBIAN_PACKAGE_PREDEPENDS

Mandatory : NO
Default : -
see http://www.debian.org/doc/debian-policy/ch-relationships.html#s-binarydeps
This field is like Depends, except that it also forces dpkg to complete installation of
the packages named before even starting the installation of the package which declares
the pre-dependency.

CPACK_DEBIAN_PACKAGE_ENHANCES

Mandatory : NO
Default : -
see http://www.debian.org/doc/debian-policy/ch-relationships.html#s-binarydeps
This field is similar to Suggests but works in the opposite direction.
It is used to declare that a package can enhance the functionality of another package.

CPACK_DEBIAN_PACKAGE_BREAKS

Mandatory : NO
Default : -
see http://www.debian.org/doc/debian-policy/ch-relationships.html#s-binarydeps
When one binary package declares that it breaks another, dpkg will refuse to allow the
package which declares Breaks be installed unless the broken package is deconfigured first,
and it will refuse to allow the broken package to be reconfigured.

CPACK_DEBIAN_PACKAGE_CONFLICTS

Mandatory : NO
Default : -
see http://www.debian.org/doc/debian-policy/ch-relationships.html#s-binarydeps
When one binary package declares a conflict with another using a Conflicts field,
dpkg will refuse to allow them to be installed on the system at the same time.

CPACK_DEBIAN_PACKAGE_PROVIDES

Mandatory : NO
Default : -
see http://www.debian.org/doc/debian-policy/ch-relationships.html#s-binarydeps
A virtual package is one which appears in the Provides control field of another package.

CPACK_DEBIAN_PACKAGE_REPLACES

Mandatory : NO
Default : -
see http://www.debian.org/doc/debian-policy/ch-relationships.html#s-binarydeps
Packages can declare in their control file that they should overwrite
files in certain other packages, or completely replace other packages.

CPACK_DEBIAN_PACKAGE_RECOMMENDS

Mandatory : NO
Default : -
see http://www.debian.org/doc/debian-policy/ch-relationships.html#s-binarydeps
Allows packages to declare a strong, but not absolute, dependency on other packages.

CPACK_DEBIAN_PACKAGE_SUGGESTS

Mandatory : NO
Default : -
see http://www.debian.org/doc/debian-policy/ch-relationships.html#s-binarydeps
Allows packages to declare a suggested package install grouping.

CPACK_DEBIAN_PACKAGE_CONTROL_EXTRA

Mandatory : NO
Default : -
This variable allow advanced user to add custom script to the control.tar.gz
Typical usage is for conffiles, postinst, postrm, prerm.
Usage: set(CPACK_DEBIAN_PACKAGE_CONTROL_EXTRA
"${CMAKE_CURRENT_SOURCE_DIR/prerm;${CMAKE_CURRENT_SOURCE_DIR}/postrm")

- **CPackNSIS**: CPack NSIS generator specific options

The following variables are specific to the graphical installers built on Windows using the Nullsoft Installation System.

CPACK_NSIS_INSTALL_ROOT - The default installation directory presented
to the end user by the NSIS installer is under this root dir. The full
directory presented to the end user is:
${CPACK_NSIS_INSTALL_ROOT}/${CPACK_PACKAGE_INSTALL_DIRECTORY}

CPACK_NSIS_MUI_ICON - An icon filename.
The name of a *.ico file used as the main icon for the generated
install program.

CPACK_NSIS_MUI_UNIICON - An icon filename.
The name of a *.ico file used as the main icon for the generated
uninstall program.

CPACK_NSIS_INSTALLER_MUI_ICON_CODE - undocumented.

CPACK_NSIS_EXTRA_PREINSTALL_COMMANDS - Extra NSIS commands that
will be added to the beginning of the install Section, before your
install tree is available on the target system.

CPACK_NSIS_EXTRA_INSTALL_COMMANDS - Extra NSIS commands that
will be added to the end of the install Section, after your
install tree is available on the target system.

CPACK_NSIS_EXTRA_UNINSTALL_COMMANDS - Extra NSIS commands that will
be added to the uninstall Section, before your install tree is
removed from the target system.

CPACK_NSIS_COMPRESSOR - The arguments that will be passed to the
NSIS SetCompressor command.

CPACK_NSIS_ENABLE_UNINSTALL_BEFORE_INSTALL - Ask about uninstalling
previous versions first.
If this is set to "ON", then an installer will look for previous
installed versions and if one is found, ask the user whether to
uninstall it before proceeding with the install.

CPACK_NSIS_MODIFY_PATH - Modify PATH toggle.
If this is set to "ON", then an extra page
will appear in the installer that will allow the user to choose
whether the program directory should be added to the system PATH
variable.

CPACK_NSIS_DISPLAY_NAME - The display name string that appears in
the Windows Add/Remove Program control panel

CPACK_NSIS_PACKAGE_NAME - The title displayed at the top of the
installer.

CPACK_NSIS_INSTALLED_ICON_NAME - A path to the executable that
contains the installer icon.

CPACK_NSIS_HELP_LINK - URL to a web site providing assistance in
installing your application.

CPACK_NSIS_URL_INFO_ABOUT - URL to a web site providing more

```
information about your application.


CPACK_NSIS_CONTACT - Contact information for questions and comments
about the installation process.


CPACK_NSIS_CREATE_ICONS_EXTRA - Additional NSIS commands for
creating start menu shortcuts.


CPACK_NSIS_DELETE_ICONS_EXTRA -Additional NSIS commands to
uninstall start menu shortcuts.


CPACK_NSIS_EXECUTABLES_DIRECTORY - Creating NSIS start menu links
assumes that they are in 'bin' unless this variable is set.
For example, you would set this to 'exec' if your executables are
in an exec directory.


CPACK_NSIS_MUI_FINISHPAGE_RUN - Specify an executable to add an option
to run on the finish page of the NSIS installer.
CPACK_NSIS_MENU_LINKS - Specify links in [application] menu.
This should contain a list of pair "link" "link name". The link
may be an URL or a path relative to installation prefix.
Like:
  set(CPACK_NSIS_MENU_LINKS
      "doc/cmake-@CMake_VERSION_MAJOR@.@CMake_VERSION_MINOR@/cmake.html" "CMake Help"
      "http://www.cmake.org" "CMake Web Site")
```

- **`CPackPackageMaker`**: PackageMaker CPack generator (Mac OS X).

  The following variable is specific to installers built on Mac OS X using PackageMaker:

```
CPACK_OSX_PACKAGE_VERSION - The version of Mac OS X that the
resulting PackageMaker archive should be compatible with. Different
versions of Mac OS X support different
features. For example, CPack can only build component-based
installers for Mac OS X 10.4 or newer, and can only build
installers that download component son-the-fly for Mac OS X 10.5
or newer. If left blank, this value will be set to the minimum
version of Mac OS X that supports the requested features. Set this
variable to some value (e.g., 10.4) only if you want to guarantee
that your installer will work on that version of Mac OS X, and
don't mind missing extra features available in the installer
shipping with later versions of Mac OS X.
```

- **`CPackRPM`**: The builtin (binary) CPack RPM generator (Unix only)

  CPackRPM may be used to create RPM package using CPack. CPackRPM is a CPack generator thus it uses the CPACK_XXX variables used by CPack : http://www.cmake.org/Wiki/CMake:CPackConfiguration

  However CPackRPM has specific features which are controlled by the specifics CPACK_RPM_XXX variables. CPackRPM is a component aware generator so when CPACK_RPM_COMPONENT_INSTALL is ON some more CPACK_RPM_<ComponentName>_XXXX variables may be used in order to have component specific values. Note however that <componentName> refers to the **grouping name**. This may be either a component name or a component GROUP name. Usually those vars correspond to RPM spec file entities, one may find information about spec files here http://www.rpm.org/wiki/Docs. You'll find a detailed usage of CPackRPM on the wiki:

  http://www.cmake.org/Wiki/CMake:CPackPackageGenerators#RPM_.28Unix_Only.29

  However as a handy reminder here comes the list of specific variables:

```
CPACK_RPM_PACKAGE_SUMMARY - The RPM package summary.
   Mandatory : YES
   Default   : CPACK_PACKAGE_DESCRIPTION_SUMMARY
CPACK_RPM_PACKAGE_NAME - The RPM package name.
   Mandatory : YES
   Default   : CPACK_PACKAGE_NAME
CPACK_RPM_PACKAGE_VERSION - The RPM package version.
   Mandatory : YES
   Default   : CPACK_PACKAGE_VERSION
CPACK_RPM_PACKAGE_ARCHITECTURE - The RPM package architecture.
```

```
   Mandatory : NO
   Default   : -
   This may be set to "noarch" if you
   know you are building a noarch package.
CPACK_RPM_PACKAGE_RELEASE - The RPM package release.
   Mandatory : YES
   Default   : 1
   This is the numbering of the RPM package
   itself, i.e. the version of the packaging and not the version of the
   content (see CPACK_RPM_PACKAGE_VERSION). One may change the default
   value if the previous packaging was buggy and/or you want to put here
   a fancy Linux distro specific numbering.
CPACK_RPM_PACKAGE_LICENSE - The RPM package license policy.
   Mandatory : YES
   Default   : "unknown"
CPACK_RPM_PACKAGE_GROUP - The RPM package group.
   Mandatory : YES
   Default   : "unknown"
CPACK_RPM_PACKAGE_VENDOR - The RPM package vendor.
   Mandatory : YES
   Default   : CPACK_PACKAGE_VENDOR if set or "unknown"
CPACK_RPM_PACKAGE_URL - The projects URL.
   Mandatory : NO
   Default   : -
CPACK_RPM_PACKAGE_DESCRIPTION - RPM package description.
   Mandatory : YES
   Default   : CPACK_PACKAGE_DESCRIPTION_FILE if set or "no package description available"
CPACK_RPM_COMPRESSION_TYPE - RPM compression type.
   Mandatory : NO
   Default   : -
   May be used to override RPM compression type to be used
   to build the RPM. For example some Linux distribution now default
   to lzma or xz compression whereas older cannot use such RPM.
   Using this one can enforce compression type to be used.
   Possible value are: lzma, xz, bzip2 and gzip.
CPACK_RPM_PACKAGE_REQUIRES - RPM spec requires field.
   Mandatory : NO
   Default   : -
   May be used to set RPM dependencies (requires).
   Note that you must enclose the complete requires string between quotes,
   for example:
   set(CPACK_RPM_PACKAGE_REQUIRES "python >= 2.5.0, cmake >= 2.8")
   The required package list of an RPM file could be printed with
   rpm -qp --requires file.rpm
CPACK_RPM_PACKAGE_SUGGESTS - RPM spec suggest field.
   Mandatory : NO
   Default   : -
   May be used to set weak RPM dependencies (suggests).
   Note that you must enclose the complete requires string between quotes.
CPACK_RPM_PACKAGE_PROVIDES - RPM spec provides field.
   Mandatory : NO
   Default   : -
   May be used to set RPM dependencies (provides).
   The provided package list of an RPM file could be printed with
   rpm -qp --provides file.rpm
CPACK_RPM_PACKAGE_OBSOLETES - RPM spec obsoletes field.
   Mandatory : NO
   Default   : -
   May be used to set RPM packages that are obsoleted by this one.
CPACK_RPM_PACKAGE_RELOCATABLE - build a relocatable RPM.
   Mandatory : NO
   Default   : CPACK_PACKAGE_RELOCATABLE
   If this variable is set to TRUE or ON CPackRPM will try
   to build a relocatable RPM package. A relocatable RPM may
   be installed using rpm --prefix or --relocate in order to
   install it at an alternate place see rpm(8).
   Note that currently this may fail if CPACK_SET_DESTDIR is set to ON.
   If CPACK_SET_DESTDIR is set then you will get a warning message
   but if there is file installed with absolute path you'll get
   unexpected behavior.
CPACK_RPM_SPEC_INSTALL_POST - [deprecated].
```

```
   Mandatory : NO
   Default   : -
   This way of specifying post-install script is deprecated use
   CPACK_RPM_POST_INSTALL_SCRIPT_FILE
   May be used to set an RPM post-install command inside the spec file.
   For example setting it to "/bin/true" may be used to prevent
   rpmbuild to strip binaries.
CPACK_RPM_SPEC_MORE_DEFINE - RPM extended spec definitions lines.
   Mandatory : NO
   Default   : -
   May be used to add any %define lines to the generated spec file.
CPACK_RPM_PACKAGE_DEBUG - Toggle CPackRPM debug output.
   Mandatory : NO
   Default   : -
   May be set when invoking cpack in order to trace debug information
   during CPack RPM run. For example you may launch CPack like this
   cpack -D CPACK_RPM_PACKAGE_DEBUG=1 -G RPM
CPACK_RPM_USER_BINARY_SPECFILE - A user provided spec file.
   Mandatory : NO
   Default   : -
   May be set by the user in order to specify a USER binary spec file
   to be used by CPackRPM instead of generating the file.
   The specified file will be processed by configure_file( @ONLY).
   One can provide a component specific file by setting
   CPACK_RPM_<componentName>_USER_BINARY_SPECFILE.
CPACK_RPM_GENERATE_USER_BINARY_SPECFILE_TEMPLATE - Spec file template.
   Mandatory : NO
   Default   : -
   If set CPack will generate a template for USER specified binary
   spec file and stop with an error. For example launch CPack like this
   cpack -D CPACK_RPM_GENERATE_USER_BINARY_SPECFILE_TEMPLATE=1 -G RPM
   The user may then use this file in order to hand-craft is own
   binary spec file which may be used with CPACK_RPM_USER_BINARY_SPECFILE.
CPACK_RPM_PRE_INSTALL_SCRIPT_FILE
CPACK_RPM_PRE_UNINSTALL_SCRIPT_FILE
   Mandatory : NO
   Default   : -
   May be used to embed a pre (un)installation script in the spec file.
   The refered script file(s) will be read and directly
   put after the %pre or %preun section
   If CPACK_RPM_COMPONENT_INSTALL is set to ON the (un)install script for
   each component can be overridden with
   CPACK_RPM_<COMPONENT>_PRE_INSTALL_SCRIPT_FILE and
   CPACK_RPM_<COMPONENT>_PRE_UNINSTALL_SCRIPT_FILE
   One may verify which scriptlet has been included with
    rpm -qp --scripts  package.rpm
CPACK_RPM_POST_INSTALL_SCRIPT_FILE
CPACK_RPM_POST_UNINSTALL_SCRIPT_FILE
   Mandatory : NO
   Default   : -
   May be used to embed a post (un)installation script in the spec file.
   The refered script file(s) will be read and directly
   put after the %post or %postun section
   If CPACK_RPM_COMPONENT_INSTALL is set to ON the (un)install script for
   each component can be overridden with
   CPACK_RPM_<COMPONENT>_POST_INSTALL_SCRIPT_FILE and
   CPACK_RPM_<COMPONENT>_POST_UNINSTALL_SCRIPT_FILE
   One may verify which scriptlet has been included with
    rpm -qp --scripts  package.rpm
CPACK_RPM_USER_FILELIST
CPACK_RPM_<COMPONENT>_USER_FILELIST
   Mandatory : NO
   Default   : -
   May be used to explicitly specify %(<directive>) file line
   in the spec file. Like %config(noreplace) or any other directive
   that be found in the %files section. Since CPackRPM is generating
   the list of files (and directories) the user specified files of
   the CPACK_RPM_<COMPONENT>_USER_FILELIST list will be removed from the generated list.
CPACK_RPM_CHANGELOG_FILE - RPM changelog file.
   Mandatory : NO
   Default   : -
```

```
    May be used to embed a changelog in the spec file.
    The refered file will be read and directly put after the %changelog
    section.
  CPACK_RPM_EXCLUDE_FROM_AUTO_FILELIST - list of path to be excluded.
    Mandatory : NO
    Default   : /etc /etc/init.d /usr /usr/share /usr/share/doc /usr/bin /usr/lib /usr/lib64 /usr/include
    May be used to exclude path (directories or files) from the auto-generated
    list of paths discovered by CPack RPM. The defaut value contains a reasonable
    set of values if the variable is not defined by the user. If the variable
    is defined by the user then CPackRPM will NOT any of the default path.
    If you want to add some path to the default list then you can use
    CPACK_RPM_EXCLUDE_FROM_AUTO_FILELIST_ADDITION variable.
  CPACK_RPM_EXCLUDE_FROM_AUTO_FILELIST_ADDITION - additional list of path to be excluded.
    Mandatory : NO
    Default   : -
    May be used to add more exclude path (directories or files) from the initial
    default list of excluded paths. See CPACK_RPM_EXCLUDE_FROM_AUTO_FILELIST.
```

- **`CPackWIX`**: CPack WiX generator specific options

  The following variables are specific to the installers built on Windows using WiX.

  ```
  CPACK_WIX_UPGRADE_GUID - Upgrade GUID (Product/@UpgradeCode)
  ```

  Will be automatically generated unless explicitly provided.

  It should be explicitly set to a constant generated gloabally unique identifier (GUID) to allow your installers to replace existing installations that use the same GUID.

  You may for example explicitly set this variable in your CMakeLists.txt to the value that has been generated per default. You should not use GUIDs that you did not generate yourself or which may belong to other projects.

  A GUID shall have the following fixed length syntax: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX

  ```
  (each X represents an uppercase hexadecimal digit)
  CPACK_WIX_PRODUCT_GUID - Product GUID (Product/@Id)
  ```

  Will be automatically generated unless explicitly provided.

  If explicitly provided this will set the Product Id of your installer.

  The installer will abort if it detects a pre-existing installation that uses the same GUID.

  The GUID shall use the syntax described for CPACK_WIX_UPGRADE_GUID.

  ```
  CPACK_WIX_LICENSE_RTF - RTF License File
  ```

  If CPACK_RESOURCE_FILE_LICENSE has an .rtf extension it is used as-is.

  If CPACK_RESOURCE_FILE_LICENSE has an .txt extension it is implicitly converted to RTF by the WiX Generator.

  With CPACK_WIX_LICENSE_RTF you can override the license file used by the WiX Generator in case CPACK_RESOURCE_FILE_LICENSE is in an unsupported format or the .txt -> .rtf conversion does not work as expected.

  CPACK_WIX_PRODUCT_ICON - The Icon shown next to the program name in Add/Remove programs.

  If set, this icon is used in place of the default icon.

  CPACK_WIX_UI_BANNER - The bitmap will appear at the top of all installer pages other than the welcome and completion dialogs.

  If set, this image will replace the default banner image.

  This image must be 493 by 58 pixels.

  CPACK_WIX_UI_DIALOG - Background bitmap used on the welcome and completion dialogs.

  If this variable is set, the installer will replace the default dialog image.

  This image must be 493 by 312 pixels.

CPACK_WIX_PROGRAM_MENU_FOLDER - Start menu folder name for launcher.

If this variable is not set, it will be initialized with CPACK_PACKAGE_NAME

CPACK_WIX_CULTURES - Language(s) of the installer

Languages are compiled into the WixUI extension library. To use them, simply provide the name of the culture. If you specify more than one culture identifier in a comma or semicolon delimited list, the first one that is found will be used. You can find a list of supported languages at: http://wix.sourceforge.net/manual-wix3/WixUI_localization.htm

CPACK_WIX_TEMPLATE - Template file for WiX generation

If this variable is set, the specified template will be used to generate the WiX wxs file. This should be used if further customization of the output is required.

If this variable is not set, the default MSI template included with CMake will be used.

- **CTest**: Configure a project for testing with CTest/CDash

  Include this module in the top CMakeLists.txt file of a project to enable testing with CTest and dashboard submissions to CDash:

  ```
  project(MyProject)
  ...
  include(CTest)
  ```

  The module automatically creates a BUILD_TESTING option that selects whether to enable testing support (ON by default). After including the module, use code like

  ```
  if(BUILD_TESTING)
    # ... CMake code to create tests ...
  endif()
  ```

  to creating tests when testing is enabled.

  To enable submissions to a CDash server, create a CTestConfig.cmake file at the top of the project with content such as

  ```
  set(CTEST_PROJECT_NAME "MyProject")
  set(CTEST_NIGHTLY_START_TIME "01:00:00 UTC")
  set(CTEST_DROP_METHOD "http")
  set(CTEST_DROP_SITE "my.cdash.org")
  set(CTEST_DROP_LOCATION "/submit.php?project=MyProject")
  set(CTEST_DROP_SITE_CDASH TRUE)
  ```

  (the CDash server can provide the file to a project administrator who configures 'MyProject'). Settings in the config file are shared by both this CTest module and the CTest command-line tool's dashboard script mode (ctest -S).

  While building a project for submission to CDash, CTest scans the build output for errors and warnings and reports them with surrounding context from the build log. This generic approach works for all build tools, but does not give details about the command invocation that produced a given problem. One may get more detailed reports by adding

  ```
  set(CTEST_USE_LAUNCHERS 1)
  ```

  to the CTestConfig.cmake file. When this option is enabled, the CTest module tells CMake's Makefile generators to invoke every command in the generated build system through a CTest launcher program. (Currently the CTEST_USE_LAUNCHERS option is ignored on non-Makefile generators.) During a manual build each launcher transparently runs the command it wraps. During a CTest-driven build for submission to CDash each launcher reports detailed information when its command fails or warns. (Setting CTEST_USE_LAUNCHERS in CTestConfig.cmake is convenient, but also adds the launcher overhead even for manual builds. One may instead set it in a CTest dashboard script and add it to the CMake cache for the build tree.)

- **CTestScriptMode**:

  This file is read by ctest in script mode (-S)

- **CTestUseLaunchers**: Set the RULE_LAUNCH_* global properties when CTEST_USE_LAUNCHERS is on.

  CTestUseLaunchers is automatically included when you include(CTest). However, it is split out into its own module file so projects can use the CTEST_USE_LAUNCHERS functionality independently.

  To use launchers, set CTEST_USE_LAUNCHERS to ON in a ctest -S dashboard script, and then also set it in the cache of the configured project. Both cmake and ctest need to know the value of it for the launchers to work properly. CMake needs to know in order to generate proper build rules, and ctest, in order to produce the proper error and warning analysis.

  For convenience, you may set the ENV variable CTEST_USE_LAUNCHERS_DEFAULT in your ctest -S script, too. Then, as long as your CMakeLists uses include(CTest) or include(CTestUseLaunchers), it will use the value of the ENV variable to initialize a CTEST_USE_LAUNCHERS cache variable. This cache variable initialization only occurs if CTEST_USE_LAUNCHERS is not already defined.

- **CheckCCompilerFlag**: Check whether the C compiler supports a given flag.

  CHECK_C_COMPILER_FLAG(<flag> <var>)

```
<flag> - the compiler flag
<var>  - variable to store the result
```

This internally calls the check_c_source_compiles macro and sets CMAKE_REQUIRED_DEFINITIONS to <flag>. See help for CheckCSourceCompiles for a listing of variables that can otherwise modify the build. The result only tells that the compiler does not give an error message when it encounters the flag. If the flag has any effect or even a specific one is beyond the scope of this module.

- **`CheckCSourceCompiles`**: Check if given C source compiles and links into an executable

  CHECK_C_SOURCE_COMPILES(<code> <var> [FAIL_REGEX <fail-regex>])

  ```
  <code>       - source code to try to compile, must define 'main'
  <var>        - variable to store whether the source code compiled
  <fail-regex> - fail if test output matches this regex
  ```

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
  CMAKE_REQUIRED_LIBRARIES = list of libraries to link
  ```

- **`CheckCSourceRuns`**: Check if the given C source code compiles and runs.

  CHECK_C_SOURCE_RUNS(<code> <var>)

  ```
  <code>   - source code to try to compile
  <var>    - variable to store the result
             (1 for success, empty for failure)
  ```

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
  CMAKE_REQUIRED_LIBRARIES = list of libraries to link
  ```

- **`CheckCXXCompilerFlag`**: Check whether the CXX compiler supports a given flag.

  CHECK_CXX_COMPILER_FLAG(<flag> <var>)

  ```
  <flag> - the compiler flag
  <var>  - variable to store the result
  ```

  This internally calls the check_cxx_source_compiles macro and sets CMAKE_REQUIRED_DEFINITIONS to <flag>. See help for CheckCXXSourceCompiles for a listing of variables that can otherwise modify the build. The result only tells that the compiler does not give an error message when it encounters the flag. If the flag has any effect or even a specific one is beyond the scope of this module.

- **`CheckCXXSourceCompiles`**: Check if given C++ source compiles and links into an executable

  CHECK_CXX_SOURCE_COMPILES(<code> <var> [FAIL_REGEX <fail-regex>])

  ```
  <code>       - source code to try to compile, must define 'main'
  <var>        - variable to store whether the source code compiled
  <fail-regex> - fail if test output matches this regex
  ```

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
  CMAKE_REQUIRED_LIBRARIES = list of libraries to link
  ```

- **`CheckCXXSourceRuns`**: Check if the given C++ source code compiles and runs.

  CHECK_CXX_SOURCE_RUNS(<code> <var>)

  ```
  <code>   - source code to try to compile
  <var>    - variable to store the result
             (1 for success, empty for failure)
  ```

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
  CMAKE_REQUIRED_LIBRARIES = list of libraries to link
  ```

- **`CheckCXXSymbolExists`**: Check if a symbol exists as a function, variable, or macro in C++

  CHECK_CXX_SYMBOL_EXISTS(<symbol> <files> <variable>)

Check that the <symbol> is available after including given header <files> and store the result in a <variable>. Specify the list of files in one argument as a semicolon-separated list. CHECK_CXX_SYMBOL_EXISTS() can be used to check in C++ files, as opposed to CHECK_SYMBOL_EXISTS(), which works only for C.

If the header files define the symbol as a macro it is considered available and assumed to work. If the header files declare the symbol as a function or variable then the symbol must also be available for linking. If the symbol is a type or enum value it will not be recognized (consider using CheckTypeSize or CheckCSourceCompiles).

The following variables may be set before calling this macro to modify the way the check is run:

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES = list of include directories
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

- **`CheckFortranFunctionExists`**: macro which checks if the Fortran function exists

  CHECK_FORTRAN_FUNCTION_EXISTS(FUNCTION VARIABLE)

  ```
  FUNCTION - the name of the Fortran function
  VARIABLE - variable to store the result
  ```

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_LIBRARIES = list of libraries to link
  ```

- **`CheckFunctionExists`**: Check if a C function can be linked

  CHECK_FUNCTION_EXISTS(<function> <variable>)

  Check that the <function> is provided by libraries on the system and store the result in a <variable>. This does not verify that any system header file declares the function, only that it can be found at link time (consider using CheckSymbolExists).

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
  CMAKE_REQUIRED_LIBRARIES = list of libraries to link
  ```

- **`CheckIncludeFile`**: macro which checks the include file exists.

  CHECK_INCLUDE_FILE(INCLUDE VARIABLE)

  ```
  INCLUDE  - name of include file
  VARIABLE - variable to return result
  ```

  an optional third argument is the CFlags to add to the compile line or you can use CMAKE_REQUIRED_FLAGS

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
  ```

- **`CheckIncludeFileCXX`**: Check if the include file exists.

  ```
  CHECK_INCLUDE_FILE_CXX(INCLUDE VARIABLE)
  ```

  ```
  INCLUDE  - name of include file
  VARIABLE - variable to return result
  ```

  An optional third argument is the CFlags to add to the compile line or you can use CMAKE_REQUIRED_FLAGS.

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
  ```

- **`CheckIncludeFiles`**: Check if the files can be included

CHECK_INCLUDE_FILES(INCLUDE VARIABLE)

```
INCLUDE  - list of files to include
VARIABLE - variable to return result
```

The following variables may be set before calling this macro to modify the way the check is run:

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES = list of include directories
```

- **`CheckLanguage`**: Check if a language can be enabled

  Usage:

  ```
  check_language(<lang>)
  ```

  where <lang> is a language that may be passed to enable_language() such as "Fortran". If CMAKE_<lang>_COMPILER is already defined the check does nothing. Otherwise it tries enabling the language in a test project. The result is cached in CMAKE_<lang>_COMPILER as the compiler that was found, or NOTFOUND if the language cannot be enabled.

  Example:

  ```
  check_language(Fortran)
  if(CMAKE_Fortran_COMPILER)
    enable_language(Fortran)
  else()
    message(STATUS "No Fortran support")
  endif()
  ```

- **`CheckLibraryExists`**: Check if the function exists.

  CHECK_LIBRARY_EXISTS (LIBRARY FUNCTION LOCATION VARIABLE)

  ```
  LIBRARY  - the name of the library you are looking for
  FUNCTION - the name of the function
  LOCATION - location where the library should be found
  VARIABLE - variable to store the result
  ```

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_LIBRARIES = list of libraries to link
  ```

- **`CheckPrototypeDefinition`**: Check if the protoype we expect is correct.

  check_prototype_definition(FUNCTION PROTOTYPE RETURN HEADER VARIABLE)

  ```
  FUNCTION - The name of the function (used to check if prototype exists)
  PROTOTYPE- The prototype to check.
  RETURN - The return value of the function.
  HEADER - The header files required.
  VARIABLE - The variable to store the result.
  ```

  Example:

  ```
  check_prototype_definition(getpwent_r
   "struct passwd *getpwent_r(struct passwd *src, char *buf, int buflen)"
   "NULL"
   "unistd.h;pwd.h"
   SOLARIS_GETPWENT_R)
  ```

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
  CMAKE_REQUIRED_LIBRARIES = list of libraries to link
  ```

- **`CheckStructHasMember`**: Check if the given struct or class has the specified member variable

  CHECK_STRUCT_HAS_MEMBER (STRUCT MEMBER HEADER VARIABLE)

  ```
  STRUCT - the name of the struct or class you are interested in
  MEMBER - the member which existence you want to check
  HEADER - the header(s) where the prototype should be declared
  VARIABLE - variable to store the result
  ```

The following variables may be set before calling this macro to modify the way the check is run:

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES = list of include directories
```

Example: CHECK_STRUCT_HAS_MEMBER("struct timeval" tv_sec sys/select.h HAVE_TIMEVAL_TV_SEC)

- **`CheckSymbolExists`**: Check if a symbol exists as a function, variable, or macro

  CHECK_SYMBOL_EXISTS(<symbol> <files> <variable>)

  Check that the <symbol> is available after including given header <files> and store the result in a <variable>. Specify the list of files in one argument as a semicolon-separated list.

  If the header files define the symbol as a macro it is considered available and assumed to work. If the header files declare the symbol as a function or variable then the symbol must also be available for linking. If the symbol is a type or enum value it will not be recognized (consider using CheckTypeSize or CheckCSourceCompiles). If the check needs to be done in C++, consider using CHECK_CXX_SYMBOL_EXISTS(), which does the same as CHECK_SYMBOL_EXISTS(), but in C++.

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
  CMAKE_REQUIRED_LIBRARIES = list of libraries to link
  ```

- **`CheckTypeSize`**: Check sizeof a type

  ```
  CHECK_TYPE_SIZE(TYPE VARIABLE [BUILTIN_TYPES_ONLY])
  ```

  Check if the type exists and determine its size. On return, "HAVE_${VARIABLE}" holds the existence of the type, and "${VARIABLE}" holds one of the following:

  ```
  <size> = type has non-zero size <size>
  "0"    = type has arch-dependent size (see below)
  ""     = type does not exist
  ```

  Furthermore, the variable "${VARIABLE}_CODE" holds C preprocessor code to define the macro "${VARIABLE}" to the size of the type, or leave the macro undefined if the type does not exist.

  The variable "${VARIABLE}" may be "0" when CMAKE_OSX_ARCHITECTURES has multiple architectures for building OS X universal binaries. This indicates that the type size varies across architectures. In this case "${VARIABLE}_CODE" contains C preprocessor tests mapping from each architecture macro to the corresponding type size. The list of architecture macros is stored in "${VARIABLE}_KEYS", and the value for each key is stored in "${VARIABLE}-${KEY}".

  If the BUILTIN_TYPES_ONLY option is not given, the macro checks for headers <sys/types.h>, <stdint.h>, and <stddef.h>, and saves results in HAVE_SYS_TYPES_H, HAVE_STDINT_H, and HAVE_STDDEF_H. The type size check automatically includes the available headers, thus supporting checks of types defined in the headers.

  Despite the name of the macro you may use it to check the size of more complex expressions, too. To check e.g. for the size of a struct member you can do something like this:

  ```
  check_type_size("((struct something*)0)->member" SIZEOF_MEMBER)
  ```

  The following variables may be set before calling this macro to modify the way the check is run:

  ```
  CMAKE_REQUIRED_FLAGS = string of compile command line flags
  CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
  CMAKE_REQUIRED_LIBRARIES = list of libraries to link
  CMAKE_EXTRA_INCLUDE_FILES = list of extra headers to include
  ```

- **`CheckVariableExists`**: Check if the variable exists.

  ```
  CHECK_VARIABLE_EXISTS(VAR VARIABLE)



  VAR      - the name of the variable
  VARIABLE - variable to store the result
  ```

  This macro is only for C variables.

  The following variables may be set before calling this macro to modify the way the check is run:

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define (-DFOO=bar)
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

- **`Dart`**: Configure a project for testing with CTest or old Dart Tcl Client

  This file is the backwards-compatibility version of the CTest module. It supports using the old Dart 1 Tcl client for driving dashboard submissions as well as testing with CTest. This module should be included in the CMakeLists.txt file at the top of a project. Typical usage:

  ```
  include(Dart)
  if(BUILD_TESTING)
    # ... testing related CMake code ...
  endif()
  ```

  The BUILD_TESTING option is created by the Dart module to determine whether testing support should be enabled. The default is ON.

- **`DeployQt4`**: Functions to help assemble a standalone Qt4 executable.

  A collection of CMake utility functions useful for deploying Qt4 executables.

  The following functions are provided by this module:

  ```
  write_qt4_conf
  resolve_qt4_paths
  fixup_qt4_executable
  install_qt4_plugin_path
  install_qt4_plugin
  install_qt4_executable
  ```

  Requires CMake 2.6 or greater because it uses function and PARENT_SCOPE. Also depends on BundleUtilities.cmake.

  ```
  WRITE_QT4_CONF(<qt_conf_dir> <qt_conf_contents>)
  ```

  Writes a qt.conf file with the <qt_conf_contents> into <qt_conf_dir>.

  ```
  RESOLVE_QT4_PATHS(<paths_var> [<executable_path>])
  ```

  Loop through <paths_var> list and if any don't exist resolve them relative to the <executable_path> (if supplied) or the CMAKE_INSTALL_PREFIX.

  ```
  FIXUP_QT4_EXECUTABLE(<executable> [<qtplugins> <libs> <dirs> <plugins_dir> <request_qt_conf>])
  ```

  Copies Qt plugins, writes a Qt configuration file (if needed) and fixes up a Qt4 executable using BundleUtilities so it is standalone and can be drag-and-drop copied to another machine as long as all of the system libraries are compatible.

  <executable> should point to the executable to be fixed-up.

  <qtplugins> should contain a list of the names or paths of any Qt plugins to be installed.

  <libs> will be passed to BundleUtilities and should be a list of any already installed plugins, libraries or executables to also be fixed-up.

  <dirs> will be passed to BundleUtilities and should contain and directories to be searched to find library dependencies.

  <plugins_dir> allows an custom plugins directory to be used.

  <request_qt_conf> will force a qt.conf file to be written even if not needed.

  ```
  INSTALL_QT4_PLUGIN_PATH(plugin executable copy installed_plugin_path_var <plugins_dir> <component> <configurations>)
  ```

  Install (or copy) a resolved <plugin> to the default plugins directory (or <plugins_dir>) relative to <executable> and store the result in <installed_plugin_path_var>.

  If <copy> is set to TRUE then the plugins will be copied rather than installed. This is to allow this module to be used at CMake time rather than install time.

  If <component> is set then anything installed will use this COMPONENT.

  ```
  INSTALL_QT4_PLUGIN(plugin executable copy installed_plugin_path_var <plugins_dir> <component>)
  ```

  Install (or copy) an unresolved <plugin> to the default plugins directory (or <plugins_dir>) relative to <executable> and store the result in <installed_plugin_path_var>. See documentation of INSTALL_QT4_PLUGIN_PATH.

  ```
  INSTALL_QT4_EXECUTABLE(<executable> [<qtplugins> <libs> <dirs> <plugins_dir> <request_qt_conf> <component>])
  ```

  Installs Qt plugins, writes a Qt configuration file (if needed) and fixes up a Qt4 executable using BundleUtilities so it is standalone and can be drag-and-drop copied to another machine as long as all of the system libraries are compatible. The executable will be fixed-up at install time. <component> is the COMPONENT used for bundle fixup and plugin installation. See documentation of FIXUP_QT4_BUNDLE.

- **`Documentation`**: DocumentationVTK.cmake

  This file provides support for the VTK documentation framework. It relies on several tools (Doxygen, Perl, etc).

- **`ExternalData`**: Manage data files stored outside source tree

  Use this module to unambiguously reference data files stored outside the source tree and fetch them at build time from arbitrary local and remote content-addressed locations. Functions provided by this module recognize arguments with the syntax "DATA{<name>}" as references to external data, replace them with full paths to local copies of those data, and create build rules to fetch and update the local copies.

  The DATA{} syntax is literal and the <name> is a full or relative path within the source tree. The source tree must contain either a real data file at <name> or a "content link" at <name><ext> containing a hash of the real file using a hash algorithm corresponding to <ext>. For example, the argument "DATA{img.png}" may be satisfied by either a real "img.png" file in the current source directory or a "img.png.md5" file containing its MD5 sum.

  The 'ExternalData_Expand_Arguments' function evaluates DATA{} references in its arguments and constructs a new list of arguments:

```
ExternalData_Expand_Arguments(
  <target>   # Name of data management target
  <outVar>   # Output variable
  [args...]  # Input arguments, DATA{} allowed
  )
```

  It replaces each DATA{} reference in an argument with the full path of a real data file on disk that will exist after the <target> builds.

  The 'ExternalData_Add_Test' function wraps around the CMake add_test() command but supports DATA{} references in its arguments:

```
ExternalData_Add_Test(
  <target>   # Name of data management target
  ...        # Arguments of add_test(), DATA{} allowed
  )
```

  It passes its arguments through ExternalData_Expand_Arguments and then invokes add_test() using the results.

  The 'ExternalData_Add_Target' function creates a custom target to manage local instances of data files stored externally:

```
ExternalData_Add_Target(
  <target>   # Name of data management target
  )
```

  It creates custom commands in the target as necessary to make data files available for each DATA{} reference previously evaluated by other functions provided by this module. A list of URL templates must be provided in the variable ExternalData_URL_TEMPLATES using the placeholders "%(algo)" and "%(hash)" in each template. Data fetch rules try each URL template in order by substituting the hash algorithm name for "%(algo)" and the hash value for "%(hash)".

  The following hash algorithms are supported:

```
%(algo)     <ext>     Description
-------     -----     -----------
MD5         .md5      Message-Digest Algorithm 5, RFC 1321
SHA1        .sha1     US Secure Hash Algorithm 1, RFC 3174
SHA224      .sha224   US Secure Hash Algorithms, RFC 4634
SHA256      .sha256   US Secure Hash Algorithms, RFC 4634
SHA384      .sha384   US Secure Hash Algorithms, RFC 4634
SHA512      .sha512   US Secure Hash Algorithms, RFC 4634
```

  Note that the hashes are used only for unique data identification and download verification. This is not security software.

  Example usage:

```
include(ExternalData)
set(ExternalData_URL_TEMPLATES "file:///local/%(algo)/%(hash)"
                               "http://data.org/%(algo)/%(hash)")
ExternalData_Add_Test(MyData
  NAME MyTest
  COMMAND MyExe DATA{MyInput.png}
  )
ExternalData_Add_Target(MyData)
```

  When test "MyTest" runs the "DATA{MyInput.png}" argument will be replaced by the full path to a real instance of the data file "MyInput.png" on disk. If the source tree contains a content link such as "MyInput.png.md5" then the "MyData" target creates a real "MyInput.png" in the build tree.

  The DATA{} syntax can be told to fetch a file series using the form "DATA{<name>,:}", where the ":" is literal. If the source tree contains a group of files or content links named like a series then a reference to one member adds rules to fetch all of them. Although all members of a series are fetched, only the file originally named by the DATA{} argument is substituted for it. The default configuration recognizes file series names ending with "#.ext", "_#.ext", ".#.ext", or "-#.ext" where "#" is a sequence of decimal digits and ".ext" is any single extension. Configure it with a regex that parses <number> and <suffix> parts from the end of <name>:

```
  ExternalData_SERIES_PARSE = regex of the form (<number>)(<suffix>)$
```

For more complicated cases set:

```
  ExternalData_SERIES_PARSE = regex with at least two () groups
  ExternalData_SERIES_PARSE_PREFIX = <prefix> regex group number, if any
  ExternalData_SERIES_PARSE_NUMBER = <number> regex group number
  ExternalData_SERIES_PARSE_SUFFIX = <suffix> regex group number
```

Configure series number matching with a regex that matches the <number> part of series members named <prefix><number><suffix>:

```
  ExternalData_SERIES_MATCH = regex matching <number> in all series members
```

Note that the <suffix> of a series does not include a hash-algorithm extension.

The DATA{} syntax can alternatively match files associated with the named file and contained in the same directory. Associated files may be specified by options using the syntax DATA{<name>,<opt1>,<opt2>,...}. Each option may specify one file by name or specify a regular expression to match file names using the syntax REGEX:<regex>. For example, the arguments

```
  DATA{MyData/MyInput.mhd,MyInput.img}                    # File pair
  DATA{MyData/MyFrames00.png,REGEX:MyFrames[0-9]+\\.png} # Series
```

will pass MyInput.mha and MyFrames00.png on the command line but ensure that the associated files are present next to them.

The DATA{} syntax may reference a directory using a trailing slash and a list of associated files. The form DATA{<name>/,<opt1>,<opt2>,...} adds rules to fetch any files in the directory that match one of the associated file options. For example, the argument DATA{MyDataDir/,REGEX:.*} will pass the full path to a MyDataDir directory on the command line and ensure that the directory contains files corresponding to every file or content link in the MyDataDir source directory.

The variable ExternalData_LINK_CONTENT may be set to the name of a supported hash algorithm to enable automatic conversion of real data files referenced by the DATA{} syntax into content links. For each such <file> a content link named "<file><ext>" is created. The original file is renamed to the form ".ExternalData_<algo>_<hash>" to stage it for future transmission to one of the locations in the list of URL templates (by means outside the scope of this module). The data fetch rule created for the content link will use the staged object if it cannot be found using any URL template.

The variable ExternalData_OBJECT_STORES may be set to a list of local directories that store objects using the layout <dir>/%(algo)/%(hash). These directories will be searched first for a needed object. If the object is not available in any store then it will be fetched remotely using the URL templates and added to the first local store listed. If no stores are specified the default is a location inside the build tree.

The variable ExternalData_SOURCE_ROOT may be set to the highest source directory containing any path named by a DATA{} reference. The default is CMAKE_SOURCE_DIR. ExternalData_SOURCE_ROOT and CMAKE_SOURCE_DIR must refer to directories within a single source distribution (e.g. they come together in one tarball).

The variable ExternalData_BINARY_ROOT may be set to the directory to hold the real data files named by expanded DATA{} references. The default is CMAKE_BINARY_DIR. The directory layout will mirror that of content links under ExternalData_SOURCE_ROOT.

Variables ExternalData_TIMEOUT_INACTIVITY and ExternalData_TIMEOUT_ABSOLUTE set the download inactivity and absolute timeouts, in seconds. The defaults are 60 seconds and 300 seconds, respectively. Set either timeout to 0 seconds to disable enforcement.

- **`ExternalProject`**: Create custom targets to build projects in external trees

  The 'ExternalProject_Add' function creates a custom target to drive download, update/patch, configure, build, install and test steps of an external project:

```
  ExternalProject_Add(<name>    # Name for custom target
    [DEPENDS projects...]       # Targets on which the project depends
    [PREFIX dir]                # Root dir for entire project
    [LIST_SEPARATOR sep]        # Sep to be replaced by ; in cmd lines
    [TMP_DIR dir]               # Directory to store temporary files
    [STAMP_DIR dir]             # Directory to store step timestamps
   #--Download step--------------
    [DOWNLOAD_NAME fname]       # File name to store (if not end of URL)
    [DOWNLOAD_DIR dir]          # Directory to store downloaded files
    [DOWNLOAD_COMMAND cmd...]   # Command to download source tree
    [CVS_REPOSITORY cvsroot]    # CVSROOT of CVS repository
    [CVS_MODULE mod]            # Module to checkout from CVS repo
    [CVS_TAG tag]               # Tag to checkout from CVS repo
    [SVN_REPOSITORY url]        # URL of Subversion repo
    [SVN_REVISION rev]          # Revision to checkout from Subversion repo
    [SVN_USERNAME john ]        # Username for Subversion checkout and update
    [SVN_PASSWORD doe ]         # Password for Subversion checkout and update
    [SVN_TRUST_CERT 1 ]         # Trust the Subversion server site certificate
```

```
 [GIT_REPOSITORY url]        # URL of git repo
 [GIT_TAG tag]               # Git branch name, commit id or tag
 [HG_REPOSITORY url]         # URL of mercurial repo
 [HG_TAG tag]                # Mercurial branch name, commit id or tag
 [URL /.../src.tgz]          # Full path or URL of source
 [URL_HASH ALGO=value]       # Hash of file at URL
 [URL_MD5 md5]               # Equivalent to URL_HASH MD5=md5
 [TLS_VERIFY bool]           # Should certificate for https be checked
 [TLS_CAINFO file]           # Path to a certificate authority file
 [TIMEOUT seconds]           # Time allowed for file download operations
#--Update/Patch step----------
 [UPDATE_COMMAND cmd...]     # Source work-tree update command
 [PATCH_COMMAND cmd...]      # Command to patch downloaded source
#--Configure step-------------
 [SOURCE_DIR dir]            # Source dir to be used for build
 [CONFIGURE_COMMAND cmd...]  # Build tree configuration command
 [CMAKE_COMMAND /.../cmake]  # Specify alternative cmake executable
 [CMAKE_GENERATOR gen]       # Specify generator for native build
 [CMAKE_GENERATOR_TOOLSET t] # Generator-specific toolset name
 [CMAKE_ARGS args...]        # Arguments to CMake command line
 [CMAKE_CACHE_ARGS args...]  # Initial cache arguments, of the form -Dvar:string=on
#--Build step-----------------
 [BINARY_DIR dir]            # Specify build dir location
 [BUILD_COMMAND cmd...]      # Command to drive the native build
 [BUILD_IN_SOURCE 1]         # Use source dir for build dir
#--Install step---------------
 [INSTALL_DIR dir]           # Installation prefix
 [INSTALL_COMMAND cmd...]    # Command to drive install after build
#--Test step------------------
 [TEST_BEFORE_INSTALL 1]     # Add test step executed before install step
 [TEST_AFTER_INSTALL 1]      # Add test step executed after install step
 [TEST_COMMAND cmd...]       # Command to drive test
#--Output logging-------------
 [LOG_DOWNLOAD 1]            # Wrap download in script to log output
 [LOG_UPDATE 1]              # Wrap update in script to log output
 [LOG_CONFIGURE 1]           # Wrap configure in script to log output
 [LOG_BUILD 1]               # Wrap build in script to log output
 [LOG_TEST 1]                # Wrap test in script to log output
 [LOG_INSTALL 1]             # Wrap install in script to log output
#--Custom targets-------------
 [STEP_TARGETS st1 st2 ...]  # Generate custom targets for these steps
 )
```

The *_DIR options specify directories for the project, with default directories computed as follows. If the PREFIX option is given to ExternalProject_Add() or the EP_PREFIX directory property is set, then an external project is built and installed under the specified prefix:

```
TMP_DIR      = <prefix>/tmp
STAMP_DIR    = <prefix>/src/<name>-stamp
DOWNLOAD_DIR = <prefix>/src
SOURCE_DIR   = <prefix>/src/<name>
BINARY_DIR   = <prefix>/src/<name>-build
INSTALL_DIR  = <prefix>
```

Otherwise, if the EP_BASE directory property is set then components of an external project are stored under the specified base:

```
TMP_DIR      = <base>/tmp/<name>
STAMP_DIR    = <base>/Stamp/<name>
DOWNLOAD_DIR = <base>/Download/<name>
SOURCE_DIR   = <base>/Source/<name>
BINARY_DIR   = <base>/Build/<name>
INSTALL_DIR  = <base>/Install/<name>
```

If no PREFIX, EP_PREFIX, or EP_BASE is specified then the default is to set PREFIX to "<name>-prefix". Relative paths are interpreted with respect to the build directory corresponding to the source directory in which ExternalProject_Add is invoked.

If SOURCE_DIR is explicitly set to an existing directory the project will be built from it. Otherwise a download step must be specified using one of the DOWNLOAD_COMMAND, CVS_*, SVN_*, or URL options. The URL option may refer locally to a directory or source tarball, or refer to a remote tarball (e.g. http://.../src.tgz).

The 'ExternalProject_Add_Step' function adds a custom step to an external project:

```
ExternalProject_Add_Step(<name> <step> # Names of project and custom step
```

```
  [COMMAND cmd...]        # Command line invoked by this step
  [COMMENT "text..."]     # Text printed when step executes
  [DEPENDEES steps...]    # Steps on which this step depends
  [DEPENDERS steps...]    # Steps that depend on this step
  [DEPENDS files...]      # Files on which this step depends
  [ALWAYS 1]              # No stamp file, step always runs
  [WORKING_DIRECTORY dir] # Working directory for command
  [LOG 1]                 # Wrap step in script to log output
  )
```

The command line, comment, and working directory of every standard and custom step is processed to replace tokens <SOURCE_DIR>, <BINARY_DIR>, <INSTALL_DIR>, and <TMP_DIR> with corresponding property values.

Any builtin step that specifies a "<step>_COMMAND cmd..." or custom step that specifies a "COMMAND cmd..." may specify additional command lines using the form "COMMAND cmd...". At build time the commands will be executed in order and aborted if any one fails. For example:

```
... BUILD_COMMAND make COMMAND echo done ...
```

specifies to run "make" and then "echo done" during the build step. Whether the current working directory is preserved between commands is not defined. Behavior of shell operators like "&&" is not defined.

The 'ExternalProject_Get_Property' function retrieves external project target properties:

```
ExternalProject_Get_Property(<name> [prop1 [prop2 [...]]])
```

It stores property values in variables of the same name. Property names correspond to the keyword argument names of 'ExternalProject_Add'.

The 'ExternalProject_Add_StepTargets' function generates custom targets for the steps listed:

```
ExternalProject_Add_StepTargets(<name> [step1 [step2 [...]]])
```

If STEP_TARGETS is set then ExternalProject_Add_StepTargets is automatically called at the end of matching calls to ExternalProject_Add_Step. Pass STEP_TARGETS explicitly to individual ExternalProject_Add calls, or implicitly to all ExternalProject_Add calls by setting the directory property EP_STEP_TARGETS.

If STEP_TARGETS is not set, clients may still manually call ExternalProject_Add_StepTargets after calling ExternalProject_Add or ExternalProject_Add_Step.

This functionality is provided to make it easy to drive the steps independently of each other by specifying targets on build command lines. For example, you may be submitting to a sub-project based dashboard, where you want to drive the configure portion of the build, then submit to the dashboard, followed by the build portion, followed by tests. If you invoke a custom target that depends on a step halfway through the step dependency chain, then all the previous steps will also run to ensure everything is up to date.

For example, to drive configure, build and test steps independently for each ExternalProject_Add call in your project, write the following line prior to any ExternalProject_Add calls in your CMakeLists file:

```
set_property(DIRECTORY PROPERTY EP_STEP_TARGETS configure build test)
```

- **`FeatureSummary`**: Macros for generating a summary of enabled/disabled features

  This module provides the macros feature_summary(), set_package_properties() and add_feature_info(). For compatibility it also still provides set_package_info(), set_feature_info(), print_enabled_features() and print_disabled_features().

  These macros can be used to generate a summary of enabled and disabled packages and/or feature for a build tree:

  ```
  -- The following OPTIONAL packages have been found:
  LibXml2 (required version >= 2.4) , XML processing library. , <http://xmlsoft.org>
     * Enables HTML-import in MyWordProcessor
     * Enables odt-export in MyWordProcessor
  PNG , A PNG image library. , <http://www.libpng.org/pub/png/>
     * Enables saving screenshots
  -- The following OPTIONAL packages have not been found:
  Lua51 , The Lua scripting language. , <http://www.lua.org>
     * Enables macros in MyWordProcessor
  Foo , Foo provides cool stuff.



  FEATURE_SUMMARY( [FILENAME <file>]
                   [APPEND]
                   [VAR <variable_name>]
                   [INCLUDE_QUIET_PACKAGES]
  ```

```
                [FATAL_ON_MISSING_REQUIRED_PACKAGES]
                [DESCRIPTION "Found packages:"]
                WHAT (ALL | PACKAGES_FOUND | PACKAGES_NOT_FOUND
                     | ENABLED_FEATURES | DISABLED_FEATURES]
              )
```

The FEATURE_SUMMARY() macro can be used to print information about enabled or disabled packages or features of a project. By default, only the names of the features/packages will be printed and their required version when one was specified. Use SET_PACKAGE_PROPERTIES() to add more useful information, like e.g. a download URL for the respective package or their purpose in the project.

The WHAT option is the only mandatory option. Here you specify what information will be printed:

```
ALL: print everything
ENABLED_FEATURES: the list of all features which are enabled
DISABLED_FEATURES: the list of all features which are disabled
PACKAGES_FOUND: the list of all packages which have been found
PACKAGES_NOT_FOUND: the list of all packages which have not been found
OPTIONAL_PACKAGES_FOUND: only those packages which have been found which have the type OPTIONAL
OPTIONAL_PACKAGES_NOT_FOUND: only those packages which have not been found which have the type OPTIONAL
RECOMMENDED_PACKAGES_FOUND: only those packages which have been found which have the type RECOMMENDED
RECOMMENDED_PACKAGES_NOT_FOUND: only those packages which have not been found which have the type RECOMMENDED
REQUIRED_PACKAGES_FOUND: only those packages which have been found which have the type REQUIRED
REQUIRED_PACKAGES_NOT_FOUND: only those packages which have not been found which have the type REQUIRED
RUNTIME_PACKAGES_FOUND: only those packages which have been found which have the type RUNTIME
RUNTIME_PACKAGES_NOT_FOUND: only those packages which have not been found which have the type RUNTIME
```

If a FILENAME is given, the information is printed into this file. If APPEND is used, it is appended to this file, otherwise the file is overwritten if it already existed. If the VAR option is used, the information is "printed" into the specified variable. If FILENAME is not used, the information is printed to the terminal. Using the DESCRIPTION option a description or headline can be set which will be printed above the actual content. If INCLUDE_QUIET_PACKAGES is given, packages which have been searched with find_package(... QUIET) will also be listed. By default they are skipped. If FATAL_ON_MISSING_REQUIRED_PACKAGES is given, CMake will abort if a package which is marked as REQUIRED has not been found.

Example 1, append everything to a file:

```
feature_summary(WHAT ALL
                FILENAME ${CMAKE_BINARY_DIR}/all.log APPEND)
```

Example 2, print the enabled features into the variable enabledFeaturesText, including QUIET packages:

```
feature_summary(WHAT ENABLED_FEATURES
                INCLUDE_QUIET_PACKAGES
                DESCRIPTION "Enabled Features:"
                VAR enabledFeaturesText)
message(STATUS "${enabledFeaturesText}")
```

```
 SET_PACKAGE_PROPERTIES(<name> PROPERTIES [ URL <url> ]
                                          [ DESCRIPTION <description> ]
                                          [ TYPE (RUNTIME|OPTIONAL|RECOMMENDED|REQUIRED) ]
                                          [ PURPOSE <purpose> ]
                       )
```

Use this macro to set up information about the named package, which can then be displayed via FEATURE_SUMMARY(). This can be done either directly in the Find-module or in the project which uses the module after the find_package() call. The features for which information can be set are added automatically by the find_package() command.

URL: this should be the homepage of the package, or something similar. Ideally this is set already directly in the Find-module.

DESCRIPTION: A short description what that package is, at most one sentence. Ideally this is set already directly in the Find-module.

TYPE: What type of dependency has the using project on that package. Default is OPTIONAL. In this case it is a package which can be used by the project when available at buildtime, but it also work without. RECOMMENDED is similar to OPTIONAL, i.e. the project will build if the package is not present, but the functionality of the resulting binaries will be severly limited. If a REQUIRED package is not available at buildtime, the project may not even build. This can be combined with the

FATAL_ON_MISSING_REQUIRED_PACKAGES argument for feature_summary(). Last, a RUNTIME package is a package which is actually not used at all during the build, but which is required for actually running the resulting binaries. So if such a package is missing, the project can still be built, but it may not work later on. If set_package_properties() is called multiple times for the same package with different TYPEs, the TYPE is only changed to higher TYPEs (RUNTIME < OPTIONAL < RECOMMENDED < REQUIRED), lower TYPEs are ignored. The TYPE property is project-specific, so it cannot be set by the Find-module, but must be set in the project.

PURPOSE: This describes which features this package enables in the project, i.e. it tells the user what functionality he gets in the resulting binaries. If set_package_properties() is called multiple times for a package, all PURPOSE properties are appended to a list of purposes of the package in the project. As the TYPE property, also the PURPOSE property is project-specific, so it cannot be set by the Find-module, but must be set in the project.

Example for setting the info for a package:

```
find_package(LibXml2)
set_package_properties(LibXml2 PROPERTIES DESCRIPTION "A XML processing library."
                                          URL "http://xmlsoft.org/")


set_package_properties(LibXml2 PROPERTIES TYPE RECOMMENDED
                                          PURPOSE "Enables HTML-import in MyWordProcessor")
...
set_package_properties(LibXml2 PROPERTIES TYPE OPTIONAL
                                          PURPOSE "Enables odt-export in MyWordProcessor")



find_package(DBUS)
set_package_properties(DBUS PROPERTIES TYPE RUNTIME
                                          PURPOSE "Necessary to disable the screensaver during a presentation" )
```

```
 ADD_FEATURE_INFO(<name> <enabled> <description>)
```

Use this macro to add information about a feature with the given <name>. <enabled> contains whether this feature is enabled or not, <description> is a text describing the feature. The information can be displayed using feature_summary() for ENABLED_FEATURES and DISABLED_FEATURES respectively.

Example for setting the info for a feature:

```
option(WITH_FOO "Help for foo" ON)
add_feature_info(Foo WITH_FOO "The Foo feature provides very cool stuff.")
```

The following macros are provided for compatibility with previous CMake versions:

```
SET_PACKAGE_INFO(<name> <description> [<url> [<purpose>] ] )
```

Use this macro to set up information about the named package, which can then be displayed via FEATURE_SUMMARY(). This can be done either directly in the Find-module or in the project which uses the module after the find_package() call. The features for which information can be set are added automatically by the find_package() command.

```
PRINT_ENABLED_FEATURES()
```

Does the same as FEATURE_SUMMARY(WHAT ENABLED_FEATURES DESCRIPTION "Enabled features:")

```
PRINT_DISABLED_FEATURES()
```

Does the same as FEATURE_SUMMARY(WHAT DISABLED_FEATURES DESCRIPTION "Disabled features:")

```
SET_FEATURE_INFO(<name> <description> [<url>] )
```

Does the same as SET_PACKAGE_INFO(<name> <description> <url>)

- **`FindALSA`**: Find alsa

  Find the alsa libraries (asound)

```
This module defines the following variables:
   ALSA_FOUND       - True if ALSA_INCLUDE_DIR & ALSA_LIBRARY are found
   ALSA_LIBRARIES   - Set when ALSA_LIBRARY is found
   ALSA_INCLUDE_DIRS - Set when ALSA_INCLUDE_DIR is found



   ALSA_INCLUDE_DIR - where to find asoundlib.h, etc.
   ALSA_LIBRARY     - the asound library
```

```
ALSA_VERSION_STRING - the version of alsa found (since CMake 2.8.8)
```

- **`FindASPELL`**: Try to find ASPELL

  Once done this will define

  ```
  ASPELL_FOUND - system has ASPELL
  ASPELL_EXECUTABLE - the ASPELL executable
  ASPELL_INCLUDE_DIR - the ASPELL include directory
  ASPELL_LIBRARIES - The libraries needed to use ASPELL
  ASPELL_DEFINITIONS - Compiler switches required for using ASPELL
  ```

- **`FindAVIFile`**: Locate AVIFILE library and include paths

  AVIFILE (http://avifile.sourceforge.net/)is a set of libraries for i386 machines to use various AVI codecs. Support is limited beyond Linux. Windows provides native AVI support, and so doesn't need this library. This module defines

  ```
  AVIFILE_INCLUDE_DIR, where to find avifile.h , etc.
  AVIFILE_LIBRARIES, the libraries to link against
  AVIFILE_DEFINITIONS, definitions to use when compiling
  AVIFILE_FOUND, If false, don't try to use AVIFILE
  ```

- **`FindArmadillo`**: Find Armadillo

  Find the Armadillo C++ library

  Using Armadillo:

  ```
  find_package(Armadillo REQUIRED)
  include_directories(${ARMADILLO_INCLUDE_DIRS})
  add_executable(foo foo.cc)
  target_link_libraries(foo ${ARMADILLO_LIBRARIES})
  ```

  This module sets the following variables:

  ```
  ARMADILLO_FOUND - set to true if the library is found
  ARMADILLO_INCLUDE_DIRS - list of required include directories
  ARMADILLO_LIBRARIES - list of libraries to be linked
  ARMADILLO_VERSION_MAJOR - major version number
  ARMADILLO_VERSION_MINOR - minor version number
  ARMADILLO_VERSION_PATCH - patch version number
  ARMADILLO_VERSION_STRING - version number as a string (ex: "1.0.4")
  ARMADILLO_VERSION_NAME - name of the version (ex: "Antipodean Antileech")
  ```

- **`FindBISON`**: Find bison executable and provides macros to generate custom build rules

  The module defines the following variables:

  ```
  BISON_EXECUTABLE - path to the bison program
  BISON_VERSION - version of bison
  BISON_FOUND - true if the program was found
  ```

  The minimum required version of bison can be specified using the standard CMake syntax, e.g. find_package(BISON 2.1.3)

  If bison is found, the module defines the macros:

  ```
  BISON_TARGET(<Name> <YaccInput> <CodeOutput> [VERBOSE <file>]
               [COMPILE_FLAGS <string>])
  ```

  which will create a custom rule to generate a parser. <YaccInput> is the path to a yacc file. <CodeOutput> is the name of the source file generated by bison. A header file is also be generated, and contains the token list. If COMPILE_FLAGS option is specified, the next parameter is added in the bison command line. if VERBOSE option is specified, <file> is created and contains verbose descriptions of the grammar and parser. The macro defines a set of variables:

  ```
  BISON_${Name}_DEFINED - true is the macro ran successfully
  BISON_${Name}_INPUT - The input source file, an alias for <YaccInput>
  BISON_${Name}_OUTPUT_SOURCE - The source file generated by bison
  BISON_${Name}_OUTPUT_HEADER - The header file generated by bison
  BISON_${Name}_OUTPUTS - The sources files generated by bison
  BISON_${Name}_COMPILE_FLAGS - Options used in the bison command line


  ============================================================================
  Example:
  ```

```
 find_package(BISON)
 BISON_TARGET(MyParser parser.y ${CMAKE_CURRENT_BINARY_DIR}/parser.cpp)
 add_executable(Foo main.cpp ${BISON_MyParser_OUTPUTS})
=============================================================================
```

- **`FindBLAS`**: Find BLAS library

  This module finds an installed fortran library that implements the BLAS linear-algebra interface (see http://www.netlib.org/blas/). The list of libraries searched for is taken from the autoconf macro file, acx_blas.m4 (distributed at http://ac-archive.sourceforge.net/ac-archive/acx_blas.html).

  This module sets the following variables:

```
BLAS_FOUND - set to true if a library implementing the BLAS interface
  is found
BLAS_LINKER_FLAGS - uncached list of required linker flags (excluding -l
  and -L).
BLAS_LIBRARIES - uncached list of libraries (using full path name) to
  link against to use BLAS
BLAS95_LIBRARIES - uncached list of libraries (using full path name)
  to link against to use BLAS95 interface
BLAS95_FOUND - set to true if a library implementing the BLAS f95 interface
  is found
BLA_STATIC  if set on this determines what kind of linkage we do (static)
BLA_VENDOR  if set checks only the specified vendor, if not set checks
   all the possibilities
BLA_F95     if set on tries to find the f95 interfaces for BLAS/LAPACK
```

  C/CXX should be enabled to use Intel mkl

- **`FindBZip2`**: Try to find BZip2

  Once done this will define

```
BZIP2_FOUND - system has BZip2
BZIP2_INCLUDE_DIR - the BZip2 include directory
BZIP2_LIBRARIES - Link these to use BZip2
BZIP2_NEED_PREFIX - this is set if the functions are prefixed with BZ2_
BZIP2_VERSION_STRING - the version of BZip2 found (since CMake 2.8.8)
```

- **`FindBoost`**: Find Boost include dirs and libraries

  Use this module by invoking find_package with the form:

```
find_package(Boost
  [version] [EXACT]      # Minimum or EXACT version e.g. 1.36.0
  [REQUIRED]             # Fail with error if Boost is not found
  [COMPONENTS <libs>...] # Boost libraries by their canonical name
  )                      # e.g. "date_time" for "libboost_date_time"
```

  This module finds headers and requested component libraries OR a CMake package configuration file provided by a "Boost CMake" build. For the latter case skip to the "Boost CMake" section below. For the former case results are reported in variables:

```
Boost_FOUND            - True if headers and requested libraries were found
Boost_INCLUDE_DIRS     - Boost include directories
Boost_LIBRARY_DIRS     - Link directories for Boost libraries
Boost_LIBRARIES        - Boost component libraries to be linked
Boost_<C>_FOUND        - True if component <C> was found (<C> is upper-case)
Boost_<C>_LIBRARY      - Libraries to link for component <C> (may include
                         target_link_libraries debug/optimized keywords)
Boost_VERSION          - BOOST_VERSION value from boost/version.hpp
Boost_LIB_VERSION      - Version string appended to library filenames
Boost_MAJOR_VERSION    - Boost major version number (X in X.y.z)
Boost_MINOR_VERSION    - Boost minor version number (Y in x.Y.z)
Boost_SUBMINOR_VERSION - Boost subminor version number (Z in x.y.Z)
Boost_LIB_DIAGNOSTIC_DEFINITIONS (Windows)
                       - Pass to add_definitions() to have diagnostic
                         information about Boost's automatic linking
                         displayed during compilation
```

  This module reads hints about search locations from variables:

```
BOOST_ROOT             - Preferred installation prefix
 (or BOOSTROOT)
BOOST_INCLUDEDIR       - Preferred include directory e.g. <prefix>/include
```

```
  BOOST_LIBRARYDIR       - Preferred library directory e.g. <prefix>/lib
  Boost_NO_SYSTEM_PATHS  - Set to ON to disable searching in locations not
                           specified by these hint variables. Default is OFF.
  Boost_ADDITIONAL_VERSIONS
                         - List of Boost versions not known to this module
                           (Boost install locations may contain the version)
```

and saves search results persistently in CMake cache entries:

```
  Boost_INCLUDE_DIR         - Directory containing Boost headers
  Boost_LIBRARY_DIR         - Directory containing Boost libraries
  Boost_<C>_LIBRARY_DEBUG   - Component <C> library debug variant
  Boost_<C>_LIBRARY_RELEASE - Component <C> library release variant
```

Users may set these hints or results as cache entries. Projects should not read these entries directly but instead use the above result variables. Note that some hint names start in upper-case "BOOST". One may specify these as environment variables if they are not specified as CMake variables or cache entries.

This module first searches for the Boost header files using the above hint variables (excluding BOOST_LIBRARYDIR) and saves the result in Boost_INCLUDE_DIR. Then it searches for requested component libraries using the above hints (excluding BOOST_INCLUDEDIR and Boost_ADDITIONAL_VERSIONS), "lib" directories near Boost_INCLUDE_DIR, and the library name configuration settings below. It saves the library directory in Boost_LIBRARY_DIR and individual library locations in Boost_<C>_LIBRARY_DEBUG and Boost_<C>_LIBRARY_RELEASE. When one changes settings used by previous searches in the same build tree (excluding environment variables) this module discards previous search results affected by the changes and searches again.

Boost libraries come in many variants encoded in their file name. Users or projects may tell this module which variant to find by setting variables:

```
  Boost_USE_MULTITHREADED  - Set to OFF to use the non-multithreaded
                             libraries ('mt' tag).  Default is ON.
  Boost_USE_STATIC_LIBS    - Set to ON to force the use of the static
                             libraries.  Default is OFF.
  Boost_USE_STATIC_RUNTIME - Set to ON or OFF to specify whether to use
                             libraries linked statically to the C++ runtime
                             ('s' tag).  Default is platform dependent.
  Boost_USE_DEBUG_PYTHON   - Set to ON to use libraries compiled with a
                             debug Python build ('y' tag). Default is OFF.
  Boost_USE_STLPORT        - Set to ON to use libraries compiled with
                             STLPort ('p' tag).  Default is OFF.
  Boost_USE_STLPORT_DEPRECATED_NATIVE_IOSTREAMS
                           - Set to ON to use libraries compiled with
                             STLPort deprecated "native iostreams"
                             ('n' tag).  Default is OFF.
  Boost_COMPILER           - Set to the compiler-specific library suffix
                             (e.g. "-gcc43").  Default is auto-computed
                             for the C++ compiler in use.
  Boost_THREADAPI          - Suffix for "thread" component library name,
                             such as "pthread" or "win32".  Names with
                             and without this suffix will both be tried.
```

Other variables one may set to control this module are:

```
  Boost_DEBUG              - Set to ON to enable debug output from FindBoost.
                             Please enable this before filing any bug report.
  Boost_DETAILED_FAILURE_MSG
                           - Set to ON to add detailed information to the
                             failure message even when the REQUIRED option
                             is not given to the find_package call.
  Boost_REALPATH           - Set to ON to resolve symlinks for discovered
                             libraries to assist with packaging.  For example,
                             the "system" component library may be resolved to
                             "/usr/lib/libboost_system.so.1.42.0" instead of
                             "/usr/lib/libboost_system.so".  This does not
                             affect linking and should not be enabled unless
                             the user needs this information.
```

On Visual Studio and Borland compilers Boost headers request automatic linking to corresponding libraries. This requires matching libraries to be linked explicitly or available in the link library search path. In this case setting Boost_USE_STATIC_LIBS to OFF may not achieve dynamic linking. Boost automatic linking typically requests static libraries with a few exceptions (such as Boost.Python). Use

```
  add_definitions(${Boost_LIB_DIAGNOSTIC_DEFINITIONS})
```

to ask Boost to report information about automatic linking requests.

Example to find Boost headers only:

```
find_package(Boost 1.36.0)
if(Boost_FOUND)
  include_directories(${Boost_INCLUDE_DIRS})
  add_executable(foo foo.cc)
endif()
```

Example to find Boost headers and some libraries:

```
set(Boost_USE_STATIC_LIBS        ON)
set(Boost_USE_MULTITHREADED      ON)
set(Boost_USE_STATIC_RUNTIME    OFF)
find_package(Boost 1.36.0 COMPONENTS date_time filesystem system ...)
if(Boost_FOUND)
  include_directories(${Boost_INCLUDE_DIRS})
  add_executable(foo foo.cc)
  target_link_libraries(foo ${Boost_LIBRARIES})
endif()
```

Boost CMake ----------

If Boost was built using the boost-cmake project it provides a package configuration file for use with find_package's Config mode. This module looks for the package configuration file called BoostConfig.cmake or boost-config.cmake and stores the result in cache entry "Boost_DIR". If found, the package configuration file is loaded and this module returns with no further action. See documentation of the Boost CMake package configuration for details on what it provides.

Set Boost_NO_BOOST_CMAKE to ON to disable the search for boost-cmake.

- **`FindBullet`**: Try to find the Bullet physics engine

```
This module defines the following variables


BULLET_FOUND - Was bullet found
BULLET_INCLUDE_DIRS - the Bullet include directories
BULLET_LIBRARIES - Link to this, by default it includes
                   all bullet components (Dynamics,
                   Collision, LinearMath, & SoftBody)


This module accepts the following variables


BULLET_ROOT - Can be set to bullet install path or Windows build path
```

- **`FindCABLE`**: Find CABLE

  This module finds if CABLE is installed and determines where the include files and libraries are. This code sets the following variables:

```
CABLE             the path to the cable executable
CABLE_TCL_LIBRARY the path to the Tcl wrapper library
CABLE_INCLUDE_DIR the path to the include directory
```

  To build Tcl wrappers, you should add shared library and link it to ${CABLE_TCL_LIBRARY}. You should also add ${CABLE_INCLUDE_DIR} as an include directory.

- **`FindCUDA`**: Tools for building CUDA C files: libraries and build dependencies.

  This script locates the NVIDIA CUDA C tools. It should work on linux, windows, and mac and should be reasonably up to date with CUDA C releases.

  This script makes use of the standard find_package arguments of <VERSION>, REQUIRED and QUIET. CUDA_FOUND will report if an acceptable version of CUDA was found.

  The script will prompt the user to specify CUDA_TOOLKIT_ROOT_DIR if the prefix cannot be determined by the location of nvcc in the system path and REQUIRED is specified to find_package(). To use a different installed version of the toolkit set the environment variable CUDA_BIN_PATH before running cmake (e.g. CUDA_BIN_PATH=/usr/local/cuda1.0 instead of the default /usr/local/cuda) or set CUDA_TOOLKIT_ROOT_DIR after configuring. If you change the value of CUDA_TOOLKIT_ROOT_DIR, various components that depend on the path will be relocated.

It might be necessary to set CUDA_TOOLKIT_ROOT_DIR manually on certain platforms, or to use a cuda runtime not installed in the default location. In newer versions of the toolkit the cuda library is included with the graphics driver- be sure that the driver version matches what is needed by the cuda runtime version.

The following variables affect the behavior of the macros in the script (in alphebetical order). Note that any of these flags can be changed multiple times in the same directory before calling CUDA_ADD_EXECUTABLE, CUDA_ADD_LIBRARY, CUDA_COMPILE, CUDA_COMPILE_PTX or CUDA_WRAP_SRCS.

```
CUDA_64_BIT_DEVICE_CODE (Default matches host bit size)
-- Set to ON to compile for 64 bit device code, OFF for 32 bit device code.
   Note that making this different from the host code when generating object
   or C files from CUDA code just won't work, because size_t gets defined by
   nvcc in the generated source.  If you compile to PTX and then load the
   file yourself, you can mix bit sizes between device and host.


CUDA_ATTACH_VS_BUILD_RULE_TO_CUDA_FILE (Default ON)
-- Set to ON if you want the custom build rule to be attached to the source
   file in Visual Studio.  Turn OFF if you add the same cuda file to multiple
   targets.


   This allows the user to build the target from the CUDA file; however, bad
   things can happen if the CUDA source file is added to multiple targets.
   When performing parallel builds it is possible for the custom build
   command to be run more than once and in parallel causing cryptic build
   errors.  VS runs the rules for every source file in the target, and a
   source can have only one rule no matter how many projects it is added to.
   When the rule is run from multiple targets race conditions can occur on
   the generated file.  Eventually everything will get built, but if the user
   is unaware of this behavior, there may be confusion.  It would be nice if
   this script could detect the reuse of source files across multiple targets
   and turn the option off for the user, but no good solution could be found.


CUDA_BUILD_CUBIN (Default OFF)
-- Set to ON to enable and extra compilation pass with the -cubin option in
   Device mode. The output is parsed and register, shared memory usage is
   printed during build.


CUDA_BUILD_EMULATION (Default OFF for device mode)
-- Set to ON for Emulation mode. -D_DEVICEEMU is defined for CUDA C files
   when CUDA_BUILD_EMULATION is TRUE.


CUDA_GENERATED_OUTPUT_DIR (Default CMAKE_CURRENT_BINARY_DIR)
-- Set to the path you wish to have the generated files placed.  If it is
   blank output files will be placed in CMAKE_CURRENT_BINARY_DIR.
   Intermediate files will always be placed in
   CMAKE_CURRENT_BINARY_DIR/CMakeFiles.


CUDA_HOST_COMPILATION_CPP (Default ON)
-- Set to OFF for C compilation of host code.


CUDA_HOST_COMPILER (Default CMAKE_C_COMPILER, $(VCInstallDir)/bin for VS)
-- Set the host compiler to be used by nvcc.  Ignored if -ccbin or
   --compiler-bindir is already present in the CUDA_NVCC_FLAGS or
   CUDA_NVCC_FLAGS_<CONFIG> variables.  For Visual Studio targets
   $(VCInstallDir)/bin is a special value that expands out to the path when
   the command is run from withing VS.


CUDA_NVCC_FLAGS
CUDA_NVCC_FLAGS_<CONFIG>
-- Additional NVCC command line arguments.  NOTE: multiple arguments must be
   semi-colon delimited (e.g. --compiler-options;-Wall)
```

```
CUDA_PROPAGATE_HOST_FLAGS (Default ON)
-- Set to ON to propagate CMAKE_{C,CXX}_FLAGS and their configuration
   dependent counterparts (e.g. CMAKE_C_FLAGS_DEBUG) automatically to the
   host compiler through nvcc's -Xcompiler flag.  This helps make the
   generated host code match the rest of the system better.  Sometimes
   certain flags give nvcc problems, and this will help you turn the flag
   propagation off.  This does not affect the flags supplied directly to nvcc
   via CUDA_NVCC_FLAGS or through the OPTION flags specified through
   CUDA_ADD_LIBRARY, CUDA_ADD_EXECUTABLE, or CUDA_WRAP_SRCS.  Flags used for
   shared library compilation are not affected by this flag.


CUDA_SEPARABLE_COMPILATION (Default OFF)
-- If set this will enable separable compilation for all CUDA runtime object
   files.  If used outside of CUDA_ADD_EXECUTABLE and CUDA_ADD_LIBRARY
   (e.g. calling CUDA_WRAP_SRCS directly),
   CUDA_COMPUTE_SEPARABLE_COMPILATION_OBJECT_FILE_NAME and
   CUDA_LINK_SEPARABLE_COMPILATION_OBJECTS should be called.


CUDA_VERBOSE_BUILD (Default OFF)
-- Set to ON to see all the commands used when building the CUDA file.  When
   using a Makefile generator the value defaults to VERBOSE (run make
   VERBOSE=1 to see output), although setting CUDA_VERBOSE_BUILD to ON will
   always print the output.
```

The script creates the following macros (in alphebetical order):

```
CUDA_ADD_CUFFT_TO_TARGET( cuda_target )
-- Adds the cufft library to the target (can be any target).  Handles whether
   you are in emulation mode or not.


CUDA_ADD_CUBLAS_TO_TARGET( cuda_target )
-- Adds the cublas library to the target (can be any target).  Handles
   whether you are in emulation mode or not.


CUDA_ADD_EXECUTABLE( cuda_target file0 file1 ...
                     [WIN32] [MACOSX_BUNDLE] [EXCLUDE_FROM_ALL] [OPTIONS ...] )
-- Creates an executable "cuda_target" which is made up of the files
   specified.  All of the non CUDA C files are compiled using the standard
   build rules specified by CMAKE and the cuda files are compiled to object
   files using nvcc and the host compiler.  In addition CUDA_INCLUDE_DIRS is
   added automatically to include_directories().  Some standard CMake target
   calls can be used on the target after calling this macro
   (e.g. set_target_properties and target_link_libraries), but setting
   properties that adjust compilation flags will not affect code compiled by
   nvcc.  Such flags should be modified before calling CUDA_ADD_EXECUTABLE,
   CUDA_ADD_LIBRARY or CUDA_WRAP_SRCS.


CUDA_ADD_LIBRARY( cuda_target file0 file1 ...
                  [STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL] [OPTIONS ...] )
-- Same as CUDA_ADD_EXECUTABLE except that a library is created.


CUDA_BUILD_CLEAN_TARGET()
-- Creates a convience target that deletes all the dependency files
   generated.  You should make clean after running this target to ensure the
   dependency files get regenerated.


CUDA_COMPILE( generated_files file0 file1 ... [STATIC | SHARED | MODULE]
              [OPTIONS ...] )
-- Returns a list of generated files from the input source files to be used
   with ADD_LIBRARY or ADD_EXECUTABLE.
```

```
CUDA_COMPILE_PTX( generated_files file0 file1 ... [OPTIONS ...] )
-- Returns a list of PTX files generated from the input source files.


CUDA_COMPUTE_SEPARABLE_COMPILATION_OBJECT_FILE_NAME( output_file_var
                                                     cuda_target
                                                     object_files )
-- Compute the name of the intermediate link file used for separable
   compilation.  This file name is typically passed into
   CUDA_LINK_SEPARABLE_COMPILATION_OBJECTS.  output_file_var is produced
   based on cuda_target the list of objects files that need separable
   compilation as specified by object_files.  If the object_files list is
   empty, then output_file_var will be empty.  This function is called
   automatically for CUDA_ADD_LIBRARY and CUDA_ADD_EXECUTABLE.  Note that
   this is a function and not a macro.


CUDA_INCLUDE_DIRECTORIES( path0 path1 ... )
-- Sets the directories that should be passed to nvcc
   (e.g. nvcc -Ipath0 -Ipath1 ... ). These paths usually contain other .cu
   files.



CUDA_LINK_SEPARABLE_COMPILATION_OBJECTS( output_file_var cuda_target
                                         nvcc_flags object_files)


-- Generates the link object required by separable compilation from the given
   object files.  This is called automatically for CUDA_ADD_EXECUTABLE and
   CUDA_ADD_LIBRARY, but can be called manually when using CUDA_WRAP_SRCS
   directly.  When called from CUDA_ADD_LIBRARY or CUDA_ADD_EXECUTABLE the
   nvcc_flags passed in are the same as the flags passed in via the OPTIONS
   argument.  The only nvcc flag added automatically is the bitness flag as
   specified by CUDA_64_BIT_DEVICE_CODE.  Note that this is a function
   instead of a macro.


CUDA_WRAP_SRCS ( cuda_target format generated_files file0 file1 ...
                 [STATIC | SHARED | MODULE] [OPTIONS ...] )
-- This is where all the magic happens.  CUDA_ADD_EXECUTABLE,
   CUDA_ADD_LIBRARY, CUDA_COMPILE, and CUDA_COMPILE_PTX all call this
   function under the hood.


   Given the list of files (file0 file1 ... fileN) this macro generates
   custom commands that generate either PTX or linkable objects (use "PTX" or
   "OBJ" for the format argument to switch).  Files that don't end with .cu
   or have the HEADER_FILE_ONLY property are ignored.


   The arguments passed in after OPTIONS are extra command line options to
   give to nvcc.  You can also specify per configuration options by
   specifying the name of the configuration followed by the options.  General
   options must preceed configuration specific options.  Not all
   configurations need to be specified, only the ones provided will be used.


      OPTIONS -DFLAG=2 "-DFLAG_OTHER=space in flag"
      DEBUG -g
      RELEASE --use_fast_math
      RELWITHDEBINFO --use_fast_math;-g
      MINSIZEREL --use_fast_math


For certain configurations (namely VS generating object files with
CUDA_ATTACH_VS_BUILD_RULE_TO_CUDA_FILE set to ON), no generated file will
be produced for the given cuda file.  This is because when you add the
cuda file to Visual Studio it knows that this file produces an object file
```

```
and will link in the resulting object file automatically.


This script will also generate a separate cmake script that is used at
build time to invoke nvcc.  This is for several reasons.


  1. nvcc can return negative numbers as return values which confuses
  Visual Studio into thinking that the command succeeded.  The script now
  checks the error codes and produces errors when there was a problem.


  2. nvcc has been known to not delete incomplete results when it
  encounters problems.  This confuses build systems into thinking the
  target was generated when in fact an unusable file exists.  The script
  now deletes the output files if there was an error.


    3. By putting all the options that affect the build into a file and then
    make the build rule dependent on the file, the output files will be
    regenerated when the options change.


  This script also looks at optional arguments STATIC, SHARED, or MODULE to
  determine when to target the object compilation for a shared library.
  BUILD_SHARED_LIBS is ignored in CUDA_WRAP_SRCS, but it is respected in
  CUDA_ADD_LIBRARY.  On some systems special flags are added for building
  objects intended for shared libraries.  A preprocessor macro,
  <target_name>_EXPORTS is defined when a shared library compilation is
  detected.


  Flags passed into add_definitions with -D or /D are passed along to nvcc.
```

The script defines the following variables:

```
  CUDA_VERSION_MAJOR    -- The major version of cuda as reported by nvcc.
  CUDA_VERSION_MINOR    -- The minor version.
  CUDA_VERSION
  CUDA_VERSION_STRING   -- CUDA_VERSION_MAJOR.CUDA_VERSION_MINOR


  CUDA_TOOLKIT_ROOT_DIR -- Path to the CUDA Toolkit (defined if not set).
  CUDA_SDK_ROOT_DIR     -- Path to the CUDA SDK.  Use this to find files in the
                           SDK.  This script will not directly support finding
                           specific libraries or headers, as that isn't
                           supported by NVIDIA.  If you want to change
                           libraries when the path changes see the
                           FindCUDA.cmake script for an example of how to clear
                           these variables.  There are also examples of how to
                           use the CUDA_SDK_ROOT_DIR to locate headers or
                           libraries, if you so choose (at your own risk).
  CUDA_INCLUDE_DIRS     -- Include directory for cuda headers.  Added automatically
                           for CUDA_ADD_EXECUTABLE and CUDA_ADD_LIBRARY.
  CUDA_LIBRARIES        -- Cuda RT library.
  CUDA_CUFFT_LIBRARIES  -- Device or emulation library for the Cuda FFT
                           implementation (alternative to:
                           CUDA_ADD_CUFFT_TO_TARGET macro)
  CUDA_CUBLAS_LIBRARIES -- Device or emulation library for the Cuda BLAS
                           implementation (alterative to:
                           CUDA_ADD_CUBLAS_TO_TARGET macro).
  CUDA_cupti_LIBRARY    -- CUDA Profiling Tools Interface library.
                           Only available for CUDA version 4.0+.
  CUDA_curand_LIBRARY   -- CUDA Random Number Generation library.
                           Only available for CUDA version 3.2+.
  CUDA_cusparse_LIBRARY -- CUDA Sparse Matrix library.
                           Only available for CUDA version 3.2+.
  CUDA_npp_LIBRARY      -- NVIDIA Performance Primitives library.
                           Only available for CUDA version 4.0+.
```

```
CUDA_nvcuvenc_LIBRARY -- CUDA Video Encoder library.
                         Only available for CUDA version 3.2+.
                         Windows only.
CUDA_nvcuvid_LIBRARY  -- CUDA Video Decoder library.
                         Only available for CUDA version 3.2+.
                         Windows only.



James Bigler, NVIDIA Corp (nvidia.com - jbigler)
Abe Stephens, SCI Institute -- http://www.sci.utah.edu/~abe/FindCuda.html


Copyright (c) 2008 - 2009 NVIDIA Corporation.  All rights reserved.


Copyright (c) 2007-2009
Scientific Computing and Imaging Institute, University of Utah


This code is licensed under the MIT License.  See the FindCUDA.cmake script
for the text of the license.
```

- **`FindCURL`**: Find curl

  Find the native CURL headers and libraries.

  ```
  CURL_INCLUDE_DIRS   - where to find curl/curl.h, etc.
  CURL_LIBRARIES      - List of libraries when using curl.
  CURL_FOUND          - True if curl found.
  CURL_VERSION_STRING - the version of curl found (since CMake 2.8.8)
  ```

- **`FindCVS`**:

  The module defines the following variables:

  ```
  CVS_EXECUTABLE - path to cvs command line client
  CVS_FOUND - true if the command line client was found
  ```

  Example usage:

  ```
  find_package(CVS)
  if(CVS_FOUND)
    message("CVS found: ${CVS_EXECUTABLE}")
  endif()
  ```

- **`FindCoin3D`**: Find Coin3D (Open Inventor)

  Coin3D is an implementation of the Open Inventor API. It provides data structures and algorithms for 3D visualization
  http://www.coin3d.org/

  This module defines the following variables

  ```
  COIN3D_FOUND         - system has Coin3D - Open Inventor
  COIN3D_INCLUDE_DIRS  - where the Inventor include directory can be found
  COIN3D_LIBRARIES     - Link to this to use Coin3D
  ```

- **`FindCups`**: Try to find the Cups printing system

  Once done this will define

  ```
  CUPS_FOUND - system has Cups
  CUPS_INCLUDE_DIR - the Cups include directory
  CUPS_LIBRARIES - Libraries needed to use Cups
  CUPS_VERSION_STRING - version of Cups found (since CMake 2.8.8)
  Set CUPS_REQUIRE_IPP_DELETE_ATTRIBUTE to TRUE if you need a version which
  features this function (i.e. at least 1.1.19)
  ```

- **`FindCurses`**: Find the curses include file and library

  ```
  CURSES_FOUND - system has Curses
  CURSES_INCLUDE_DIR - the Curses include directory
  CURSES_LIBRARIES - The libraries needed to use Curses
  CURSES_HAVE_CURSES_H - true if curses.h is available
  ```

```
  CURSES_HAVE_NCURSES_H - true if ncurses.h is available
  CURSES_HAVE_NCURSES_NCURSES_H - true if ncurses/ncurses.h is available
  CURSES_HAVE_NCURSES_CURSES_H - true if ncurses/curses.h is available
  CURSES_LIBRARY - set for backwards compatibility with 2.4 CMake
```

Set CURSES_NEED_NCURSES to TRUE before the find_package() command if NCurses functionality is required.

- **`FindCxxTest`**: Find CxxTest

  Find the CxxTest suite and declare a helper macro for creating unit tests and integrating them with CTest. For more details on CxxTest see http://cxxtest.tigris.org

  INPUT Variables

```
  CXXTEST_USE_PYTHON [deprecated since 1.3]
      Only used in the case both Python & Perl
      are detected on the system to control
      which CxxTest code generator is used.
      Valid only for CxxTest version 3.


      NOTE: In older versions of this Find Module,
      this variable controlled if the Python test
      generator was used instead of the Perl one,
      regardless of which scripting language the
      user had installed.


  CXXTEST_TESTGEN_ARGS (since CMake 2.8.3)
      Specify a list of options to pass to the CxxTest code
      generator.  If not defined, --error-printer is
      passed.
```

OUTPUT Variables

```
  CXXTEST_FOUND
      True if the CxxTest framework was found
  CXXTEST_INCLUDE_DIRS
      Where to find the CxxTest include directory
  CXXTEST_PERL_TESTGEN_EXECUTABLE
      The perl-based test generator
  CXXTEST_PYTHON_TESTGEN_EXECUTABLE
      The python-based test generator
  CXXTEST_TESTGEN_EXECUTABLE (since CMake 2.8.3)
      The test generator that is actually used (chosen using user preferences
      and interpreters found in the system)
  CXXTEST_TESTGEN_INTERPRETER (since CMake 2.8.3)
      The full path to the Perl or Python executable on the system
```

MACROS for optional use by CMake users:

```
   CXXTEST_ADD_TEST(<test_name> <gen_source_file> <input_files_to_testgen...>)
      Creates a CxxTest runner and adds it to the CTest testing suite
      Parameters:
          test_name               The name of the test
          gen_source_file         The generated source filename to be
                                  generated by CxxTest
          input_files_to_testgen  The list of header files containing the
                                  CxxTest::TestSuite's to be included in
                                  this runner


      #==============
      Example Usage:


          find_package(CxxTest)
          if(CXXTEST_FOUND)
              include_directories(${CXXTEST_INCLUDE_DIR})
              enable_testing()
```

```
        CXXTEST_ADD_TEST(unittest_foo foo_test.cc
                         ${CMAKE_CURRENT_SOURCE_DIR}/foo_test.h)
        target_link_libraries(unittest_foo foo) # as needed
    endif()


       This will (if CxxTest is found):
       1. Invoke the testgen executable to autogenerate foo_test.cc in the
          binary tree from "foo_test.h" in the current source directory.
       2. Create an executable and test called unittest_foo.


#=============
Example foo_test.h:


    #include <cxxtest/TestSuite.h>


    class MyTestSuite : public CxxTest::TestSuite
    {
    public:
       void testAddition( void )
       {
          TS_ASSERT( 1 + 1 > 1 );
          TS_ASSERT_EQUALS( 1 + 1, 2 );
       }
    };
```

- **`FindCygwin`**: this module looks for Cygwin

- **`FindDCMTK`**: find DCMTK libraries and applications

- **`FindDart`**: Find DART

  This module looks for the dart testing software and sets DART_ROOT to point to where it found it.

- **`FindDevIL`**:

  This module locates the developer's image library. http://openil.sourceforge.net/

  This module sets:

```
   IL_LIBRARIES -    the name of the IL library. These include the full path to
                     the core DevIL library. This one has to be linked into the
                     application.
   ILU_LIBRARIES -   the name of the ILU library. Again, the full path. This
                     library is for filters and effects, not actual loading. It
                     doesn't have to be linked if the functionality it provides
                     is not used.
   ILUT_LIBRARIES -  the name of the ILUT library. Full path. This part of the
                     library interfaces with OpenGL. It is not strictly needed
                     in applications.
   IL_INCLUDE_DIR -  where to find the il.h, ilu.h and ilut.h files.
   IL_FOUND -        this is set to TRUE if all the above variables were set.
                     This will be set to false if ILU or ILUT are not found,
                     even if they are not needed. In most systems, if one
                     library is found all the others are as well. That's the
                     way the DevIL developers release it.
```

- **`FindDoxygen`**: This module looks for Doxygen and the path to Graphviz's dot

  Doxygen is a documentation generation tool. Please see http://www.doxygen.org

  This module accepts the following optional variables:

```
   DOXYGEN_SKIP_DOT       = If true this module will skip trying to find Dot
                            (an optional component often used by Doxygen)
```

This modules defines the following variables:

```
DOXYGEN_EXECUTABLE     = The path to the doxygen command.
DOXYGEN_FOUND          = Was Doxygen found or not?
DOXYGEN_VERSION        = The version reported by doxygen --version


DOXYGEN_DOT_EXECUTABLE = The path to the dot program used by doxygen.
DOXYGEN_DOT_FOUND      = Was Dot found or not?
DOXYGEN_DOT_PATH       = The path to dot not including the executable
```

- **`FindEXPAT`**: Find expat

  Find the native EXPAT headers and libraries.

  ```
  EXPAT_INCLUDE_DIRS - where to find expat.h, etc.
  EXPAT_LIBRARIES    - List of libraries when using expat.
  EXPAT_FOUND        - True if expat found.
  ```

- **`FindFLEX`**: Find flex executable and provides a macro to generate custom build rules

  The module defines the following variables:

  ```
  FLEX_FOUND - true is flex executable is found
  FLEX_EXECUTABLE - the path to the flex executable
  FLEX_VERSION - the version of flex
  FLEX_LIBRARIES - The flex libraries
  FLEX_INCLUDE_DIRS - The path to the flex headers
  ```

  The minimum required version of flex can be specified using the standard syntax, e.g. find_package(FLEX 2.5.13)

  If flex is found on the system, the module provides the macro:

  ```
  FLEX_TARGET(Name FlexInput FlexOutput [COMPILE_FLAGS <string>])
  ```

  which creates a custom command to generate the <FlexOutput> file from the <FlexInput> file. If COMPILE_FLAGS option is specified, the next parameter is added to the flex command line. Name is an alias used to get details of this custom command. Indeed the macro defines the following variables:

  ```
  FLEX_${Name}_DEFINED - true is the macro ran successfully
  FLEX_${Name}_OUTPUTS - the source file generated by the custom rule, an
  alias for FlexOutput
  FLEX_${Name}_INPUT - the flex source file, an alias for ${FlexInput}
  ```

  Flex scanners oftenly use tokens defined by Bison: the code generated by Flex depends of the header generated by Bison. This module also defines a macro:

  ```
  ADD_FLEX_BISON_DEPENDENCY(FlexTarget BisonTarget)
  ```

  which adds the required dependency between a scanner and a parser where <FlexTarget> and <BisonTarget> are the first parameters of respectively FLEX_TARGET and BISON_TARGET macros.

  ```
  ============================================================================
  Example:


   find_package(BISON)
   find_package(FLEX)


   BISON_TARGET(MyParser parser.y ${CMAKE_CURRENT_BINARY_DIR}/parser.cpp)
   FLEX_TARGET(MyScanner lexer.l  ${CMAKE_CURRENT_BINARY_DIR}/lexer.cpp)
   ADD_FLEX_BISON_DEPENDENCY(MyScanner MyParser)


   include_directories(${CMAKE_CURRENT_BINARY_DIR})
   add_executable(Foo
      Foo.cc
  ```

```
    ${BISON_MyParser_OUTPUTS}
    ${FLEX_MyScanner_OUTPUTS}
  )
 ============================================================================
```

- **`FindFLTK`**: Find the native FLTK includes and library

  By default FindFLTK.cmake will search for all of the FLTK components and add them to the FLTK_LIBRARIES variable.

  ```
  You can limit the components which get placed in FLTK_LIBRARIES by
  defining one or more of the following three options:


    FLTK_SKIP_OPENGL, set to true to disable searching for opengl and
                      the FLTK GL library
    FLTK_SKIP_FORMS, set to true to disable searching for fltk_forms
    FLTK_SKIP_IMAGES, set to true to disable searching for fltk_images


    FLTK_SKIP_FLUID, set to true if the fluid binary need not be present
                     at build time
  ```

  The following variables will be defined:

  ```
    FLTK_FOUND, True if all components not skipped were found
    FLTK_INCLUDE_DIR, where to find include files
    FLTK_LIBRARIES, list of fltk libraries you should link against
    FLTK_FLUID_EXECUTABLE, where to find the Fluid tool
    FLTK_WRAP_UI, This enables the FLTK_WRAP_UI command
  ```

  The following cache variables are assigned but should not be used. See the FLTK_LIBRARIES variable instead.

  ```
    FLTK_BASE_LIBRARY   = the full path to fltk.lib
    FLTK_GL_LIBRARY     = the full path to fltk_gl.lib
    FLTK_FORMS_LIBRARY  = the full path to fltk_forms.lib
    FLTK_IMAGES_LIBRARY = the full path to fltk_images.lib
  ```

- **`FindFLTK2`**: Find the native FLTK2 includes and library

  The following settings are defined

  ```
    FLTK2_FLUID_EXECUTABLE, where to find the Fluid tool
    FLTK2_WRAP_UI, This enables the FLTK2_WRAP_UI command
    FLTK2_INCLUDE_DIR, where to find include files
    FLTK2_LIBRARIES, list of fltk2 libraries
    FLTK2_FOUND, Don't use FLTK2 if false.
  ```

  The following settings should not be used in general.

  ```
    FLTK2_BASE_LIBRARY   = the full path to fltk2.lib
    FLTK2_GL_LIBRARY     = the full path to fltk2_gl.lib
    FLTK2_IMAGES_LIBRARY = the full path to fltk2_images.lib
  ```

- **`FindFreetype`**: Locate FreeType library

  This module defines

  ```
    FREETYPE_LIBRARIES, the library to link against
    FREETYPE_FOUND, if false, do not try to link to FREETYPE
    FREETYPE_INCLUDE_DIRS, where to find headers.
    FREETYPE_VERSION_STRING, the version of freetype found (since CMake 2.8.8)
    This is the concatenation of the paths:
    FREETYPE_INCLUDE_DIR_ft2build
    FREETYPE_INCLUDE_DIR_freetype2
  ```

  $FREETYPE_DIR is an environment variable that would correspond to the ./configure --prefix=$FREETYPE_DIR used in building FREETYPE.

- **`FindGCCXML`**: Find the GCC-XML front-end executable.

  This module will define the following variables:

```
GCCXML - the GCC-XML front-end executable.
```

- **FindGDAL**:

  Locate gdal

  This module accepts the following environment variables:

  ```
  GDAL_DIR or GDAL_ROOT - Specify the location of GDAL
  ```

  This module defines the following CMake variables:

  ```
  GDAL_FOUND - True if libgdal is found
  GDAL_LIBRARY - A variable pointing to the GDAL library
  GDAL_INCLUDE_DIR - Where to find the headers
  ```

- **FindGIF**:

  This module searches giflib and defines GIF_LIBRARIES - libraries to link to in order to use GIF GIF_FOUND, if false, do not try to link GIF_INCLUDE_DIR, where to find the headers GIF_VERSION, reports either version 4 or 3 (for everything before version 4)

  The minimum required version of giflib can be specified using the standard syntax, e.g. find_package(GIF 4)

  $GIF_DIR is an environment variable that would correspond to the ./configure --prefix=$GIF_DIR

- **FindGLEW**: Find the OpenGL Extension Wrangler Library (GLEW)

  This module defines the following variables:

  ```
  GLEW_INCLUDE_DIRS - include directories for GLEW
  GLEW_LIBRARIES - libraries to link against GLEW
  GLEW_FOUND - true if GLEW has been found and can be used
  ```

- **FindGLUT**: try to find glut library and include files

  ```
  GLUT_INCLUDE_DIR, where to find GL/glut.h, etc.
  GLUT_LIBRARIES, the libraries to link against
  GLUT_FOUND, If false, do not try to use GLUT.
  ```

  Also defined, but not for general use are:

  ```
  GLUT_glut_LIBRARY = the full path to the glut library.
  GLUT_Xmu_LIBRARY  = the full path to the Xmu library.
  GLUT_Xi_LIBRARY   = the full path to the Xi Library.
  ```

- **FindGTK**: try to find GTK (and glib) and GTKGLArea

  ```
  GTK_INCLUDE_DIR   - Directories to include to use GTK
  GTK_LIBRARIES     - Files to link against to use GTK
  GTK_FOUND         - GTK was found
  GTK_GL_FOUND      - GTK's GL features were found
  ```

- **FindGTK2**: FindGTK2.cmake

  This module can find the GTK2 widget libraries and several of its other optional components like gtkmm, glade, and glademm.

  NOTE: If you intend to use version checking, CMake 2.6.2 or later is

  ```
  required.
  ```

  Specify one or more of the following components as you call this find module. See example below.

  ```
  gtk
  gtkmm
  glade
  glademm
  ```

  The following variables will be defined for your use

  ```
  GTK2_FOUND - Were all of your specified components found?
  GTK2_INCLUDE_DIRS - All include directories
  GTK2_LIBRARIES - All libraries
  GTK2_DEFINITIONS - Additional compiler flags
  ```

  ```
  GTK2_VERSION - The version of GTK2 found (x.y.z)
  GTK2_MAJOR_VERSION - The major version of GTK2
  ```

```
   GTK2_MINOR_VERSION - The minor version of GTK2
   GTK2_PATCH_VERSION - The patch version of GTK2
```

Optional variables you can define prior to calling this module:

```
   GTK2_DEBUG - Enables verbose debugging of the module
   GTK2_ADDITIONAL_SUFFIXES - Allows defining additional directories to
                              search for include files
```

================== Example Usage:

```
   Call find_package() once, here are some examples to pick from:


   Require GTK 2.6 or later
       find_package(GTK2 2.6 REQUIRED gtk)


      Require GTK 2.10 or later and Glade
          find_package(GTK2 2.10 REQUIRED gtk glade)


      Search for GTK/GTKMM 2.8 or later
          find_package(GTK2 2.8 COMPONENTS gtk gtkmm)


      if(GTK2_FOUND)
         include_directories(${GTK2_INCLUDE_DIRS})
         add_executable(mygui mygui.cc)
         target_link_libraries(mygui ${GTK2_LIBRARIES})
      endif()
```

- **FindGTest**: --------------------

  Locate the Google C++ Testing Framework.

  Defines the following variables:

  ```
     GTEST_FOUND - Found the Google Testing framework
     GTEST_INCLUDE_DIRS - Include directories
  ```

  Also defines the library variables below as normal variables. These contain debug/optimized keywords when a debugging library is found.

  ```
     GTEST_BOTH_LIBRARIES - Both libgtest & libgtest-main
     GTEST_LIBRARIES - libgtest
     GTEST_MAIN_LIBRARIES - libgtest-main
  ```

  Accepts the following variables as input:

  ```
     GTEST_ROOT - (as a CMake or environment variable)
                  The root directory of the gtest install prefix


     GTEST_MSVC_SEARCH - If compiling with MSVC, this variable can be set to
                         "MD" or "MT" to enable searching a GTest build tree
                         (defaults: "MD")
  ```

  Example Usage:

  ```
      enable_testing()
      find_package(GTest REQUIRED)
      include_directories(${GTEST_INCLUDE_DIRS})


      add_executable(foo foo.cc)
      target_link_libraries(foo ${GTEST_BOTH_LIBRARIES})
  ```

```
add_test(AllTestsInFoo foo)
```

If you would like each Google test to show up in CTest as a test you may use the following macro. NOTE: It will slow down your tests by running an executable for each test and test fixture. You will also have to rerun CMake after adding or removing tests or test fixtures.

GTEST_ADD_TESTS(executable extra_args ARGN)

```
executable = The path to the test executable
extra_args = Pass a list of extra arguments to be passed to
             executable enclosed in quotes (or "" for none)
ARGN =       A list of source files to search for tests & test
             fixtures.
```

```
Example:
   set(FooTestArgs --foo 1 --bar 2)
   add_executable(FooTest FooUnitTest.cc)
   GTEST_ADD_TESTS(FooTest "${FooTestArgs}" FooUnitTest.cc)
```

- **FindGettext**: Find GNU gettext tools

  This module looks for the GNU gettext tools. This module defines the following values:

```
GETTEXT_MSGMERGE_EXECUTABLE: the full path to the msgmerge tool.
GETTEXT_MSGFMT_EXECUTABLE: the full path to the msgfmt tool.
GETTEXT_FOUND: True if gettext has been found.
GETTEXT_VERSION_STRING: the version of gettext found (since CMake 2.8.8)
```

  Additionally it provides the following macros: GETTEXT_CREATE_TRANSLATIONS (outputFile [ALL] file1 ... fileN)

```
This will create a target "translations" which will convert the
given input po files into the binary output mo file. If the
ALL option is used, the translations will also be created when
building the default target.
```

  GETTEXT_PROCESS_POT(<potfile> [ALL] [INSTALL_DESTINATION <destdir>] LANGUAGES <lang1> <lang2> ...)

```
Process the given pot file to mo files.
If INSTALL_DESTINATION is given then automatically install rules will be created,
the language subdirectory will be taken into account (by default use share/locale/).
If ALL is specified, the pot file is processed when building the all traget.
It creates a custom target "potfile".
```

  GETTEXT_PROCESS_PO_FILES(<lang> [ALL] [INSTALL_DESTINATION <dir>] PO_FILES <po1> <po2> ...)

```
Process the given po files to mo files for the given language.
If INSTALL_DESTINATION is given then automatically install rules will be created,
the language subdirectory will be taken into account (by default use share/locale/).
If ALL is specified, the po files are processed when building the all traget.
It creates a custom target "pofiles".
```

- **FindGit**:

  The module defines the following variables:

```
GIT_EXECUTABLE - path to git command line client
GIT_FOUND - true if the command line client was found
GIT_VERSION_STRING - the version of git found (since CMake 2.8.8)
```

  Example usage:

```
find_package(Git)
if(GIT_FOUND)
  message("git found: ${GIT_EXECUTABLE}")
endif()
```

- **FindGnuTLS**: Try to find the GNU Transport Layer Security library (gnutls)

  Once done this will define

```
GNUTLS_FOUND - System has gnutls
```

```
GNUTLS_INCLUDE_DIR - The gnutls include directory
GNUTLS_LIBRARIES - The libraries needed to use gnutls
GNUTLS_DEFINITIONS - Compiler switches required for using gnutls
```

- **`FindGnuplot`**: this module looks for gnuplot

  Once done this will define

  ```
  GNUPLOT_FOUND - system has Gnuplot
  GNUPLOT_EXECUTABLE - the Gnuplot executable
  GNUPLOT_VERSION_STRING - the version of Gnuplot found (since CMake 2.8.8)
  ```

  GNUPLOT_VERSION_STRING will not work for old versions like 3.7.1.

- **`FindHDF5`**: Find HDF5, a library for reading and writing self describing array data.

  This module invokes the HDF5 wrapper compiler that should be installed alongside HDF5. Depending upon the HDF5 Configuration, the wrapper compiler is called either h5cc or h5pcc. If this succeeds, the module will then call the compiler with the -show argument to see what flags are used when compiling an HDF5 client application.

  The module will optionally accept the COMPONENTS argument. If no COMPONENTS are specified, then the find module will default to finding only the HDF5 C library. If one or more COMPONENTS are specified, the module will attempt to find the language bindings for the specified components. The only valid components are C, CXX, Fortran, HL, and Fortran_HL. If the COMPONENTS argument is not given, the module will attempt to find only the C bindings.

  On UNIX systems, this module will read the variable HDF5_USE_STATIC_LIBRARIES to determine whether or not to prefer a static link to a dynamic link for HDF5 and all of it's dependencies. To use this feature, make sure that the HDF5_USE_STATIC_LIBRARIES variable is set before the call to find_package.

  To provide the module with a hint about where to find your HDF5 installation, you can set the environment variable HDF5_ROOT. The Find module will then look in this path when searching for HDF5 executables, paths, and libraries.

  In addition to finding the includes and libraries required to compile an HDF5 client application, this module also makes an effort to find tools that come with the HDF5 distribution that may be useful for regression testing.

  This module will define the following variables:

  ```
  HDF5_INCLUDE_DIRS - Location of the hdf5 includes
  HDF5_INCLUDE_DIR - Location of the hdf5 includes (deprecated)
  HDF5_DEFINITIONS - Required compiler definitions for HDF5
  HDF5_C_LIBRARIES - Required libraries for the HDF5 C bindings.
  HDF5_CXX_LIBRARIES - Required libraries for the HDF5 C++ bindings
  HDF5_Fortran_LIBRARIES - Required libraries for the HDF5 Fortran bindings
  HDF5_HL_LIBRARIES - Required libraries for the HDF5 high level API
  HDF5_Fortran_HL_LIBRARIES - Required libraries for the high level Fortran
                              bindings.
  HDF5_LIBRARIES - Required libraries for all requested bindings
  HDF5_FOUND - true if HDF5 was found on the system
  HDF5_LIBRARY_DIRS - the full set of library directories
  HDF5_IS_PARALLEL - Whether or not HDF5 was found with parallel IO support
  HDF5_C_COMPILER_EXECUTABLE - the path to the HDF5 C wrapper compiler
  HDF5_CXX_COMPILER_EXECUTABLE - the path to the HDF5 C++ wrapper compiler
  HDF5_Fortran_COMPILER_EXECUTABLE - the path to the HDF5 Fortran wrapper compiler
  HDF5_DIFF_EXECUTABLE - the path to the HDF5 dataset comparison tool
  ```

- **`FindHSPELL`**: Try to find Hspell

  Once done this will define

  ```
  HSPELL_FOUND - system has Hspell
  HSPELL_INCLUDE_DIR - the Hspell include directory
  HSPELL_LIBRARIES - The libraries needed to use Hspell
  HSPELL_DEFINITIONS - Compiler switches required for using Hspell


  HSPELL_VERSION_STRING - The version of Hspell found (x.y)
  HSPELL_MAJOR_VERSION  - the major version of Hspell
  HSPELL_MINOR_VERSION  - The minor version of Hspell
  ```

- **`FindHTMLHelp`**: This module looks for Microsoft HTML Help Compiler

  It defines:

  ```
   HTML_HELP_COMPILER     : full path to the Compiler (hhc.exe)
  ```

```
HTML_HELP_INCLUDE_PATH : include path to the API (htmlhelp.h)
HTML_HELP_LIBRARY      : full path to the library (htmlhelp.lib)
```

- **`FindHg`**:

  The module defines the following variables:

  ```
  HG_EXECUTABLE - path to mercurial command line client (hg)
  HG_FOUND - true if the command line client was found
  HG_VERSION_STRING - the version of mercurial found
  ```

  Example usage:

  ```
  find_package(Hg)
  if(HG_FOUND)
    message("hg found: ${HG_EXECUTABLE}")
  endif()
  ```

- **`FindITK`**: Find an ITK installation or build tree.
- **`FindIcotool`**: Find icotool

  This module looks for icotool. This module defines the following values:

  ```
  ICOTOOL_EXECUTABLE: the full path to the icotool tool.
  ICOTOOL_FOUND: True if icotool has been found.
  ICOTOOL_VERSION_STRING: the version of icotool found.
  ```

- **`FindImageMagick`**: Find the ImageMagick binary suite.

  This module will search for a set of ImageMagick tools specified as components in the FIND_PACKAGE call. Typical components include, but are not limited to (future versions of ImageMagick might have additional components not listed here):

  ```
  animate
  compare
  composite
  conjure
  convert
  display
  identify
  import
  mogrify
  montage
  stream
  ```

  If no component is specified in the FIND_PACKAGE call, then it only searches for the ImageMagick executable directory. This code defines the following variables:

  ```
  ImageMagick_FOUND                  - TRUE if all components are found.
  ImageMagick_EXECUTABLE_DIR         - Full path to executables directory.
  ImageMagick_<component>_FOUND      - TRUE if <component> is found.
  ImageMagick_<component>_EXECUTABLE - Full path to <component> executable.
  ImageMagick_VERSION_STRING         - the version of ImageMagick found
                                       (since CMake 2.8.8)
  ```

  ImageMagick_VERSION_STRING will not work for old versions like 5.2.3.

  There are also components for the following ImageMagick APIs:

  ```
  Magick++
  MagickWand
  MagickCore
  ```

  For these components the following variables are set:

  ```
  ImageMagick_FOUND                     - TRUE if all components are found.
  ImageMagick_INCLUDE_DIRS              - Full paths to all include dirs.
  ImageMagick_LIBRARIES                 - Full paths to all libraries.
  ImageMagick_<component>_FOUND         - TRUE if <component> is found.
  ImageMagick_<component>_INCLUDE_DIRS - Full path to <component> include dirs.
  ImageMagick_<component>_LIBRARIES     - Full path to <component> libraries.
  ```

Example Usages:

```
find_package(ImageMagick)
find_package(ImageMagick COMPONENTS convert)
find_package(ImageMagick COMPONENTS convert mogrify display)
find_package(ImageMagick COMPONENTS Magick++)
find_package(ImageMagick COMPONENTS Magick++ convert)
```

Note that the standard FIND_PACKAGE features are supported (i.e., QUIET, REQUIRED, etc.).

- **FindJNI**: Find JNI java libraries.

  This module finds if Java is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

```
JNI_INCLUDE_DIRS      = the include dirs to use
JNI_LIBRARIES         = the libraries to use
JNI_FOUND             = TRUE if JNI headers and libraries were found.
JAVA_AWT_LIBRARY      = the path to the jawt library
JAVA_JVM_LIBRARY      = the path to the jvm library
JAVA_INCLUDE_PATH     = the include path to jni.h
JAVA_INCLUDE_PATH2    = the include path to jni_md.h
JAVA_AWT_INCLUDE_PATH = the include path to jawt.h
```

- **FindJPEG**: Find JPEG

  Find the native JPEG includes and library This module defines

```
JPEG_INCLUDE_DIR, where to find jpeglib.h, etc.
JPEG_LIBRARIES, the libraries needed to use JPEG.
JPEG_FOUND, If false, do not try to use JPEG.
```

  also defined, but not for general use are

```
JPEG_LIBRARY, where to find the JPEG library.
```

- **FindJasper**: Try to find the Jasper JPEG2000 library

  Once done this will define

```
JASPER_FOUND - system has Jasper
JASPER_INCLUDE_DIR - the Jasper include directory
JASPER_LIBRARIES - the libraries needed to use Jasper
JASPER_VERSION_STRING - the version of Jasper found (since CMake 2.8.8)
```

- **FindJava**: Find Java

  This module finds if Java is installed and determines where the include files and libraries are. This code sets the following variables:

```
Java_JAVA_EXECUTABLE    = the full path to the Java runtime
Java_JAVAC_EXECUTABLE   = the full path to the Java compiler
Java_JAVAH_EXECUTABLE   = the full path to the Java header generator
Java_JAVADOC_EXECUTABLE = the full path to the Java documention generator
Java_JAR_EXECUTABLE     = the full path to the Java archiver
Java_VERSION_STRING     = Version of the package found (java version), eg. 1.6.0_12
Java_VERSION_MAJOR      = The major version of the package found.
Java_VERSION_MINOR      = The minor version of the package found.
Java_VERSION_PATCH      = The patch version of the package found.
Java_VERSION_TWEAK      = The tweak version of the package found (after '_')
Java_VERSION            = This is set to: $major.$minor.$patch(.$tweak)
```

The minimum required version of Java can be specified using the standard CMake syntax, e.g. find_package(Java 1.5)

NOTE: ${Java_VERSION} and ${Java_VERSION_STRING} are not guaranteed to be identical. For example some java version may return: Java_VERSION_STRING = 1.5.0_17 and Java_VERSION = 1.5.0.17

another example is the Java OEM, with: Java_VERSION_STRING = 1.6.0-oem and Java_VERSION = 1.6.0

For these components the following variables are set:

```
Java_FOUND                    - TRUE if all components are found.
Java_INCLUDE_DIRS             - Full paths to all include dirs.
Java_LIBRARIES                - Full paths to all libraries.
```

```
  Java_<component>_FOUND        - TRUE if <component> is found.
```

Example Usages:

```
  find_package(Java)
  find_package(Java COMPONENTS Runtime)
  find_package(Java COMPONENTS Development)
```

- **`FindKDE3`**: Find the KDE3 include and library dirs, KDE preprocessors and define a some macros

This module defines the following variables:

```
  KDE3_DEFINITIONS         - compiler definitions required for compiling KDE software
  KDE3_INCLUDE_DIR         - the KDE include directory
  KDE3_INCLUDE_DIRS        - the KDE and the Qt include directory, for use with include_directories()
  KDE3_LIB_DIR             - the directory where the KDE libraries are installed, for use with link_directories()
  QT_AND_KDECORE_LIBS      - this contains both the Qt and the kdecore library
  KDE3_DCOPIDL_EXECUTABLE  - the dcopidl executable
  KDE3_DCOPIDL2CPP_EXECUTABLE - the dcopidl2cpp executable
  KDE3_KCFGC_EXECUTABLE    - the kconfig_compiler executable
  KDE3_FOUND               - set to TRUE if all of the above has been found
```

The following user adjustable options are provided:

```
  KDE3_BUILD_TESTS - enable this to build KDE testcases
```

It also adds the following macros (from KDE3Macros.cmake) SRCS_VAR is always the variable which contains the list of source files for your application or library.

KDE3_AUTOMOC(file1 ... fileN)

```
    Call this if you want to have automatic moc file handling.
    This means if you include "foo.moc" in the source file foo.cpp
    a moc file for the header foo.h will be created automatically.
    You can set the property SKIP_AUTOMAKE using set_source_files_properties()
    to exclude some files in the list from being processed.
```

KDE3_ADD_MOC_FILES(SRCS_VAR file1 ... fileN)

```
    If you don't use the KDE3_AUTOMOC() macro, for the files
    listed here moc files will be created (named "foo.moc.cpp")
```

KDE3_ADD_DCOP_SKELS(SRCS_VAR header1.h ... headerN.h)

```
    Use this to generate DCOP skeletions from the listed headers.
```

KDE3_ADD_DCOP_STUBS(SRCS_VAR header1.h ... headerN.h)

```
     Use this to generate DCOP stubs from the listed headers.
```

KDE3_ADD_UI_FILES(SRCS_VAR file1.ui ... fileN.ui)

```
    Use this to add the Qt designer ui files to your application/library.
```

KDE3_ADD_KCFG_FILES(SRCS_VAR file1.kcfgc ... fileN.kcfgc)

```
    Use this to add KDE kconfig compiler files to your application/library.
```

KDE3_INSTALL_LIBTOOL_FILE(target)

```
    This will create and install a simple libtool file for the given target.
```

KDE3_ADD_EXECUTABLE(name file1 ... fileN)

```
Currently identical to add_executable(), may provide some advanced features in the future.
```

KDE3_ADD_KPART(name [WITH_PREFIX] file1 ... fileN)

```
Create a KDE plugin (KPart, kioslave, etc.) from the given source files.
If WITH_PREFIX is given, the resulting plugin will have the prefix "lib", otherwise it won't.
It creates and installs an appropriate libtool la-file.
```

KDE3_ADD_KDEINIT_EXECUTABLE(name file1 ... fileN)

```
Create a KDE application in the form of a module loadable via kdeinit.
A library named kdeinit_<name> will be created and a small executable which links to it.
```

The option KDE3_ENABLE_FINAL to enable all-in-one compilation is no longer supported.

Author: Alexander Neundorf <neundorf@kde.org>

- **`FindKDE4`**:

  Find KDE4 and provide all necessary variables and macros to compile software for it. It looks for KDE 4 in the following directories in the given order:

  ```
  CMAKE_INSTALL_PREFIX
  KDEDIRS
  /opt/kde4
  ```

  Please look in FindKDE4Internal.cmake and KDE4Macros.cmake for more information. They are installed with the KDE 4 libraries in $KDEDIRS/share/apps/cmake/modules/.

  Author: Alexander Neundorf <neundorf@kde.org>

- **`FindLAPACK`**: Find LAPACK library

  This module finds an installed fortran library that implements the LAPACK linear-algebra interface (see http://www.netlib.org/lapack/).

  The approach follows that taken for the autoconf macro file, acx_lapack.m4 (distributed at http://ac-archive.sourceforge.net/ac-archive/acx_lapack.html).

  This module sets the following variables:

  ```
  LAPACK_FOUND - set to true if a library implementing the LAPACK interface
    is found
  LAPACK_LINKER_FLAGS - uncached list of required linker flags (excluding -l
    and -L).
  LAPACK_LIBRARIES - uncached list of libraries (using full path name) to
    link against to use LAPACK
  LAPACK95_LIBRARIES - uncached list of libraries (using full path name) to
    link against to use LAPACK95
  LAPACK95_FOUND - set to true if a library implementing the LAPACK f95
    interface is found
  BLA_STATIC  if set on this determines what kind of linkage we do (static)
  BLA_VENDOR  if set checks only the specified vendor, if not set checks
     all the possibilities
  BLA_F95     if set on tries to find the f95 interfaces for BLAS/LAPACK
  ```

- **`FindLATEX`**: Find Latex

  This module finds if Latex is installed and determines where the executables are. This code sets the following variables:

  ```
  LATEX_COMPILER:       path to the LaTeX compiler
  PDFLATEX_COMPILER:    path to the PdfLaTeX compiler
  BIBTEX_COMPILER:      path to the BibTeX compiler
  MAKEINDEX_COMPILER:   path to the MakeIndex compiler
  DVIPS_CONVERTER:      path to the DVIPS converter
  PS2PDF_CONVERTER:     path to the PS2PDF converter
  LATEX2HTML_CONVERTER: path to the LaTeX2Html converter
  ```

- **FindLibArchive**: Find libarchive library and headers

  The module defines the following variables:

  ```
  LibArchive_FOUND        - true if libarchive was found
  LibArchive_INCLUDE_DIRS - include search path
  LibArchive_LIBRARIES    - libraries to link
  LibArchive_VERSION      - libarchive 3-component version number
  ```

- **FindLibLZMA**: Find LibLZMA

  Find LibLZMA headers and library

  ```
  LIBLZMA_FOUND             - True if liblzma is found.
  LIBLZMA_INCLUDE_DIRS      - Directory where liblzma headers are located.
  LIBLZMA_LIBRARIES         - Lzma libraries to link against.
  LIBLZMA_HAS_AUTO_DECODER  - True if lzma_auto_decoder() is found (required).
  LIBLZMA_HAS_EASY_ENCODER  - True if lzma_easy_encoder() is found (required).
  LIBLZMA_HAS_LZMA_PRESET   - True if lzma_lzma_preset() is found (required).
  LIBLZMA_VERSION_MAJOR     - The major version of lzma
  LIBLZMA_VERSION_MINOR     - The minor version of lzma
  LIBLZMA_VERSION_PATCH     - The patch version of lzma
  LIBLZMA_VERSION_STRING    - version number as a string (ex: "5.0.3")
  ```

- **FindLibXml2**: Try to find the LibXml2 xml processing library

  Once done this will define

  ```
  LIBXML2_FOUND - System has LibXml2
  LIBXML2_INCLUDE_DIR - The LibXml2 include directory
  LIBXML2_LIBRARIES - The libraries needed to use LibXml2
  LIBXML2_DEFINITIONS - Compiler switches required for using LibXml2
  LIBXML2_XMLLINT_EXECUTABLE - The XML checking tool xmllint coming with LibXml2
  LIBXML2_VERSION_STRING - the version of LibXml2 found (since CMake 2.8.8)
  ```

- **FindLibXslt**: Try to find the LibXslt library

  Once done this will define

  ```
  LIBXSLT_FOUND - system has LibXslt
  LIBXSLT_INCLUDE_DIR - the LibXslt include directory
  LIBXSLT_LIBRARIES - Link these to LibXslt
  LIBXSLT_DEFINITIONS - Compiler switches required for using LibXslt
  LIBXSLT_VERSION_STRING - version of LibXslt found (since CMake 2.8.8)
  ```

  Additionally, the following two variables are set (but not required for using xslt):

  ```
  LIBXSLT_EXSLT_LIBRARIES - Link to these if you need to link against the exslt library
  LIBXSLT_XSLTPROC_EXECUTABLE - Contains the full path to the xsltproc executable if found
  ```

- **FindLua50**:

  Locate Lua library This module defines

  ```
  LUA50_FOUND, if false, do not try to link to Lua
  LUA_LIBRARIES, both lua and lualib
  LUA_INCLUDE_DIR, where to find lua.h and lualib.h (and probably lauxlib.h)
  ```

  Note that the expected include convention is

  ```
  #include "lua.h"
  ```

  and not

  ```
  #include <lua/lua.h>
  ```

  This is because, the lua location is not standardized and may exist in locations other than lua/

- **FindLua51**:

  Locate Lua library This module defines

  ```
  LUA51_FOUND, if false, do not try to link to Lua
  LUA_LIBRARIES
  LUA_INCLUDE_DIR, where to find lua.h
  LUA_VERSION_STRING, the version of Lua found (since CMake 2.8.8)
  ```

  Note that the expected include convention is

```
  #include "lua.h"
```

and not

```
  #include <lua/lua.h>
```

This is because, the lua location is not standardized and may exist in locations other than lua/

- **`FindMFC`**: Find MFC on Windows

  Find the native MFC - i.e. decide if an application can link to the MFC libraries.

```
  MFC_FOUND - Was MFC support found
```

  You don't need to include anything or link anything to use it.

- **`FindMPEG`**: Find the native MPEG includes and library

  This module defines

```
  MPEG_INCLUDE_DIR, where to find MPEG.h, etc.
  MPEG_LIBRARIES, the libraries required to use MPEG.
  MPEG_FOUND, If false, do not try to use MPEG.
```

  also defined, but not for general use are

```
  MPEG_mpeg2_LIBRARY, where to find the MPEG library.
  MPEG_vo_LIBRARY, where to find the vo library.
```

- **`FindMPEG2`**: Find the native MPEG2 includes and library

  This module defines

```
  MPEG2_INCLUDE_DIR, path to mpeg2dec/mpeg2.h, etc.
  MPEG2_LIBRARIES, the libraries required to use MPEG2.
  MPEG2_FOUND, If false, do not try to use MPEG2.
```

  also defined, but not for general use are

```
  MPEG2_mpeg2_LIBRARY, where to find the MPEG2 library.
  MPEG2_vo_LIBRARY, where to find the vo library.
```

- **`FindMPI`**: Find a Message Passing Interface (MPI) implementation

  The Message Passing Interface (MPI) is a library used to write high-performance distributed-memory parallel applications, and is typically deployed on a cluster. MPI is a standard interface (defined by the MPI forum) for which many implementations are available. All of them have somewhat different include paths, libraries to link against, etc., and this module tries to smooth out those differences.

  === Variables ===

  This module will set the following variables per language in your project, where <lang> is one of C, CXX, or Fortran:

```
  MPI_<lang>_FOUND           TRUE if FindMPI found MPI flags for <lang>
  MPI_<lang>_COMPILER        MPI Compiler wrapper for <lang>
  MPI_<lang>_COMPILE_FLAGS   Compilation flags for MPI programs
  MPI_<lang>_INCLUDE_PATH    Include path(s) for MPI header
  MPI_<lang>_LINK_FLAGS      Linking flags for MPI programs
  MPI_<lang>_LIBRARIES       All libraries to link MPI programs against
```

  Additionally, FindMPI sets the following variables for running MPI programs from the command line:

```
  MPIEXEC                    Executable for running MPI programs
  MPIEXEC_NUMPROC_FLAG       Flag to pass to MPIEXEC before giving
                             it the number of processors to run on
  MPIEXEC_PREFLAGS           Flags to pass to MPIEXEC directly
                             before the executable to run.
  MPIEXEC_POSTFLAGS          Flags to pass to MPIEXEC after other flags
```

  === Usage ===

  To use this module, simply call FindMPI from a CMakeLists.txt file, or run find_package(MPI), then run CMake. If you are happy with the auto- detected configuration for your language, then you're done. If not, you have two options:

```
  1. Set MPI_<lang>_COMPILER to the MPI wrapper (mpicc, etc.) of your
     choice and reconfigure.  FindMPI will attempt to determine all the
     necessary variables using THAT compiler's compile and link flags.
  2. If this fails, or if your MPI implementation does not come with
     a compiler wrapper, then set both MPI_<lang>_LIBRARIES and
     MPI_<lang>_INCLUDE_PATH.  You may also set any other variables
     listed above, but these two are required.  This will circumvent
     autodetection entirely.
```

When configuration is successful, MPI_<lang>_COMPILER will be set to the compiler wrapper for <lang>, if it was found. MPI_<lang>_FOUND and other variables above will be set if any MPI implementation was found for <lang>, regardless of whether a compiler was found.

When using MPIEXEC to execute MPI applications, you should typically use all of the MPIEXEC flags as follows:

```
${MPIEXEC} ${MPIEXEC_NUMPROC_FLAG} PROCS
  ${MPIEXEC_PREFLAGS} EXECUTABLE ${MPIEXEC_POSTFLAGS} ARGS
```

where PROCS is the number of processors on which to execute the program, EXECUTABLE is the MPI program, and ARGS are the arguments to pass to the MPI program.

=== Backward Compatibility ===

For backward compatibility with older versions of FindMPI, these variables are set, but deprecated:

```
MPI_FOUND           MPI_COMPILER        MPI_LIBRARY
MPI_COMPILE_FLAGS   MPI_INCLUDE_PATH    MPI_EXTRA_LIBRARY
MPI_LINK_FLAGS      MPI_LIBRARIES
```

In new projects, please use the MPI_<lang>_XXX equivalents.

- **`FindMatlab`**: this module looks for Matlab

  Defines:

  ```
  MATLAB_INCLUDE_DIR: include path for mex.h, engine.h
  MATLAB_LIBRARIES:   required libraries: libmex, etc
  MATLAB_MEX_LIBRARY: path to libmex.lib
  MATLAB_MX_LIBRARY:  path to libmx.lib
  MATLAB_ENG_LIBRARY: path to libeng.lib
  ```

- **`FindMotif`**: Try to find Motif (or lesstif)

  Once done this will define:

  ```
  MOTIF_FOUND        - system has MOTIF
  MOTIF_INCLUDE_DIR  - include paths to use Motif
  MOTIF_LIBRARIES    - Link these to use Motif
  ```

- **`FindOpenAL`**:

  Locate OpenAL This module defines OPENAL_LIBRARY OPENAL_FOUND, if false, do not try to link to OpenAL OPENAL_INCLUDE_DIR, where to find the headers

  $OPENALDIR is an environment variable that would correspond to the ./configure --prefix=$OPENALDIR used in building OpenAL.

  Created by Eric Wing. This was influenced by the FindSDL.cmake module.

- **`FindOpenGL`**: Try to find OpenGL

  Once done this will define

  ```
  OPENGL_FOUND        - system has OpenGL
  OPENGL_XMESA_FOUND  - system has XMESA
  OPENGL_GLU_FOUND    - system has GLU
  OPENGL_INCLUDE_DIR  - the GL include directory
  OPENGL_LIBRARIES    - Link these to use OpenGL and GLU
  ```

  If you want to use just GL you can use these values

  ```
  OPENGL_gl_LIBRARY   - Path to OpenGL Library
  OPENGL_glu_LIBRARY  - Path to GLU Library
  ```

  On OSX default to using the framework version of opengl People will have to change the cache values of OPENGL_glu_LIBRARY and OPENGL_gl_LIBRARY to use OpenGL with X11 on OSX

- **`FindOpenMP`**: Finds OpenMP support

  This module can be used to detect OpenMP support in a compiler. If the compiler supports OpenMP, the flags required to compile with OpenMP support are returned in variables for the different languages. The variables may be empty if the compiler does not need a special flag to support OpenMP.

  The following variables are set:

  ```
  OpenMP_C_FLAGS - flags to add to the C compiler for OpenMP support
  OpenMP_CXX_FLAGS - flags to add to the CXX compiler for OpenMP support
  OPENMP_FOUND - true if openmp is detected
  ```

Supported compilers can be found at http://openmp.org/wp/openmp-compilers/

- **`FindOpenSSL`**: Try to find the OpenSSL encryption library

  Once done this will define

  ```
  OPENSSL_ROOT_DIR - Set this variable to the root installation of OpenSSL
  ```

  Read-Only variables:

  ```
  OPENSSL_FOUND - system has the OpenSSL library
  OPENSSL_INCLUDE_DIR - the OpenSSL include directory
  OPENSSL_LIBRARIES - The libraries needed to use OpenSSL
  OPENSSL_VERSION - This is set to $major.$minor.$revision$path (eg. 0.9.8s)
  ```

- **`FindOpenSceneGraph`**: Find OpenSceneGraph

  This module searches for the OpenSceneGraph core "osg" library as well as OpenThreads, and whatever additional COMPONENTS (nodekits) that you specify.

  ```
  See http://www.openscenegraph.org
  ```

  NOTE: To use this module effectively you must either require CMake >= 2.6.3 with cmake_minimum_required(VERSION 2.6.3) or download and place FindOpenThreads.cmake, Findosg_functions.cmake, Findosg.cmake, and Find<etc>.cmake files into your CMAKE_MODULE_PATH.

  ```
  ====================================
  ```

  This module accepts the following variables (note mixed case)

  ```
  OpenSceneGraph_DEBUG - Enable debugging output


  OpenSceneGraph_MARK_AS_ADVANCED - Mark cache variables as advanced
                                    automatically
  ```

  The following environment variables are also respected for finding the OSG and it's various components. CMAKE_PREFIX_PATH can also be used for this (see find_library() CMake documentation).

  ```
  <MODULE>_DIR (where MODULE is of the form "OSGVOLUME" and there is a FindosgVolume.cmake file)
  OSG_DIR
  OSGDIR
  OSG_ROOT
  ```

  [CMake 2.8.10]: The CMake variable OSG_DIR can now be used as well to influence detection, instead of needing to specify an environment variable.

  This module defines the following output variables:

  ```
  OPENSCENEGRAPH_FOUND - Was the OSG and all of the specified components found?


  OPENSCENEGRAPH_VERSION - The version of the OSG which was found


    OPENSCENEGRAPH_INCLUDE_DIRS - Where to find the headers


    OPENSCENEGRAPH_LIBRARIES - The OSG libraries
  ```

  ```
  ==================================== Example Usage:

    find_package(OpenSceneGraph 2.0.0 REQUIRED osgDB osgUtil)
        # libOpenThreads & libosg automatically searched
    include_directories(${OPENSCENEGRAPH_INCLUDE_DIRS})


    add_executable(foo foo.cc)
    target_link_libraries(foo ${OPENSCENEGRAPH_LIBRARIES})
  ```

- **`FindOpenThreads`**:

  OpenThreads is a C++ based threading library. Its largest userbase seems to OpenSceneGraph so you might notice I accept OSGDIR as an environment path. I consider this part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module.

  Locate OpenThreads This module defines OPENTHREADS_LIBRARY OPENTHREADS_FOUND, if false, do not try to link to OpenThreads OPENTHREADS_INCLUDE_DIR, where to find the headers

  $OPENTHREADS_DIR is an environment variable that would correspond to the ./configure --prefix=$OPENTHREADS_DIR used in building osg.

  [CMake 2.8.10]: The CMake variables OPENTHREADS_DIR or OSG_DIR can now be used as well to influence detection, instead of needing to specify an environment variable.

  Created by Eric Wing.

- **`FindPHP4`**: Find PHP4

  This module finds if PHP4 is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

  ```
  PHP4_INCLUDE_PATH       = path to where php.h can be found
  PHP4_EXECUTABLE         = full path to the php4 binary
  ```

- **`FindPNG`**: Find the native PNG includes and library

  This module searches libpng, the library for working with PNG images.

  It defines the following variables

  ```
  PNG_INCLUDE_DIRS, where to find png.h, etc.
  PNG_LIBRARIES, the libraries to link against to use PNG.
  PNG_DEFINITIONS - You should add_definitons(${PNG_DEFINITIONS}) before compiling code that includes png library files.
  PNG_FOUND, If false, do not try to use PNG.
  PNG_VERSION_STRING - the version of the PNG library found (since CMake 2.8.8)
  ```

  Also defined, but not for general use are

  ```
  PNG_LIBRARY, where to find the PNG library.
  ```

  For backward compatiblity the variable PNG_INCLUDE_DIR is also set. It has the same value as PNG_INCLUDE_DIRS.

  Since PNG depends on the ZLib compression library, none of the above will be defined unless ZLib can be found.

- **`FindPackageHandleStandardArgs`**:

  FIND_PACKAGE_HANDLE_STANDARD_ARGS(<name> ...)

  This function is intended to be used in FindXXX.cmake modules files. It handles the REQUIRED, QUIET and version-related arguments to find_package(). It also sets the <packagename>_FOUND variable. The package is considered found if all variables <var1>... listed contain valid results, e.g. valid filepaths.

  There are two modes of this function. The first argument in both modes is the name of the Find-module where it is called (in original casing).

  The first simple mode looks like this:

  ```
  FIND_PACKAGE_HANDLE_STANDARD_ARGS(<name> (DEFAULT_MSG|"Custom failure message") <var1>...<varN> )
  ```

  If the variables <var1> to <varN> are all valid, then <UPPERCASED_NAME>_FOUND will be set to TRUE. If DEFAULT_MSG is given as second argument, then the function will generate itself useful success and error messages. You can also supply a custom error message for the failure case. This is not recommended.

  The second mode is more powerful and also supports version checking:

  ```
  FIND_PACKAGE_HANDLE_STANDARD_ARGS(NAME [FOUND_VAR <resultVar>]
                                         [REQUIRED_VARS <var1>...<varN>]
                                         [VERSION_VAR   <versionvar>]
                                         [HANDLE_COMPONENTS]
                                         [CONFIG_MODE]
                                         [FAIL_MESSAGE "Custom failure message"] )
  ```

  In this mode, the name of the result-variable can be set either to either <UPPERCASED_NAME>_FOUND or <OriginalCase_Name>_FOUND using the FOUND_VAR option. Other names for the result-variable are not allowed. So for a Find-module named FindFooBar.cmake, the two possible names are FooBar_FOUND and FOOBAR_FOUND. It is recommended

to use the original case version. If the FOUND_VAR option is not used, the default is <UPPERCASED_NAME>_FOUND.

As in the simple mode, if <var1> through <varN> are all valid, <packagename>_FOUND will be set to TRUE. After REQUIRED_VARS the variables which are required for this package are listed. Following VERSION_VAR the name of the variable can be specified which holds the version of the package which has been found. If this is done, this version will be checked against the (potentially) specified required version used in the find_package() call. The EXACT keyword is also handled. The default messages include information about the required version and the version which has been actually found, both if the version is ok or not. If the package supports components, use the HANDLE_COMPONENTS option to enable handling them. In this case, find_package_handle_standard_args() will report which components have been found and which are missing, and the <packagename>_FOUND variable will be set to FALSE if any of the required components (i.e. not the ones listed after OPTIONAL_COMPONENTS) are missing. Use the option CONFIG_MODE if your FindXXX.cmake module is a wrapper for a find_package(... NO_MODULE) call. In this case VERSION_VAR will be set to <NAME>_VERSION and the macro will automatically check whether the Config module was found. Via FAIL_MESSAGE a custom failure message can be specified, if this is not used, the default message will be displayed.

Example for mode 1:

```
find_package_handle_standard_args(LibXml2  DEFAULT_MSG  LIBXML2_LIBRARY LIBXML2_INCLUDE_DIR)
```

LibXml2 is considered to be found, if both LIBXML2_LIBRARY and LIBXML2_INCLUDE_DIR are valid. Then also LIBXML2_FOUND is set to TRUE. If it is not found and REQUIRED was used, it fails with FATAL_ERROR, independent whether QUIET was used or not. If it is found, success will be reported, including the content of <var1>. On repeated Cmake runs, the same message won't be printed again.

Example for mode 2:

```
find_package_handle_standard_args(LibXslt FOUND_VAR LibXslt_FOUND
                                          REQUIRED_VARS LibXslt_LIBRARIES LibXslt_INCLUDE_DIRS
                                          VERSION_VAR LibXslt_VERSION_STRING)
```

In this case, LibXslt is considered to be found if the variable(s) listed after REQUIRED_VAR are all valid, i.e. LibXslt_LIBRARIES and LibXslt_INCLUDE_DIRS in this case. The result will then be stored in LibXslt_FOUND . Also the version of LibXslt will be checked by using the version contained in LibXslt_VERSION_STRING. Since no FAIL_MESSAGE is given, the default messages will be printed.

Another example for mode 2:

```
find_package(Automoc4 QUIET NO_MODULE HINTS /opt/automoc4)
find_package_handle_standard_args(Automoc4  CONFIG_MODE)
```

In this case, FindAutmoc4.cmake wraps a call to find_package(Automoc4 NO_MODULE) and adds an additional search directory for automoc4. Here the result will be stored in AUTOMOC4_FOUND. The following FIND_PACKAGE_HANDLE_STANDARD_ARGS() call produces a proper success/error message.

- **`FindPackageMessage`**:

  FIND_PACKAGE_MESSAGE(<name> "message for user" "find result details")

  This macro is intended to be used in FindXXX.cmake modules files. It will print a message once for each unique find result. This is useful for telling the user where a package was found. The first argument specifies the name (XXX) of the package. The second argument specifies the message to display. The third argument lists details about the find result so that if they change the message will be displayed again. The macro also obeys the QUIET argument to the find_package command.

  Example:

  ```
  if(X11_FOUND)
    FIND_PACKAGE_MESSAGE(X11 "Found X11: ${X11_X11_LIB}"
      "[${X11_X11_LIB}][${X11_INCLUDE_DIR}]")
  else()
   ...
  endif()
  ```

- **`FindPerl`**: Find perl

  this module looks for Perl

  ```
  PERL_EXECUTABLE     - the full path to perl
  PERL_FOUND          - If false, don't attempt to use perl.
  PERL_VERSION_STRING - version of perl found (since CMake 2.8.8)
  ```

- **`FindPerlLibs`**: Find Perl libraries

  This module finds if PERL is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

  ```
  PERLLIBS_FOUND    = True if perl.h & libperl were found
  PERL_INCLUDE_PATH = path to where perl.h is found
  PERL_LIBRARY      = path to libperl
  PERL_EXECUTABLE   = full path to the perl binary
  ```

The minimum required version of Perl can be specified using the standard syntax, e.g. find_package(PerlLibs 6.0)

```
The following variables are also available if needed
(introduced after CMake 2.6.4)



PERL_SITESEARCH    = path to the sitesearch install dir
PERL_SITELIB       = path to the sitelib install directory
PERL_VENDORARCH    = path to the vendor arch install directory
PERL_VENDORLIB     = path to the vendor lib install directory
PERL_ARCHLIB       = path to the arch lib install directory
PERL_PRIVLIB       = path to the priv lib install directory
PERL_EXTRA_C_FLAGS = Compilation flags used to build perl
```

- **FindPhysFS**:

  Locate PhysFS library This module defines PHYSFS_LIBRARY, the name of the library to link against PHYSFS_FOUND, if false, do not try to link to PHYSFS PHYSFS_INCLUDE_DIR, where to find physfs.h

  $PHYSFSDIR is an environment variable that would correspond to the ./configure --prefix=$PHYSFSDIR used in building PHYSFS.

  Created by Eric Wing.

- **FindPike**: Find Pike

  This module finds if PIKE is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

  ```
  PIKE_INCLUDE_PATH       = path to where program.h is found
  PIKE_EXECUTABLE         = full path to the pike binary
  ```

- **FindPkgConfig**: a pkg-config module for CMake

  Usage:

  ```
  pkg_check_modules(<PREFIX> [REQUIRED] [QUIET] <MODULE> [<MODULE>]*)
    checks for all the given modules



  pkg_search_module(<PREFIX> [REQUIRED] [QUIET] <MODULE> [<MODULE>]*)
    checks for given modules and uses the first working one
  ```

  When the 'REQUIRED' argument was set, macros will fail with an error when module(s) could not be found

  When the 'QUIET' argument is set, no status messages will be printed.

  It sets the following variables:

  ```
  PKG_CONFIG_FOUND          ... if pkg-config executable was found
  PKG_CONFIG_EXECUTABLE     ... pathname of the pkg-config program
  PKG_CONFIG_VERSION_STRING ... the version of the pkg-config program found
                                (since CMake 2.8.8)
  ```

  For the following variables two sets of values exist; first one is the common one and has the given PREFIX. The second set contains flags which are given out when pkgconfig was called with the '--static' option.

  ```
  <XPREFIX>_FOUND           ... set to 1 if module(s) exist
  <XPREFIX>_LIBRARIES       ... only the libraries (w/o the '-l')
  <XPREFIX>_LIBRARY_DIRS    ... the paths of the libraries (w/o the '-L')
  <XPREFIX>_LDFLAGS         ... all required linker flags
  <XPREFIX>_LDFLAGS_OTHER   ... all other linker flags
  <XPREFIX>_INCLUDE_DIRS    ... the '-I' preprocessor flags (w/o the '-I')
  <XPREFIX>_CFLAGS          ... all required cflags
  <XPREFIX>_CFLAGS_OTHER    ... the other compiler flags



  <XPREFIX> = <PREFIX>        for common case
  <XPREFIX> = <PREFIX>_STATIC for static linking
  ```

There are some special variables whose prefix depends on the count of given modules. When there is only one module, <PREFIX> stays unchanged. When there are multiple modules, the prefix will be changed to <PREFIX>_<MODNAME>:

```
<XPREFIX>_VERSION    ... version of the module
<XPREFIX>_PREFIX     ... prefix-directory of the module
<XPREFIX>_INCLUDEDIR ... include-dir of the module
<XPREFIX>_LIBDIR     ... lib-dir of the module


<XPREFIX> = <PREFIX>  when |MODULES| == 1, else
<XPREFIX> = <PREFIX>_<MODNAME>
```

A <MODULE> parameter can have the following formats:

```
{MODNAME}            ... matches any version
{MODNAME}>={VERSION} ... at least version <VERSION> is required
{MODNAME}={VERSION}  ... exactly version <VERSION> is required
{MODNAME}<={VERSION} ... modules must not be newer than <VERSION>
```

Examples

```
pkg_check_modules (GLIB2   glib-2.0)


pkg_check_modules (GLIB2   glib-2.0>=2.10)
  requires at least version 2.10 of glib2 and defines e.g.
    GLIB2_VERSION=2.10.3


pkg_check_modules (FOO     glib-2.0>=2.10 gtk+-2.0)
  requires both glib2 and gtk2, and defines e.g.
    FOO_glib-2.0_VERSION=2.10.3
    FOO_gtk+-2.0_VERSION=2.8.20


pkg_check_modules (XRENDER REQUIRED xrender)
  defines e.g.:
    XRENDER_LIBRARIES=Xrender;X11
    XRENDER_STATIC_LIBRARIES=Xrender;X11;pthread;Xau;Xdmcp


pkg_search_module (BAR     libxml-2.0 libxml2 libxml>=2)
```

- **`FindPostgreSQL`**: Find the PostgreSQL installation.

  In Windows, we make the assumption that, if the PostgreSQL files are installed, the default directory will be C:\Program Files\PostgreSQL.

  This module defines

  ```
  PostgreSQL_LIBRARIES - the PostgreSQL libraries needed for linking
  PostgreSQL_INCLUDE_DIRS - the directories of the PostgreSQL headers
  PostgreSQL_VERSION_STRING - the version of PostgreSQL found (since CMake 2.8.8)
  ```

- **`FindProducer`**:

  Though Producer isn't directly part of OpenSceneGraph, its primary user is OSG so I consider this part of the Findosg* suite used to find OpenSceneGraph components. You'll notice that I accept OSGDIR as an environment path.

  Each component is separate and you must opt in to each module. You must also opt into OpenGL (and OpenThreads?) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate Producer This module defines PRODUCER_LIBRARY PRODUCER_FOUND, if false, do not try to link to Producer PRODUCER_INCLUDE_DIR, where to find the headers

  $PRODUCER_DIR is an environment variable that would correspond to the ./configure --prefix=$PRODUCER_DIR used in building osg.

  Created by Eric Wing.

- **FindProtobuf**:

  Locate and configure the Google Protocol Buffers library.

  The following variables can be set and are optional:

```
  PROTOBUF_SRC_ROOT_FOLDER - When compiling with MSVC, if this cache variable is set
                             the protobuf-default VS project build locations
                             (vsprojects/Debug & vsprojects/Release) will be searched
                             for libraries and binaries.


  PROTOBUF_IMPORT_DIRS     - List of additional directories to be searched for
                             imported .proto files. (New in CMake 2.8.8)
```

  Defines the following variables:

```
  PROTOBUF_FOUND - Found the Google Protocol Buffers library (libprotobuf & header files)
  PROTOBUF_INCLUDE_DIRS - Include directories for Google Protocol Buffers
  PROTOBUF_LIBRARIES - The protobuf libraries
```

  [New in CMake 2.8.5]

```
  PROTOBUF_PROTOC_LIBRARIES - The protoc libraries
  PROTOBUF_LITE_LIBRARIES - The protobuf-lite libraries
```

  The following cache variables are also available to set or use:

```
  PROTOBUF_LIBRARY - The protobuf library
  PROTOBUF_PROTOC_LIBRARY   - The protoc library
  PROTOBUF_INCLUDE_DIR - The include directory for protocol buffers
  PROTOBUF_PROTOC_EXECUTABLE - The protoc compiler
```

  [New in CMake 2.8.5]

```
  PROTOBUF_LIBRARY_DEBUG - The protobuf library (debug)
  PROTOBUF_PROTOC_LIBRARY_DEBUG   - The protoc library (debug)
  PROTOBUF_LITE_LIBRARY - The protobuf lite library
  PROTOBUF_LITE_LIBRARY_DEBUG - The protobuf lite library (debug)


  ============================================================================
  Example:


   find_package(Protobuf REQUIRED)
   include_directories(${PROTOBUF_INCLUDE_DIRS})


   include_directories(${CMAKE_CURRENT_BINARY_DIR})
   PROTOBUF_GENERATE_CPP(PROTO_SRCS PROTO_HDRS foo.proto)
   add_executable(bar bar.cc ${PROTO_SRCS} ${PROTO_HDRS})
   target_link_libraries(bar ${PROTOBUF_LIBRARIES})


  NOTE: You may need to link against pthreads, depending
        on the platform.


  NOTE: The PROTOBUF_GENERATE_CPP macro & add_executable() or add_library()
        calls only work properly within the same directory.


  ============================================================================


  PROTOBUF_GENERATE_CPP (public function)

    SRCS = Variable to define with autogenerated
           source files
    HDRS = Variable to define with autogenerated
```

```
        header files
 ARGN = proto files


=============================================================================
```

- **`FindPythonInterp`**: Find python interpreter

  This module finds if Python interpreter is installed and determines where the executables are. This code sets the following variables:

```
PYTHONINTERP_FOUND         - Was the Python executable found
PYTHON_EXECUTABLE          - path to the Python interpreter


PYTHON_VERSION_STRING      - Python version found e.g. 2.5.2
PYTHON_VERSION_MAJOR       - Python major version found e.g. 2
PYTHON_VERSION_MINOR       - Python minor version found e.g. 5
PYTHON_VERSION_PATCH       - Python patch version found e.g. 2
```

  The Python_ADDITIONAL_VERSIONS variable can be used to specify a list of version numbers that should be taken into account when searching for Python. You need to set this variable before calling find_package(PythonInterp).

- **`FindPythonLibs`**: Find python libraries

  This module finds if Python is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

```
PYTHONLIBS_FOUND           - have the Python libs been found
PYTHON_LIBRARIES           - path to the python library
PYTHON_INCLUDE_PATH        - path to where Python.h is found (deprecated)
PYTHON_INCLUDE_DIRS        - path to where Python.h is found
PYTHON_DEBUG_LIBRARIES     - path to the debug library (deprecated)
PYTHONLIBS_VERSION_STRING  - version of the Python libs found (since CMake 2.8.8)
```

  The Python_ADDITIONAL_VERSIONS variable can be used to specify a list of version numbers that should be taken into account when searching for Python. You need to set this variable before calling find_package(PythonLibs).

  If you'd like to specify the installation of Python to use, you should modify the following cache variables:

```
PYTHON_LIBRARY             - path to the python library
PYTHON_INCLUDE_DIR         - path to where Python.h is found
```

- **`FindQt`**: Searches for all installed versions of Qt.

  This should only be used if your project can work with multiple versions of Qt. If not, you should just directly use FindQt4 or FindQt3. If multiple versions of Qt are found on the machine, then The user must set the option DESIRED_QT_VERSION to the version they want to use. If only one version of qt is found on the machine, then the DESIRED_QT_VERSION is set to that version and the matching FindQt3 or FindQt4 module is included. Once the user sets DESIRED_QT_VERSION, then the FindQt3 or FindQt4 module is included.

```
QT_REQUIRED if this is set to TRUE then if CMake can
            not find Qt4 or Qt3 an error is raised
            and a message is sent to the user.


DESIRED_QT_VERSION OPTION is created
QT4_INSTALLED is set to TRUE if qt4 is found.
QT3_INSTALLED is set to TRUE if qt3 is found.
```

- **`FindQt3`**: Locate Qt include paths and libraries

  This module defines:

```
QT_INCLUDE_DIR    - where to find qt.h, etc.
QT_LIBRARIES      - the libraries to link against to use Qt.
QT_DEFINITIONS    - definitions to use when
                    compiling code that uses Qt.
QT_FOUND          - If false, don't try to use Qt.
QT_VERSION_STRING - the version of Qt found
```

  If you need the multithreaded version of Qt, set QT_MT_REQUIRED to TRUE

  Also defined, but not for general use are:

```
QT_MOC_EXECUTABLE, where to find the moc tool.
QT_UIC_EXECUTABLE, where to find the uic tool.
QT_QT_LIBRARY, where to find the Qt library.
QT_QTMAIN_LIBRARY, where to find the qtmain
 library. This is only required by Qt3 on Windows.
```

- **`FindQt4`**: Find Qt 4

  This module can be used to find Qt4. The most important issue is that the Qt4 qmake is available via the system path. This qmake is then used to detect basically everything else. This module defines a number of key variables and macros. The variable QT_USE_FILE is set which is the path to a CMake file that can be included to compile Qt 4 applications and libraries. It sets up the compilation environment for include directories, preprocessor defines and populates a QT_LIBRARIES variable.

  Typical usage could be something like:

  ```
  find_package(Qt4 4.4.3 REQUIRED QtCore QtGui QtXml)
  include(${QT_USE_FILE})
  add_executable(myexe main.cpp)
  target_link_libraries(myexe ${QT_LIBRARIES})
  ```

The minimum required version can be specified using the standard find_package()-syntax (see example above). For compatibility with older versions of FindQt4.cmake it is also possible to set the variable QT_MIN_VERSION to the minimum required version of Qt4 before the find_package(Qt4) command. If both are used, the version used in the find_package() command overrides the one from QT_MIN_VERSION.

When using the components argument, QT_USE_QT* variables are automatically set for the QT_USE_FILE to pick up. If one wishes to manually set them, the available ones to set include:

```
QT_DONT_USE_QTCORE
QT_DONT_USE_QTGUI
QT_USE_QT3SUPPORT
QT_USE_QTASSISTANT
QT_USE_QAXCONTAINER
QT_USE_QAXSERVER
QT_USE_QTDESIGNER
QT_USE_QTMOTIF
QT_USE_QTMAIN
QT_USE_QTMULTIMEDIA
QT_USE_QTNETWORK
QT_USE_QTNSPLUGIN
QT_USE_QTOPENGL
QT_USE_QTSQL
QT_USE_QTXML
QT_USE_QTSVG
QT_USE_QTTEST
QT_USE_QTUITOOLS
QT_USE_QTDBUS
QT_USE_QTSCRIPT
QT_USE_QTASSISTANTCLIENT
QT_USE_QTHELP
QT_USE_QTWEBKIT
QT_USE_QTXMLPATTERNS
QT_USE_PHONON
QT_USE_QTSCRIPTTOOLS
QT_USE_QTDECLARATIVE
```

```
QT_USE_IMPORTED_TARGETS
      If this variable is set to TRUE, FindQt4.cmake will create imported
      library targets for the various Qt libraries and set the
      library variables like QT_QTCORE_LIBRARY to point at these imported
      targets instead of the library file on disk. This provides much better
      handling of the release and debug versions of the Qt libraries and is
     also always backwards compatible, except for the case that dependencies
     of libraries are exported, these will then also list the names of the
     imported targets as dependency and not the file location on disk. This
     is much more flexible, but requires that FindQt4.cmake is executed before
     such an exported dependency file is processed.


     Note that if using IMPORTED targets, the qtmain.lib static library is
     automatically linked on Windows. To disable that globally, set the
```

```
      QT4_NO_LINK_QTMAIN variable before finding Qt4. To disable that for a
      particular executable, set the QT4_NO_LINK_QTMAIN target property to
      True on the executable.


  QT_INCLUDE_DIRS_NO_SYSTEM
        If this variable is set to TRUE, the Qt include directories
        in the QT_USE_FILE will NOT have the SYSTEM keyword set.
```

There are also some files that need processing by some Qt tools such as moc and uic. Listed below are macros that may be used to process those files.

```
  macro QT4_WRAP_CPP(outfiles inputfile ... OPTIONS ...)
        create moc code from a list of files containing Qt class with
        the Q_OBJECT declaration.  Per-directory preprocessor definitions
        are also added.  Options may be given to moc, such as those found
        when executing "moc -help".


  macro QT4_WRAP_UI(outfiles inputfile ... OPTIONS ...)
        create code from a list of Qt designer ui files.
        Options may be given to uic, such as those found
        when executing "uic -help"


  macro QT4_ADD_RESOURCES(outfiles inputfile ... OPTIONS ...)
        create code from a list of Qt resource files.
        Options may be given to rcc, such as those found
        when executing "rcc -help"


  macro QT4_GENERATE_MOC(inputfile outputfile )
        creates a rule to run moc on infile and create outfile.
        Use this if for some reason QT4_WRAP_CPP() isn't appropriate, e.g.
        because you need a custom filename for the moc file or something similar.


  macro QT4_AUTOMOC(sourcefile1 sourcefile2 ... )
        The qt4_automoc macro is obsolete.  Use the CMAKE_AUTOMOC feature instead.
        This macro is still experimental.
        It can be used to have moc automatically handled.
        So if you have the files foo.h and foo.cpp, and in foo.h a
        a class uses the Q_OBJECT macro, moc has to run on it. If you don't
        want to use QT4_WRAP_CPP() (which is reliable and mature), you can insert
        #include "foo.moc"
        in foo.cpp and then give foo.cpp as argument to QT4_AUTOMOC(). This will the
        scan all listed files at cmake-time for such included moc files and if it finds
        them cause a rule to be generated to run moc at build time on the
        accompanying header file foo.h.
        If a source file has the SKIP_AUTOMOC property set it will be ignored by this macro.


        You should have a look on the AUTOMOC property for targets to achieve the same results.


  macro QT4_ADD_DBUS_INTERFACE(outfiles interface basename)
        Create the interface header and implementation files with the
        given basename from the given interface xml file and add it to
        the list of sources.


        You can pass additional parameters to the qdbusxml2cpp call by setting
        properties on the input file:


        INCLUDE the given file will be included in the generate interface header


        CLASSNAME the generated class is named accordingly
```

```
      NO_NAMESPACE the generated class is not wrapped in a namespace


macro QT4_ADD_DBUS_INTERFACES(outfiles inputfile ... )
      Create the interface header and implementation files
      for all listed interface xml files.
      The basename will be automatically determined from the name of the xml file.

      The source file properties described for QT4_ADD_DBUS_INTERFACE also apply here.


macro QT4_ADD_DBUS_ADAPTOR(outfiles xmlfile parentheader parentclassname [basename] [classname])
      create a dbus adaptor (header and implementation file) from the xml file
      describing the interface, and add it to the list of sources. The adaptor
      forwards the calls to a parent class, defined in parentheader and named
      parentclassname. The name of the generated files will be
      <basename>adaptor.{cpp,h} where basename defaults to the basename of the xml file.
      If <classname> is provided, then it will be used as the classname of the
      adaptor itself.


macro QT4_GENERATE_DBUS_INTERFACE( header [interfacename] OPTIONS ...)
      generate the xml interface file from the given header.
      If the optional argument interfacename is omitted, the name of the
      interface file is constructed from the basename of the header with
      the suffix .xml appended.
      Options may be given to qdbuscpp2xml, such as those found when executing "qdbuscpp2xml --help"


macro QT4_CREATE_TRANSLATION( qm_files directories ... sources ...
                              ts_files ... OPTIONS ...)
      out: qm_files
      in:  directories sources ts_files
      options: flags to pass to lupdate, such as -extensions to specify
      extensions for a directory scan.
      generates commands to create .ts (vie lupdate) and .qm
      (via lrelease) - files from directories and/or sources. The ts files are
      created and/or updated in the source tree (unless given with full paths).
      The qm files are generated in the build tree.
      Updating the translations can be done by adding the qm_files
      to the source list of your library/executable, so they are
      always updated, or by adding a custom target to control when
      they get updated/generated.


macro QT4_ADD_TRANSLATION( qm_files ts_files ... )
      out: qm_files
      in:  ts_files
      generates commands to create .qm from .ts - files. The generated
      filenames can be found in qm_files. The ts_files
      must exist and are not updated in any way.
```

function QT4_USE_MODULES(target [link_type] modules...)

```
      This function is obsolete. Use target_link_libraries with IMPORTED targets instead.
      Make <target> use the <modules> from Qt. Using a Qt module means
      to link to the library, add the relevant include directories for the module,
      and add the relevant compiler defines for using the module.
      Modules are roughly equivalent to components of Qt4, so usage would be
      something like:
       qt4_use_modules(myexe Core Gui Declarative)
      to use QtCore, QtGui and QtDeclarative. The optional <link_type> argument can
      be specified as either LINK_PUBLIC or LINK_PRIVATE to specify the same argument
      to the target_link_libraries call.
```

```
Below is a detailed list of variables that FindQt4.cmake sets.
QT_FOUND         If false, don't try to use Qt.
Qt4_FOUND        If false, don't try to use Qt 4.
QT4_FOUND        If false, don't try to use Qt 4. This variable is for compatibility only.


QT_VERSION_MAJOR The major version of Qt found.
QT_VERSION_MINOR The minor version of Qt found.
QT_VERSION_PATCH The patch version of Qt found.


QT_EDITION                Set to the edition of Qt (i.e. DesktopLight)
QT_EDITION_DESKTOPLIGHT  True if QT_EDITION == DesktopLight
QT_QTCORE_FOUND          True if QtCore was found.
QT_QTGUI_FOUND           True if QtGui was found.
QT_QT3SUPPORT_FOUND      True if Qt3Support was found.
QT_QTASSISTANT_FOUND     True if QtAssistant was found.
QT_QTASSISTANTCLIENT_FOUND  True if QtAssistantClient was found.
QT_QAXCONTAINER_FOUND    True if QAxContainer was found (Windows only).
QT_QAXSERVER_FOUND       True if QAxServer was found (Windows only).
QT_QTDBUS_FOUND          True if QtDBus was found.
QT_QTDESIGNER_FOUND      True if QtDesigner was found.
QT_QTDESIGNERCOMPONENTS  True if QtDesignerComponents was found.
QT_QTHELP_FOUND          True if QtHelp was found.
QT_QTMOTIF_FOUND         True if QtMotif was found.
QT_QTMULTIMEDIA_FOUND    True if QtMultimedia was found (since Qt 4.6.0).
QT_QTNETWORK_FOUND       True if QtNetwork was found.
QT_QTNSPLUGIN_FOUND      True if QtNsPlugin was found.
QT_QTOPENGL_FOUND        True if QtOpenGL was found.
QT_QTSQL_FOUND           True if QtSql was found.
QT_QTSVG_FOUND           True if QtSvg was found.
QT_QTSCRIPT_FOUND        True if QtScript was found.
QT_QTSCRIPTTOOLS_FOUND   True if QtScriptTools was found.
QT_QTTEST_FOUND          True if QtTest was found.
QT_QTUITOOLS_FOUND       True if QtUiTools was found.
QT_QTWEBKIT_FOUND        True if QtWebKit was found.
QT_QTXML_FOUND           True if QtXml was found.
QT_QTXMLPATTERNS_FOUND   True if QtXmlPatterns was found.
QT_PHONON_FOUND          True if phonon was found.
QT_QTDECLARATIVE_FOUND   True if QtDeclarative was found.


QT_MAC_USE_COCOA    For Mac OS X, its whether Cocoa or Carbon is used.
                    In general, this should not be used, but its useful
                    when having platform specific code.


QT_DEFINITIONS   Definitions to use when compiling code that uses Qt.
                 You do not need to use this if you include QT_USE_FILE.
                 The QT_USE_FILE will also define QT_DEBUG and QT_NO_DEBUG
                 to fit your current build type.  Those are not contained
                 in QT_DEFINITIONS.


QT_INCLUDES      List of paths to all include directories of
                 Qt4 QT_INCLUDE_DIR and QT_QTCORE_INCLUDE_DIR are
                 always in this variable even if NOTFOUND,
                 all other INCLUDE_DIRS are
                 only added if they are found.
                 You do not need to use this if you include QT_USE_FILE.



Include directories for the Qt modules are listed here.
You do not need to use these variables if you include QT_USE_FILE.


QT_INCLUDE_DIR              Path to "include" of Qt4
QT_QT3SUPPORT_INCLUDE_DIR   Path to "include/Qt3Support"
```

```
QT_QTASSISTANT_INCLUDE_DIR  Path to "include/QtAssistant"
QT_QTASSISTANTCLIENT_INCLUDE_DIR       Path to "include/QtAssistant"
QT_QAXCONTAINER_INCLUDE_DIR Path to "include/ActiveQt" (Windows only)
QT_QAXSERVER_INCLUDE_DIR    Path to "include/ActiveQt" (Windows only)
QT_QTCORE_INCLUDE_DIR       Path to "include/QtCore"
QT_QTDBUS_INCLUDE_DIR       Path to "include/QtDBus"
QT_QTDESIGNER_INCLUDE_DIR   Path to "include/QtDesigner"
QT_QTDESIGNERCOMPONENTS_INCLUDE_DIR   Path to "include/QtDesigner"
QT_QTGUI_INCLUDE_DIR        Path to "include/QtGui"
QT_QTHELP_INCLUDE_DIR       Path to "include/QtHelp"
QT_QTMOTIF_INCLUDE_DIR      Path to "include/QtMotif"
QT_QTMULTIMEDIA_INCLUDE_DIR Path to "include/QtMultimedia"
QT_QTNETWORK_INCLUDE_DIR    Path to "include/QtNetwork"
QT_QTNSPLUGIN_INCLUDE_DIR   Path to "include/QtNsPlugin"
QT_QTOPENGL_INCLUDE_DIR     Path to "include/QtOpenGL"
QT_QTSCRIPT_INCLUDE_DIR     Path to "include/QtScript"
QT_QTSQL_INCLUDE_DIR        Path to "include/QtSql"
QT_QTSVG_INCLUDE_DIR        Path to "include/QtSvg"
QT_QTTEST_INCLUDE_DIR       Path to "include/QtTest"
QT_QTWEBKIT_INCLUDE_DIR     Path to "include/QtWebKit"
QT_QTXML_INCLUDE_DIR        Path to "include/QtXml"
QT_QTXMLPATTERNS_INCLUDE_DIR  Path to "include/QtXmlPatterns"
QT_PHONON_INCLUDE_DIR       Path to "include/phonon"
QT_QTSCRIPTTOOLS_INCLUDE_DIR       Path to "include/QtScriptTools"
QT_QTDECLARATIVE_INCLUDE_DIR       Path to "include/QtDeclarative"


QT_BINARY_DIR               Path to "bin" of Qt4
QT_LIBRARY_DIR              Path to "lib" of Qt4
QT_PLUGINS_DIR              Path to "plugins" for Qt4
QT_TRANSLATIONS_DIR         Path to "translations" of Qt4
QT_IMPORTS_DIR              Path to "imports" of Qt4
QT_DOC_DIR                  Path to "doc" of Qt4
QT_MKSPECS_DIR              Path to "mkspecs" of Qt4
```

The Qt toolkit may contain both debug and release libraries. In that case, the following library variables will contain both. You do not need to use these variables if you include QT_USE_FILE, and use QT_LIBRARIES.

```
QT_QT3SUPPORT_LIBRARY            The Qt3Support library
QT_QTASSISTANT_LIBRARY           The QtAssistant library
QT_QTASSISTANTCLIENT_LIBRARY     The QtAssistantClient library
QT_QAXCONTAINER_LIBRARY           The QAxContainer library (Windows only)
QT_QAXSERVER_LIBRARY               The QAxServer library (Windows only)
QT_QTCORE_LIBRARY                The QtCore library
QT_QTDBUS_LIBRARY                The QtDBus library
QT_QTDESIGNER_LIBRARY            The QtDesigner library
QT_QTDESIGNERCOMPONENTS_LIBRARY  The QtDesignerComponents library
QT_QTGUI_LIBRARY                 The QtGui library
QT_QTHELP_LIBRARY                The QtHelp library
QT_QTMOTIF_LIBRARY               The QtMotif library
QT_QTMULTIMEDIA_LIBRARY          The QtMultimedia library
QT_QTNETWORK_LIBRARY             The QtNetwork library
QT_QTNSPLUGIN_LIBRARY            The QtNsPLugin library
QT_QTOPENGL_LIBRARY              The QtOpenGL library
QT_QTSCRIPT_LIBRARY              The QtScript library
QT_QTSQL_LIBRARY                 The QtSql library
QT_QTSVG_LIBRARY                 The QtSvg library
QT_QTTEST_LIBRARY                The QtTest library
QT_QTUITOOLS_LIBRARY             The QtUiTools library
QT_QTWEBKIT_LIBRARY              The QtWebKit library
QT_QTXML_LIBRARY                 The QtXml library
QT_QTXMLPATTERNS_LIBRARY         The QtXmlPatterns library
QT_QTMAIN_LIBRARY                The qtmain library for Windows
QT_PHONON_LIBRARY                The phonon library
QT_QTSCRIPTTOOLS_LIBRARY         The QtScriptTools library
```

The QtDeclarative library: QT_QTDECLARATIVE_LIBRARY

also defined, but NOT for general use are

```
QT_MOC_EXECUTABLE                   Where to find the moc tool.
QT_UIC_EXECUTABLE                   Where to find the uic tool.
QT_UIC3_EXECUTABLE                  Where to find the uic3 tool.
QT_RCC_EXECUTABLE                   Where to find the rcc tool
QT_DBUSCPP2XML_EXECUTABLE           Where to find the qdbuscpp2xml tool.
QT_DBUSXML2CPP_EXECUTABLE           Where to find the qdbusxml2cpp tool.
QT_LUPDATE_EXECUTABLE               Where to find the lupdate tool.
QT_LRELEASE_EXECUTABLE              Where to find the lrelease tool.
QT_QCOLLECTIONGENERATOR_EXECUTABLE  Where to find the qcollectiongenerator tool.
QT_DESIGNER_EXECUTABLE              Where to find the Qt designer tool.
QT_LINGUIST_EXECUTABLE              Where to find the Qt linguist tool.
```

These are around for backwards compatibility they will be set

```
QT_WRAP_CPP  Set true if QT_MOC_EXECUTABLE is found
QT_WRAP_UI   Set true if QT_UIC_EXECUTABLE is found
```

These variables do _NOT_ have any effect anymore (compared to FindQt.cmake)

```
QT_MT_REQUIRED         Qt4 is now always multithreaded
```

These variables are set to "" Because Qt structure changed (They make no sense in Qt4)

```
QT_QT_LIBRARY        Qt-Library is now split
```

- **`FindQuickTime`**:

  Locate QuickTime This module defines QUICKTIME_LIBRARY QUICKTIME_FOUND, if false, do not try to link to gdal QUICKTIME_INCLUDE_DIR, where to find the headers

  $QUICKTIME_DIR is an environment variable that would correspond to the ./configure --prefix=$QUICKTIME_DIR

  Created by Eric Wing.

- **`FindRTI`**: Try to find M&S HLA RTI libraries

  This module finds if any HLA RTI is installed and locates the standard RTI include files and libraries.

  RTI is a simulation infrastructure standardized by IEEE and SISO. It has a well defined C++ API that assures that simulation applications are independent on a particular RTI implementation.

  http://en.wikipedia.org/wiki/Run-Time_Infrastructure_`(simulation)`

  This code sets the following variables:

  ```
  RTI_INCLUDE_DIR = the directory where RTI includes file are found
  RTI_LIBRARIES = The libraries to link against to use RTI
  RTI_DEFINITIONS = -DRTI_USES_STD_FSTREAM
  RTI_FOUND = Set to FALSE if any HLA RTI was not found
  ```

  Report problems to <certi-devel@nongnu.org>

- **`FindRuby`**: Find Ruby

  This module finds if Ruby is installed and determines where the include files and libraries are. Ruby 1.8 and 1.9 are supported.

  The minimum required version of Ruby can be specified using the standard syntax, e.g. find_package(Ruby 1.8)

  It also determines what the name of the library is. This code sets the following variables:

  ```
  RUBY_EXECUTABLE   = full path to the ruby binary
  RUBY_INCLUDE_DIRS = include dirs to be used when using the ruby library
  RUBY_LIBRARY      = full path to the ruby library
  RUBY_VERSION      = the version of ruby which was found, e.g. "1.8.7"
  RUBY_FOUND        = set to true if ruby ws found successfully


  RUBY_INCLUDE_PATH = same as RUBY_INCLUDE_DIRS, only provided for compatibility reasons, don't use it
  ```

- **`FindSDL`**: Locate SDL library

This module defines

```
SDL_LIBRARY, the name of the library to link against
SDL_FOUND, if false, do not try to link to SDL
SDL_INCLUDE_DIR, where to find SDL.h
SDL_VERSION_STRING, human-readable string containing the version of SDL
```

This module responds to the the flag:

```
SDL_BUILDING_LIBRARY
  If this is defined, then no SDL_main will be linked in because
  only applications need main().
  Otherwise, it is assumed you are building an application and this
  module will attempt to locate and set the the proper link flags
  as part of the returned SDL_LIBRARY variable.
```

Don't forget to include SDLmain.h and SDLmain.m your project for the OS X framework based version. (Other versions link to -lSDLmain which this module will try to find on your behalf.) Also for OS X, this module will automatically add the -framework Cocoa on your behalf.

Additional Note: If you see an empty SDL_LIBRARY_TEMP in your configuration and no SDL_LIBRARY, it means CMake did not find your SDL library (SDL.dll, libsdl.so, SDL.framework, etc). Set SDL_LIBRARY_TEMP to point to your SDL library, and configure again. Similarly, if you see an empty SDLMAIN_LIBRARY, you should set this value as appropriate. These values are used to generate the final SDL_LIBRARY variable, but when these values are unset, SDL_LIBRARY does not get created.

$SDLDIR is an environment variable that would correspond to the ./configure --prefix=$SDLDIR used in building SDL. l.e.galup 9-20-02

Modified by Eric Wing. Added code to assist with automated building by using environmental variables and providing a more controlled/consistent search behavior. Added new modifications to recognize OS X frameworks and additional Unix paths (FreeBSD, etc). Also corrected the header search path to follow "proper" SDL guidelines. Added a search for SDLmain which is needed by some platforms. Added a search for threads which is needed by some platforms. Added needed compile switches for MinGW.

On OSX, this will prefer the Framework version (if found) over others. People will have to manually change the cache values of SDL_LIBRARY to override this selection or set the CMake environment CMAKE_INCLUDE_PATH to modify the search paths.

Note that the header path has changed from SDL/SDL.h to just SDL.h This needed to change because "proper" SDL convention is #include "SDL.h", not <SDL/SDL.h>. This is done for portability reasons because not all systems place things in SDL/ (see FreeBSD).

- **`FindSDL_image`**: Locate SDL_image library

  This module defines:

  ```
  SDL_IMAGE_LIBRARIES, the name of the library to link against
  SDL_IMAGE_INCLUDE_DIRS, where to find the headers
  SDL_IMAGE_FOUND, if false, do not try to link against
  SDL_IMAGE_VERSION_STRING - human-readable string containing the version of SDL_image
  ```

  For backward compatiblity the following variables are also set:

  ```
  SDLIMAGE_LIBRARY (same value as SDL_IMAGE_LIBRARIES)
  SDLIMAGE_INCLUDE_DIR (same value as SDL_IMAGE_INCLUDE_DIRS)
  SDLIMAGE_FOUND (same value as SDL_IMAGE_FOUND)
  ```

  $SDLDIR is an environment variable that would correspond to the ./configure --prefix=$SDLDIR used in building SDL.

  Created by Eric Wing. This was influenced by the FindSDL.cmake module, but with modifications to recognize OS X frameworks and additional Unix paths (FreeBSD, etc).

- **`FindSDL_mixer`**: Locate SDL_mixer library

  This module defines:

  ```
  SDL_MIXER_LIBRARIES, the name of the library to link against
  SDL_MIXER_INCLUDE_DIRS, where to find the headers
  SDL_MIXER_FOUND, if false, do not try to link against
  SDL_MIXER_VERSION_STRING - human-readable string containing the version of SDL_mixer
  ```

For backward compatiblity the following variables are also set:

```
SDLMIXER_LIBRARY (same value as SDL_MIXER_LIBRARIES)
SDLMIXER_INCLUDE_DIR (same value as SDL_MIXER_INCLUDE_DIRS)
SDLMIXER_FOUND (same value as SDL_MIXER_FOUND)
```

$SDLDIR is an environment variable that would correspond to the ./configure --prefix=$SDLDIR used in building SDL.

Created by Eric Wing. This was influenced by the FindSDL.cmake module, but with modifications to recognize OS X frameworks and additional Unix paths (FreeBSD, etc).

- **`FindSDL_net`**: Locate SDL_net library

  This module defines:

  ```
  SDL_NET_LIBRARIES, the name of the library to link against
  SDL_NET_INCLUDE_DIRS, where to find the headers
  SDL_NET_FOUND, if false, do not try to link against
  SDL_NET_VERSION_STRING - human-readable string containing the version of SDL_net
  ```

  For backward compatiblity the following variables are also set:

  ```
  SDLNET_LIBRARY (same value as SDL_NET_LIBRARIES)
  SDLNET_INCLUDE_DIR (same value as SDL_NET_INCLUDE_DIRS)
  SDLNET_FOUND (same value as SDL_NET_FOUND)
  ```

  $SDLDIR is an environment variable that would correspond to the ./configure --prefix=$SDLDIR used in building SDL.

  Created by Eric Wing. This was influenced by the FindSDL.cmake module, but with modifications to recognize OS X frameworks and additional Unix paths (FreeBSD, etc).

- **`FindSDL_sound`**: Locates the SDL_sound library

  This module depends on SDL being found and must be called AFTER FindSDL.cmake is called.

  This module defines

  ```
  SDL_SOUND_INCLUDE_DIR, where to find SDL_sound.h
  SDL_SOUND_FOUND, if false, do not try to link to SDL_sound
  SDL_SOUND_LIBRARIES, this contains the list of libraries that you need
    to link against. This is a read-only variable and is marked INTERNAL.
  SDL_SOUND_EXTRAS, this is an optional variable for you to add your own
    flags to SDL_SOUND_LIBRARIES. This is prepended to SDL_SOUND_LIBRARIES.
    This is available mostly for cases this module failed to anticipate for
    and you must add additional flags. This is marked as ADVANCED.
  SDL_SOUND_VERSION_STRING, human-readable string containing the version of SDL_sound
  ```

  This module also defines (but you shouldn't need to use directly)

  ```
  SDL_SOUND_LIBRARY, the name of just the SDL_sound library you would link
  against. Use SDL_SOUND_LIBRARIES for you link instructions and not this one.
  ```

  And might define the following as needed

  ```
  MIKMOD_LIBRARY
  MODPLUG_LIBRARY
  OGG_LIBRARY
  VORBIS_LIBRARY
  SMPEG_LIBRARY
  FLAC_LIBRARY
  SPEEX_LIBRARY
  ```

  Typically, you should not use these variables directly, and you should use SDL_SOUND_LIBRARIES which contains SDL_SOUND_LIBRARY and the other audio libraries (if needed) to successfully compile on your system.

  Created by Eric Wing. This module is a bit more complicated than the other FindSDL* family modules. The reason is that SDL_sound can be compiled in a large variety of different ways which are independent of platform. SDL_sound may dynamically link against other 3rd party libraries to get additional codec support, such as Ogg Vorbis, SMPEG, ModPlug, MikMod, FLAC, Speex, and potentially others. Under some circumstances which I don't fully understand, there seems to be a

requirement that dependent libraries of libraries you use must also be explicitly linked against in order to successfully compile. SDL_sound does not currently have any system in place to know how it was compiled. So this CMake module does the hard work in trying to discover which 3rd party libraries are required for building (if any). This module uses a brute force approach to create a test program that uses SDL_sound, and then tries to build it. If the build fails, it parses the error output for known symbol names to figure out which libraries are needed.

Responds to the $SDLDIR and $SDLSOUNDDIR environmental variable that would correspond to the ./configure --prefix=$SDLDIR used in building SDL.

On OSX, this will prefer the Framework version (if found) over others. People will have to manually change the cache values of SDL_LIBRARY to override this selectionor set the CMake environment CMAKE_INCLUDE_PATH to modify the search paths.

- **`FindSDL_ttf`**: Locate SDL_ttf library

  This module defines:

  ```
  SDL_TTF_LIBRARIES, the name of the library to link against
  SDL_TTF_INCLUDE_DIRS, where to find the headers
  SDL_TTF_FOUND, if false, do not try to link against
  SDL_TTF_VERSION_STRING - human-readable string containing the version of SDL_ttf
  ```

  For backward compatiblity the following variables are also set:

  ```
  SDLTTF_LIBRARY (same value as SDL_TTF_LIBRARIES)
  SDLTTF_INCLUDE_DIR (same value as SDL_TTF_INCLUDE_DIRS)
  SDLTTF_FOUND (same value as SDL_TTF_FOUND)
  ```

  $SDLDIR is an environment variable that would correspond to the ./configure --prefix=$SDLDIR used in building SDL.

  Created by Eric Wing. This was influenced by the FindSDL.cmake module, but with modifications to recognize OS X frameworks and additional Unix paths (FreeBSD, etc).

- **`FindSWIG`**: Find SWIG

  This module finds an installed SWIG. It sets the following variables:

  ```
  SWIG_FOUND - set to true if SWIG is found
  SWIG_DIR - the directory where swig is installed
  SWIG_EXECUTABLE - the path to the swig executable
  SWIG_VERSION   - the version number of the swig executable
  ```

  The minimum required version of SWIG can be specified using the standard syntax, e.g. find_package(SWIG 1.1)

  All information is collected from the SWIG_EXECUTABLE so the version to be found can be changed from the command line by means of setting SWIG_EXECUTABLE

- **`FindSelfPackers`**: Find upx

  This module looks for some executable packers (i.e. software that compress executables or shared libs into on-the-fly self-extracting executables or shared libs. Examples:

  `UPX:` http://wildsau.idv.uni-linz.ac.at/mfx/upx.html

- **`FindSquish`**: -- Typical Use

  This module can be used to find Squish. Currently Squish versions 3 and 4 are supported.

  ```
  SQUISH_FOUND                  If false, don't try to use Squish
  SQUISH_VERSION                The full version of Squish found
  SQUISH_VERSION_MAJOR          The major version of Squish found
  SQUISH_VERSION_MINOR          The minor version of Squish found
  SQUISH_VERSION_PATCH          The patch version of Squish found

  SQUISH_INSTALL_DIR            The Squish installation directory (containing bin, lib, etc)
  SQUISH_SERVER_EXECUTABLE      The squishserver executable
  SQUISH_CLIENT_EXECUTABLE      The squishrunner executable

  SQUISH_INSTALL_DIR_FOUND      Was the install directory found?
  SQUISH_SERVER_EXECUTABLE_FOUND  Was the server executable found?
  SQUISH_CLIENT_EXECUTABLE_FOUND  Was the client executable found?
  ```

It provides the function squish_v4_add_test() for adding a squish test to cmake using Squish 4.x:

```
squish_v4_add_test(cmakeTestName AUT targetName SUITE suiteName TEST squishTestName
                [SETTINGSGROUP group] [PRE_COMMAND command] [POST_COMMAND command] )
```

The arguments have the following meaning:

```
cmakeTestName: this will be used as the first argument for add_test()
AUT targetName: the name of the cmake target which will be used as AUT, i.e. the
                executable which will be tested.
SUITE suiteName: this is either the full path to the squish suite, or just the
                 last directory of the suite, i.e. the suite name. In this case
                 the CMakeLists.txt which calls squish_add_test() must be located
                 in the parent directory of the suite directory.
TEST squishTestName: the name of the squish test, i.e. the name of the subdirectory
                     of the test inside the suite directory.
SETTINGSGROUP group: if specified, the given settings group will be used for executing the test.
                     If not specified, the groupname will be "CTest_<username>"
PRE_COMMAND command:  if specified, the given command will be executed before starting the squish test.
POST_COMMAND command: same as PRE_COMMAND, but after the squish test has been executed.
```

```
enable_testing()
find_package(Squish 4.0)
if (SQUISH_FOUND)
   squish_v4_add_test(myTestName AUT myApp SUITE ${CMAKE_SOURCE_DIR}/tests/mySuite TEST someSquishTest SETTINGSGROUP myGroup )
endif ()
```

For users of Squish version 3.x the macro squish_v3_add_test() is provided:

```
squish_v3_add_test(testName applicationUnderTest testCase envVars testWrapper)
Use this macro to add a test using Squish 3.x.
```

```
enable_testing()
find_package(Squish)
if (SQUISH_FOUND)
  squish_v3_add_test(myTestName myApplication testCase envVars testWrapper)
endif ()
```

macro SQUISH_ADD_TEST(testName applicationUnderTest testCase envVars testWrapper)

```
 This is deprecated. Use SQUISH_V3_ADD_TEST() if you are using Squish 3.x instead.
```

- **`FindSubversion`**: Extract information from a subversion working copy

  The module defines the following variables:

```
  Subversion_SVN_EXECUTABLE - path to svn command line client
  Subversion_VERSION_SVN - version of svn command line client
  Subversion_FOUND - true if the command line client was found
  SUBVERSION_FOUND - same as Subversion_FOUND, set for compatiblity reasons
```

  The minimum required version of Subversion can be specified using the standard syntax, e.g. find_package(Subversion 1.4)

  If the command line client executable is found two macros are defined:

```
  Subversion_WC_INFO(<dir> <var-prefix>)
  Subversion_WC_LOG(<dir> <var-prefix>)
```

  Subversion_WC_INFO extracts information of a subversion working copy at a given location. This macro defines the following variables:

```
  <var-prefix>_WC_URL - url of the repository (at <dir>)
  <var-prefix>_WC_ROOT - root url of the repository
  <var-prefix>_WC_REVISION - current revision
  <var-prefix>_WC_LAST_CHANGED_AUTHOR - author of last commit
  <var-prefix>_WC_LAST_CHANGED_DATE - date of last commit
  <var-prefix>_WC_LAST_CHANGED_REV - revision of last commit
```

```
<var-prefix>_WC_INFO - output of command `svn info <dir>'
```

Subversion_WC_LOG retrieves the log message of the base revision of a subversion working copy at a given location. This macro defines the variable:

```
<var-prefix>_LAST_CHANGED_LOG - last log of base revision
```

Example usage:

```
find_package(Subversion)
if(SUBVERSION_FOUND)
  Subversion_WC_INFO(${PROJECT_SOURCE_DIR} Project)
  message("Current revision is ${Project_WC_REVISION}")
  Subversion_WC_LOG(${PROJECT_SOURCE_DIR} Project)
  message("Last changed log is ${Project_LAST_CHANGED_LOG}")
endif()
```

- **`FindTCL`**: TK_INTERNAL_PATH was removed.

  This module finds if Tcl is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

```
TCL_FOUND              = Tcl was found
TK_FOUND               = Tk was found
TCLTK_FOUND            = Tcl and Tk were found
TCL_LIBRARY            = path to Tcl library (tcl tcl80)
TCL_INCLUDE_PATH       = path to where tcl.h can be found
TCL_TCLSH              = path to tclsh binary (tcl tcl80)
TK_LIBRARY             = path to Tk library (tk tk80 etc)
TK_INCLUDE_PATH        = path to where tk.h can be found
TK_WISH                = full path to the wish executable
```

In an effort to remove some clutter and clear up some issues for people who are not necessarily Tcl/Tk gurus/developpers, some variables were moved or removed. Changes compared to CMake 2.4 are:

```
=> they were only useful for people writing Tcl/Tk extensions.
=> these libs are not packaged by default with Tcl/Tk distributions.
   Even when Tcl/Tk is built from source, several flavors of debug libs
   are created and there is no real reason to pick a single one
   specifically (say, amongst tcl84g, tcl84gs, or tcl84sgx).
   Let's leave that choice to the user by allowing him to assign
   TCL_LIBRARY to any Tcl library, debug or not.
=> this ended up being only a Win32 variable, and there is a lot of
   confusion regarding the location of this file in an installed Tcl/Tk
   tree anyway (see 8.5 for example). If you need the internal path at
   this point it is safer you ask directly where the *source* tree is
   and dig from there.
```

- **`FindTIFF`**: Find TIFF library

  Find the native TIFF includes and library This module defines

```
TIFF_INCLUDE_DIR, where to find tiff.h, etc.
TIFF_LIBRARIES, libraries to link against to use TIFF.
TIFF_FOUND, If false, do not try to use TIFF.
```

  also defined, but not for general use are

```
TIFF_LIBRARY, where to find the TIFF library.
```

- **`FindTclStub`**: TCL_STUB_LIBRARY_DEBUG and TK_STUB_LIBRARY_DEBUG were removed.

  This module finds Tcl stub libraries. It first finds Tcl include files and libraries by calling FindTCL.cmake. How to Use the Tcl Stubs Library:

  http://tcl.activestate.com/doc/howto/stubs.html

  Using Stub Libraries:

  http://safari.oreilly.com/0130385603/ch48lev1sec3

  This code sets the following variables:

```
TCL_STUB_LIBRARY       = path to Tcl stub library
TK_STUB_LIBRARY        = path to Tk stub library
TTK_STUB_LIBRARY       = path to ttk stub library
```

In an effort to remove some clutter and clear up some issues for people who are not necessarily Tcl/Tk gurus/developpers, some variables were moved or removed. Changes compared to CMake 2.4 are:

```
=> these libs are not packaged by default with Tcl/Tk distributions.
   Even when Tcl/Tk is built from source, several flavors of debug libs
   are created and there is no real reason to pick a single one
   specifically (say, amongst tclstub84g, tclstub84gs, or tclstub84sgx).
   Let's leave that choice to the user by allowing him to assign
   TCL_STUB_LIBRARY to any Tcl library, debug or not.
```

- **FindTclsh**: Find tclsh

  This module finds if TCL is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

  ```
  TCLSH_FOUND = TRUE if tclsh has been found
  TCL_TCLSH = the path to the tclsh executable
  ```

  In cygwin, look for the cygwin version first. Don't look for it later to avoid finding the cygwin version on a Win32 build.

- **FindThreads**: This module determines the thread library of the system.

  The following variables are set

  ```
  CMAKE_THREAD_LIBS_INIT     - the thread library
  CMAKE_USE_SPROC_INIT       - are we using sproc?
  CMAKE_USE_WIN32_THREADS_INIT - using WIN32 threads?
  CMAKE_USE_PTHREADS_INIT    - are we using pthreads
  CMAKE_HP_PTHREADS_INIT     - are we using hp pthreads
  ```

  For systems with multiple thread libraries, caller can set

  ```
  CMAKE_THREAD_PREFER_PTHREAD
  ```

- **FindUnixCommands**: Find unix commands from cygwin

  This module looks for some usual Unix commands.

- **FindVTK**: Find a VTK installation or build tree.

  The following variables are set if VTK is found. If VTK is not found, VTK_FOUND is set to false.

  ```
  VTK_FOUND         - Set to true when VTK is found.
  VTK_USE_FILE      - CMake file to use VTK.
  VTK_MAJOR_VERSION - The VTK major version number.
  VTK_MINOR_VERSION - The VTK minor version number
                      (odd non-release).
  VTK_BUILD_VERSION - The VTK patch level
                      (meaningless for odd minor).
  VTK_INCLUDE_DIRS  - Include directories for VTK
  VTK_LIBRARY_DIRS  - Link directories for VTK libraries
  VTK_KITS          - List of VTK kits, in CAPS
                      (COMMON,IO,) etc.
  VTK_LANGUAGES     - List of wrapped languages, in CAPS
                      (TCL, PYHTON,) etc.
  ```

  The following cache entries must be set by the user to locate VTK:

  ```
  VTK_DIR  - The directory containing VTKConfig.cmake.
             This is either the root of the build tree,
             or the lib/vtk directory.  This is the
             only cache entry.
  ```

  The following variables are set for backward compatibility and should not be used in new code:

  ```
  USE_VTK_FILE - The full path to the UseVTK.cmake file.
                 This is provided for backward
                 compatibility.  Use VTK_USE_FILE
                 instead.
  ```

- **FindWget**: Find wget

  This module looks for wget. This module defines the following values:

  ```
  WGET_EXECUTABLE: the full path to the wget tool.
  WGET_FOUND: True if wget has been found.
  ```

- **FindWish**: Find wish installation

  This module finds if TCL is installed and determines where the include files and libraries are. It also determines what the name

of the library is. This code sets the following variables:

```
TK_WISH = the path to the wish executable
```

if UNIX is defined, then it will look for the cygwin version first

- **`FindX11`**: Find X11 installation

  Try to find X11 on UNIX systems. The following values are defined

```
X11_FOUND        - True if X11 is available
X11_INCLUDE_DIR  - include directories to use X11
X11_LIBRARIES    - link against these to use X11
```

  and also the following more fine grained variables: Include paths: X11_ICE_INCLUDE_PATH, X11_ICE_LIB, X11_ICE_FOUND

```
X11_SM_INCLUDE_PATH,           X11_SM_LIB,         X11_SM_FOUND
X11_X11_INCLUDE_PATH,          X11_X11_LIB
X11_Xaccessrules_INCLUDE_PATH,                     X11_Xaccess_FOUND
X11_Xaccessstr_INCLUDE_PATH,                       X11_Xaccess_FOUND
X11_Xau_INCLUDE_PATH,          X11_Xau_LIB,        X11_Xau_FOUND
X11_Xcomposite_INCLUDE_PATH,   X11_Xcomposite_LIB, X11_Xcomposite_FOUND
X11_Xcursor_INCLUDE_PATH,      X11_Xcursor_LIB,    X11_Xcursor_FOUND
X11_Xdamage_INCLUDE_PATH,      X11_Xdamage_LIB,    X11_Xdamage_FOUND
X11_Xdmcp_INCLUDE_PATH,        X11_Xdmcp_LIB,      X11_Xdmcp_FOUND
                               X11_Xext_LIB,       X11_Xext_FOUND
X11_dpms_INCLUDE_PATH,         (in X11_Xext_LIB),  X11_dpms_FOUND
X11_XShm_INCLUDE_PATH,         (in X11_Xext_LIB),  X11_XShm_FOUND
X11_Xshape_INCLUDE_PATH,       (in X11_Xext_LIB),  X11_Xshape_FOUND
X11_xf86misc_INCLUDE_PATH,     X11_Xxf86misc_LIB,  X11_xf86misc_FOUND
X11_xf86vmode_INCLUDE_PATH,    X11_Xxf86vm_LIB     X11_xf86vmode_FOUND
X11_Xfixes_INCLUDE_PATH,       X11_Xfixes_LIB,     X11_Xfixes_FOUND
X11_Xft_INCLUDE_PATH,          X11_Xft_LIB,        X11_Xft_FOUND
X11_Xi_INCLUDE_PATH,           X11_Xi_LIB,         X11_Xi_FOUND
X11_Xinerama_INCLUDE_PATH,     X11_Xinerama_LIB,   X11_Xinerama_FOUND
X11_Xinput_INCLUDE_PATH,       X11_Xinput_LIB,     X11_Xinput_FOUND
X11_Xkb_INCLUDE_PATH,                              X11_Xkb_FOUND
X11_Xkblib_INCLUDE_PATH,                           X11_Xkb_FOUND
X11_Xkbfile_INCLUDE_PATH,      X11_Xkbfile_LIB,    X11_Xkbfile_FOUND
X11_Xmu_INCLUDE_PATH,          X11_Xmu_LIB,        X11_Xmu_FOUND
X11_Xpm_INCLUDE_PATH,          X11_Xpm_LIB,        X11_Xpm_FOUND
X11_XTest_INCLUDE_PATH,        X11_XTest_LIB,      X11_XTest_FOUND
X11_Xrandr_INCLUDE_PATH,       X11_Xrandr_LIB,     X11_Xrandr_FOUND
X11_Xrender_INCLUDE_PATH,      X11_Xrender_LIB,    X11_Xrender_FOUND
X11_Xscreensaver_INCLUDE_PATH, X11_Xscreensaver_LIB, X11_Xscreensaver_FOUND
X11_Xt_INCLUDE_PATH,           X11_Xt_LIB,         X11_Xt_FOUND
X11_Xutil_INCLUDE_PATH,                            X11_Xutil_FOUND
X11_Xv_INCLUDE_PATH,           X11_Xv_LIB,         X11_Xv_FOUND
X11_XSync_INCLUDE_PATH,        (in X11_Xext_LIB),  X11_XSync_FOUND
```

- **`FindXMLRPC`**: Find xmlrpc

  Find the native XMLRPC headers and libraries.

```
XMLRPC_INCLUDE_DIRS      - where to find xmlrpc.h, etc.
XMLRPC_LIBRARIES         - List of libraries when using xmlrpc.
XMLRPC_FOUND             - True if xmlrpc found.
```

  XMLRPC modules may be specified as components for this find module. Modules may be listed by running "xmlrpc-c-config".
  Modules include:

```
c++            C++ wrapper code
libwww-client  libwww-based client
cgi-server     CGI-based server
abyss-server   ABYSS-based server
```

  Typical usage:

```
find_package(XMLRPC REQUIRED libwww-client)
```

- **`FindZLIB`**: Find zlib

  Find the native ZLIB includes and library. Once done this will define

```
ZLIB_INCLUDE_DIRS   - where to find zlib.h, etc.
```

```
ZLIB_LIBRARIES      - List of libraries when using zlib.
ZLIB_FOUND          - True if zlib found.


ZLIB_VERSION_STRING - The version of zlib found (x.y.z)
ZLIB_VERSION_MAJOR  - The major version of zlib
ZLIB_VERSION_MINOR  - The minor version of zlib
ZLIB_VERSION_PATCH  - The patch version of zlib
ZLIB_VERSION_TWEAK  - The tweak version of zlib
```

The following variable are provided for backward compatibility

```
ZLIB_MAJOR_VERSION  - The major version of zlib
ZLIB_MINOR_VERSION  - The minor version of zlib
ZLIB_PATCH_VERSION  - The patch version of zlib
```

An includer may set ZLIB_ROOT to a zlib installation root to tell this module where to look.

- **`Findosg`**:

  NOTE: It is highly recommended that you use the new FindOpenSceneGraph.cmake introduced in CMake 2.6.3 and not use this Find module directly.

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osg This module defines

  OSG_FOUND - Was the Osg found? OSG_INCLUDE_DIR - Where to find the headers OSG_LIBRARIES - The libraries to link against for the OSG (use this)

  OSG_LIBRARY - The OSG library OSG_LIBRARY_DEBUG - The OSG debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

- **`FindosgAnimation`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgAnimation This module defines

  OSGANIMATION_FOUND - Was osgAnimation found? OSGANIMATION_INCLUDE_DIR - Where to find the headers OSGANIMATION_LIBRARIES - The libraries to link against for the OSG (use this)

  OSGANIMATION_LIBRARY - The OSG library OSGANIMATION_LIBRARY_DEBUG - The OSG debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

- **`FindosgDB`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgDB This module defines

  OSGDB_FOUND - Was osgDB found? OSGDB_INCLUDE_DIR - Where to find the headers OSGDB_LIBRARIES - The libraries to link against for the osgDB (use this)

  OSGDB_LIBRARY - The osgDB library OSGDB_LIBRARY_DEBUG - The osgDB debug library

$OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

Created by Eric Wing.

- **`FindosgFX`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgFX This module defines

  OSGFX_FOUND - Was osgFX found? OSGFX_INCLUDE_DIR - Where to find the headers OSGFX_LIBRARIES - The libraries to link against for the osgFX (use this)

  OSGFX_LIBRARY - The osgFX library OSGFX_LIBRARY_DEBUG - The osgFX debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

- **`FindosgGA`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgGA This module defines

  OSGGA_FOUND - Was osgGA found? OSGGA_INCLUDE_DIR - Where to find the headers OSGGA_LIBRARIES - The libraries to link against for the osgGA (use this)

  OSGGA_LIBRARY - The osgGA library OSGGA_LIBRARY_DEBUG - The osgGA debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

  Created by Eric Wing.

- **`FindosgIntrospection`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgINTROSPECTION This module defines

  OSGINTROSPECTION_FOUND - Was osgIntrospection found? OSGINTROSPECTION_INCLUDE_DIR - Where to find the headers OSGINTROSPECTION_LIBRARIES - The libraries to link for osgIntrospection (use this)

  OSGINTROSPECTION_LIBRARY - The osgIntrospection library OSGINTROSPECTION_LIBRARY_DEBUG - The osgIntrospection debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

  Created by Eric Wing.

- **`FindosgManipulator`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgManipulator This module defines

  OSGMANIPULATOR_FOUND - Was osgManipulator found? OSGMANIPULATOR_INCLUDE_DIR - Where to find the headers OSGMANIPULATOR_LIBRARIES - The libraries to link for osgManipulator (use this)

  OSGMANIPULATOR_LIBRARY - The osgManipulator library OSGMANIPULATOR_LIBRARY_DEBUG - The osgManipulator debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

Created by Eric Wing.

- **`FindosgParticle`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgParticle This module defines

  OSGPARTICLE_FOUND - Was osgParticle found? OSGPARTICLE_INCLUDE_DIR - Where to find the headers OSGPARTICLE_LIBRARIES - The libraries to link for osgParticle (use this)

  OSGPARTICLE_LIBRARY - The osgParticle library OSGPARTICLE_LIBRARY_DEBUG - The osgParticle debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

- **`FindosgPresentation`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgPresentation This module defines

  OSGPRESENTATION_FOUND - Was osgPresentation found? OSGPRESENTATION_INCLUDE_DIR - Where to find the headers OSGPRESENTATION_LIBRARIES - The libraries to link for osgPresentation (use this)

  OSGPRESENTATION_LIBRARY - The osgPresentation library OSGPRESENTATION_LIBRARY_DEBUG - The osgPresentation debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing. Modified to work with osgPresentation by Robert Osfield, January 2012.

- **`FindosgProducer`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgProducer This module defines

  OSGPRODUCER_FOUND - Was osgProducer found? OSGPRODUCER_INCLUDE_DIR - Where to find the headers OSGPRODUCER_LIBRARIES - The libraries to link for osgProducer (use this)

  OSGPRODUCER_LIBRARY - The osgProducer library OSGPRODUCER_LIBRARY_DEBUG - The osgProducer debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

- **`FindosgQt`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgQt This module defines

  OSGQT_FOUND - Was osgQt found? OSGQT_INCLUDE_DIR - Where to find the headers OSGQT_LIBRARIES - The libraries to link for osgQt (use this)

  OSGQT_LIBRARY - The osgQt library OSGQT_LIBRARY_DEBUG - The osgQt debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing. Modified to work with osgQt by Robert Osfield, January 2012.

- **`FindosgShadow`**:

This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

Locate osgShadow This module defines

OSGSHADOW_FOUND - Was osgShadow found? OSGSHADOW_INCLUDE_DIR - Where to find the headers OSGSHADOW_LIBRARIES - The libraries to link for osgShadow (use this)

OSGSHADOW_LIBRARY - The osgShadow library OSGSHADOW_LIBRARY_DEBUG - The osgShadow debug library

$OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

Created by Eric Wing.

- **`FindosgSim`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgSim This module defines

  OSGSIM_FOUND - Was osgSim found? OSGSIM_INCLUDE_DIR - Where to find the headers OSGSIM_LIBRARIES - The libraries to link for osgSim (use this)

  OSGSIM_LIBRARY - The osgSim library OSGSIM_LIBRARY_DEBUG - The osgSim debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

- **`FindosgTerrain`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgTerrain This module defines

  OSGTERRAIN_FOUND - Was osgTerrain found? OSGTERRAIN_INCLUDE_DIR - Where to find the headers OSGTERRAIN_LIBRARIES - The libraries to link for osgTerrain (use this)

  OSGTERRAIN_LIBRARY - The osgTerrain library OSGTERRAIN_LIBRARY_DEBUG - The osgTerrain debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

- **`FindosgText`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgText This module defines

  OSGTEXT_FOUND - Was osgText found? OSGTEXT_INCLUDE_DIR - Where to find the headers OSGTEXT_LIBRARIES - The libraries to link for osgText (use this)

  OSGTEXT_LIBRARY - The osgText library OSGTEXT_LIBRARY_DEBUG - The osgText debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

- **`FindosgUtil`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or

change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

Locate osgUtil This module defines

OSGUTIL_FOUND - Was osgUtil found? OSGUTIL_INCLUDE_DIR - Where to find the headers OSGUTIL_LIBRARIES - The libraries to link for osgUtil (use this)

OSGUTIL_LIBRARY - The osgUtil library OSGUTIL_LIBRARY_DEBUG - The osgUtil debug library

$OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

Created by Eric Wing.

- **`FindosgViewer`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgViewer This module defines

  OSGVIEWER_FOUND - Was osgViewer found? OSGVIEWER_INCLUDE_DIR - Where to find the headers OSGVIEWER_LIBRARIES - The libraries to link for osgViewer (use this)

  OSGVIEWER_LIBRARY - The osgViewer library OSGVIEWER_LIBRARY_DEBUG - The osgViewer debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

- **`FindosgVolume`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgVolume This module defines

  OSGVOLUME_FOUND - Was osgVolume found? OSGVOLUME_INCLUDE_DIR - Where to find the headers OSGVOLUME_LIBRARIES - The libraries to link for osgVolume (use this)

  OSGVOLUME_LIBRARY - The osgVolume library OSGVOLUME_LIBRARY_DEBUG - The osgVolume debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  Created by Eric Wing.

- **`FindosgWidget`**:

  This is part of the Findosg* suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default FindOpenGL.cmake module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the FindOpenSceneGraph.cmake instead of the Findosg*.cmake modules.

  Locate osgWidget This module defines

  OSGWIDGET_FOUND - Was osgWidget found? OSGWIDGET_INCLUDE_DIR - Where to find the headers OSGWIDGET_LIBRARIES - The libraries to link for osgWidget (use this)

  OSGWIDGET_LIBRARY - The osgWidget library OSGWIDGET_LIBRARY_DEBUG - The osgWidget debug library

  $OSGDIR is an environment variable that would correspond to the ./configure --prefix=$OSGDIR used in building osg.

  FindosgWidget.cmake tweaked from Findosg* suite as created by Eric Wing.

- **`Findosg_functions`**:

  This CMake file contains two macros to assist with searching for OSG libraries and nodekits. Please see FindOpenSceneGraph.cmake for full documentation.

- **`FindwxWidgets`**: Find a wxWidgets (a.k.a., wxWindows) installation.

  This module finds if wxWidgets is installed and selects a default configuration to use. wxWidgets is a modular library. To specify

the modules that you will use, you need to name them as components to the package:

find_package(wxWidgets COMPONENTS core base ...)

There are two search branches: a windows style and a unix style. For windows, the following variables are searched for and set to defaults in case of multiple choices. Change them if the defaults are not desired (i.e., these are the only variables you should change to select a configuration):

```
wxWidgets_ROOT_DIR      - Base wxWidgets directory
                          (e.g., C:/wxWidgets-2.6.3).
wxWidgets_LIB_DIR       - Path to wxWidgets libraries
                          (e.g., C:/wxWidgets-2.6.3/lib/vc_lib).
wxWidgets_CONFIGURATION - Configuration to use
                          (e.g., msw, mswd, mswu, mswunivud, etc.)
wxWidgets_EXCLUDE_COMMON_LIBRARIES
                        - Set to TRUE to exclude linking of
                          commonly required libs (e.g., png tiff
                          jpeg zlib regex expat).
```

For unix style it uses the wx-config utility. You can select between debug/release, unicode/ansi, universal/non-universal, and static/shared in the QtDialog or ccmake interfaces by turning ON/OFF the following variables:

```
wxWidgets_USE_DEBUG
wxWidgets_USE_UNICODE
wxWidgets_USE_UNIVERSAL
wxWidgets_USE_STATIC
```

There is also a wxWidgets_CONFIG_OPTIONS variable for all other options that need to be passed to the wx-config utility. For example, to use the base toolkit found in the /usr/local path, set the variable (before calling the FIND_PACKAGE command) as such:

```
set(wxWidgets_CONFIG_OPTIONS --toolkit=base --prefix=/usr)
```

The following are set after the configuration is done for both windows and unix style:

```
wxWidgets_FOUND            - Set to TRUE if wxWidgets was found.
wxWidgets_INCLUDE_DIRS     - Include directories for WIN32
                             i.e., where to find "wx/wx.h" and
                             "wx/setup.h"; possibly empty for unices.
wxWidgets_LIBRARIES        - Path to the wxWidgets libraries.
wxWidgets_LIBRARY_DIRS     - compile time link dirs, useful for
                             rpath on UNIX. Typically an empty string
                             in WIN32 environment.
wxWidgets_DEFINITIONS      - Contains defines required to compile/link
                             against WX, e.g. WXUSINGDLL
wxWidgets_DEFINITIONS_DEBUG- Contains defines required to compile/link
                             against WX debug builds, e.g. __WXDEBUG__
wxWidgets_CXX_FLAGS        - Include dirs and compiler flags for
                             unices, empty on WIN32. Essentially
                             "`wx-config --cxxflags`".
wxWidgets_USE_FILE         - Convenience include file.
```

Sample usage:

```
# Note that for MinGW users the order of libs is important!
find_package(wxWidgets COMPONENTS net gl core base)
if(wxWidgets_FOUND)
  include(${wxWidgets_USE_FILE})
  # and for each of your dependent executable/library targets:
  target_link_libraries(<YourTarget> ${wxWidgets_LIBRARIES})
endif()
```

If wxWidgets is required (i.e., not an optional part):

```
find_package(wxWidgets REQUIRED net gl core base)
include(${wxWidgets_USE_FILE})
# and for each of your dependent executable/library targets:
target_link_libraries(<YourTarget> ${wxWidgets_LIBRARIES})
```

- **`FindwxWindows`**: Find wxWindows (wxWidgets) installation

  This module finds if wxWindows/wxWidgets is installed and determines where the include files and libraries are. It also determines what the name of the library is. Please note this file is DEPRECATED and replaced by FindwxWidgets.cmake. This code sets the following variables:

  ```
  WXWINDOWS_FOUND     = system has WxWindows
  WXWINDOWS_LIBRARIES = path to the wxWindows libraries
                        on Unix/Linux with additional
                        linker flags from
                        "wx-config --libs"
  CMAKE_WXWINDOWS_CXX_FLAGS  = Compiler flags for wxWindows,
                               essentially "`wx-config --cxxflags`"
                               on Linux
  WXWINDOWS_INCLUDE_DIR      = where to find "wx/wx.h" and "wx/setup.h"
  WXWINDOWS_LINK_DIRECTORIES = link directories, useful for rpath on
                                Unix
  WXWINDOWS_DEFINITIONS      = extra defines
  ```

  OPTIONS If you need OpenGL support please

  ```
  set(WXWINDOWS_USE_GL 1)
  ```

  in your CMakeLists.txt *before* you include this file.

  ```
  HAVE_ISYSTEM      - true required to replace -I by -isystem on g++
  ```

  For convenience include Use_wxWindows.cmake in your project's CMakeLists.txt using include(${CMAKE_CURRENT_LIST_DIR}/Use_wxWindows.cmake).

  USAGE

  ```
  set(WXWINDOWS_USE_GL 1)
  find_package(wxWindows)
  ```

  NOTES wxWidgets 2.6.x is supported for monolithic builds e.g. compiled in wx/build/msw dir as:

  ```
  nmake -f makefile.vc BUILD=debug SHARED=0 USE_OPENGL=1 MONOLITHIC=1
  ```

  DEPRECATED

  ```
  CMAKE_WX_CAN_COMPILE
  WXWINDOWS_LIBRARY
  CMAKE_WX_CXX_FLAGS
  WXWINDOWS_INCLUDE_PATH
  ```

  AUTHOR Jan Woetzel <http://www.mip.informatik.uni-kiel.de/~jw> (07/2003-01/2006)

- **`FortranCInterface`**: Fortran/C Interface Detection

  This module automatically detects the API by which C and Fortran languages interact. Variables indicate if the mangling is found:

  ```
  FortranCInterface_GLOBAL_FOUND = Global subroutines and functions
  FortranCInterface_MODULE_FOUND = Module subroutines and functions
                                   (declared by "MODULE PROCEDURE")
  ```

  A function is provided to generate a C header file containing macros to mangle symbol names:

  ```
  FortranCInterface_HEADER(<file>
                           [MACRO_NAMESPACE <macro-ns>]
                           [SYMBOL_NAMESPACE <ns>]
                           [SYMBOLS [<module>:]<function> ...])
  ```

  It generates in <file> definitions of the following macros:

  ```
  #define FortranCInterface_GLOBAL (name,NAME) ...
  #define FortranCInterface_GLOBAL_(name,NAME) ...
  #define FortranCInterface_MODULE (mod,name, MOD,NAME) ...
  #define FortranCInterface_MODULE_(mod,name, MOD,NAME) ...
  ```

  These macros mangle four categories of Fortran symbols, respectively:

```
- Global symbols without '_': call mysub()
- Global symbols with '_'   : call my_sub()
- Module symbols without '_': use mymod; call mysub()
- Module symbols with '_'   : use mymod; call my_sub()
```

If mangling for a category is not known, its macro is left undefined. All macros require raw names in both lower case and upper case. The MACRO_NAMESPACE option replaces the default "FortranCInterface_" prefix with a given namespace "<macro-ns>".

The SYMBOLS option lists symbols to mangle automatically with C preprocessor definitions:

```
<function>          ==> #define <ns><function> ...
<module>:<function> ==> #define <ns><module>_<function> ...
```

If the mangling for some symbol is not known then no preprocessor definition is created, and a warning is displayed. The SYMBOL_NAMESPACE option prefixes all preprocessor definitions generated by the SYMBOLS option with a given namespace "<ns>".

Example usage:

```
include(FortranCInterface)
FortranCInterface_HEADER(FC.h MACRO_NAMESPACE "FC_")
```

This creates a "FC.h" header that defines mangling macros FC_GLOBAL(), FC_GLOBAL_(), FC_MODULE(), and FC_MODULE_().

Example usage:

```
include(FortranCInterface)
FortranCInterface_HEADER(FCMangle.h
                         MACRO_NAMESPACE "FC_"
                         SYMBOL_NAMESPACE "FC_"
                         SYMBOLS mysub mymod:my_sub)
```

This creates a "FCMangle.h" header that defines the same FC_*() mangling macros as the previous example plus preprocessor symbols FC_mysub and FC_mymod_my_sub.

Another function is provided to verify that the Fortran and C/C++ compilers work together:

```
FortranCInterface_VERIFY([CXX] [QUIET])
```

It tests whether a simple test executable using Fortran and C (and C++ when the CXX option is given) compiles and links successfully. The result is stored in the cache entry FortranCInterface_VERIFIED_C (or FortranCInterface_VERIFIED_CXX if CXX is given) as a boolean. If the check fails and QUIET is not given the function terminates with a FATAL_ERROR message describing the problem. The purpose of this check is to stop a build early for incompatible compiler combinations. The test is built in the Release configuration.

FortranCInterface is aware of possible GLOBAL and MODULE manglings for many Fortran compilers, but it also provides an interface to specify new possible manglings. Set the variables

```
FortranCInterface_GLOBAL_SYMBOLS
FortranCInterface_MODULE_SYMBOLS
```

before including FortranCInterface to specify manglings of the symbols "MySub", "My_Sub", "MyModule:MySub", and "My_Module:My_Sub". For example, the code:

```
set(FortranCInterface_GLOBAL_SYMBOLS mysub_ my_sub__ MYSUB_)
  #                                  ^^^^^  ^^^^^^   ^^^^^
set(FortranCInterface_MODULE_SYMBOLS
    __mymodule_MOD_mysub __my_module_MOD_my_sub)
  #   ^^^^^^^^     ^^^^^   ^^^^^^^^^     ^^^^^^
include(FortranCInterface)
```

tells FortranCInterface to try given GLOBAL and MODULE manglings. (The carets point at raw symbol names for clarity in this example but are not needed.)

- **`GNUInstallDirs`**: Define GNU standard installation directories

  Provides install directory variables as defined for GNU software:

  http://www.gnu.org/prep/standards/html_node/Directory-Variables.html

  Inclusion of this module defines the following variables:

  ```
  CMAKE_INSTALL_<dir>      - destination for files of a given type
  CMAKE_INSTALL_FULL_<dir> - corresponding absolute path
  ```

  where <dir> is one of:

  ```
  BINDIR           - user executables (bin)
  SBINDIR          - system admin executables (sbin)
  LIBEXECDIR       - program executables (libexec)
  SYSCONFDIR       - read-only single-machine data (etc)
  ```

```
SHAREDSTATEDIR   - modifiable architecture-independent data (com)
LOCALSTATEDIR    - modifiable single-machine data (var)
LIBDIR           - object code libraries (lib or lib64 or lib/<multiarch-tuple> on Debian)
INCLUDEDIR       - C header files (include)
OLDINCLUDEDIR    - C header files for non-gcc (/usr/include)
DATAROOTDIR      - read-only architecture-independent data root (share)
DATADIR          - read-only architecture-independent data (DATAROOTDIR)
INFODIR          - info documentation (DATAROOTDIR/info)
LOCALEDIR        - locale-dependent data (DATAROOTDIR/locale)
MANDIR           - man documentation (DATAROOTDIR/man)
DOCDIR           - documentation root (DATAROOTDIR/doc/PROJECT_NAME)
```

Each CMAKE_INSTALL_<dir> value may be passed to the DESTINATION options of install() commands for the corresponding file type. If the includer does not define a value the above-shown default will be used and the value will appear in the cache for editing by the user. Each CMAKE_INSTALL_FULL_<dir> value contains an absolute path constructed from the corresponding destination by prepending (if necessary) the value of CMAKE_INSTALL_PREFIX.

- **GenerateExportHeader**: Function for generation of export macros for libraries

  This module provides the function GENERATE_EXPORT_HEADER() and the accompanying ADD_COMPILER_EXPORT_FLAGS() function.

  The GENERATE_EXPORT_HEADER function can be used to generate a file suitable for preprocessor inclusion which contains EXPORT macros to be used in library classes.

  GENERATE_EXPORT_HEADER(LIBRARY_TARGET

  ```
              [BASE_NAME <base_name>]
              [EXPORT_MACRO_NAME <export_macro_name>]
              [EXPORT_FILE_NAME <export_file_name>]
              [DEPRECATED_MACRO_NAME <deprecated_macro_name>]
              [NO_EXPORT_MACRO_NAME <no_export_macro_name>]
              [STATIC_DEFINE <static_define>]
              [NO_DEPRECATED_MACRO_NAME <no_deprecated_macro_name>]
              [DEFINE_NO_DEPRECATED]
              [PREFIX_NAME <prefix_name>]
  ```

)

  ADD_COMPILER_EXPORT_FLAGS([<output_variable>])

  By default GENERATE_EXPORT_HEADER() generates macro names in a file name determined by the name of the library. The ADD_COMPILER_EXPORT_FLAGS function adds -fvisibility=hidden to CMAKE_CXX_FLAGS if supported, and is a no-op on Windows which does not need extra compiler flags for exporting support. You may optionally pass a single argument to ADD_COMPILER_EXPORT_FLAGS that will be populated with the required CXX_FLAGS required to enable visibility support for the compiler/architecture in use.

  This means that in the simplest case, users of these functions will be equivalent to:

  ```
  add_compiler_export_flags()
  add_library(somelib someclass.cpp)
  generate_export_header(somelib)
  install(TARGETS somelib DESTINATION ${LIBRARY_INSTALL_DIR})
  install(FILES
   someclass.h
   ${PROJECT_BINARY_DIR}/somelib_export.h DESTINATION ${INCLUDE_INSTALL_DIR}
  )
  ```

  And in the ABI header files:

  ```
  #include "somelib_export.h"
  class SOMELIB_EXPORT SomeClass {
    ...
  };
  ```

  The CMake fragment will generate a file in the ${CMAKE_CURRENT_BINARY_DIR} called somelib_export.h containing the macros SOMELIB_EXPORT, SOMELIB_NO_EXPORT, SOMELIB_DEPRECATED, SOMELIB_DEPRECATED_EXPORT and SOMELIB_DEPRECATED_NO_EXPORT. The resulting file should be installed with other headers in the library.

  The BASE_NAME argument can be used to override the file name and the names used for the macros

  ```
  add_library(somelib someclass.cpp)
  generate_export_header(somelib
    BASE_NAME other_name
  )
  ```

Generates a file called other_name_export.h containing the macros OTHER_NAME_EXPORT, OTHER_NAME_NO_EXPORT and OTHER_NAME_DEPRECATED etc.

The BASE_NAME may be overridden by specifiying other options in the function. For example:

```
add_library(somelib someclass.cpp)
generate_export_header(somelib
  EXPORT_MACRO_NAME OTHER_NAME_EXPORT
)
```

creates the macro OTHER_NAME_EXPORT instead of SOMELIB_EXPORT, but other macros and the generated file name is as default.

```
add_library(somelib someclass.cpp)
generate_export_header(somelib
  DEPRECATED_MACRO_NAME KDE_DEPRECATED
)
```

creates the macro KDE_DEPRECATED instead of SOMELIB_DEPRECATED.

If LIBRARY_TARGET is a static library, macros are defined without values.

If the same sources are used to create both a shared and a static library, the uppercased symbol ${BASE_NAME}_STATIC_DEFINE should be used when building the static library

```
add_library(shared_variant SHARED ${lib_SRCS})
add_library(static_variant ${lib_SRCS})
generate_export_header(shared_variant BASE_NAME libshared_and_static)
set_target_properties(static_variant PROPERTIES
  COMPILE_FLAGS -DLIBSHARED_AND_STATIC_STATIC_DEFINE)
```

This will cause the export macros to expand to nothing when building the static library.

If DEFINE_NO_DEPRECATED is specified, then a macro ${BASE_NAME}_NO_DEPRECATED will be defined This macro can be used to remove deprecated code from preprocessor output.

```
option(EXCLUDE_DEPRECATED "Exclude deprecated parts of the library" FALSE)
if (EXCLUDE_DEPRECATED)
  set(NO_BUILD_DEPRECATED DEFINE_NO_DEPRECATED)
endif()
generate_export_header(somelib ${NO_BUILD_DEPRECATED})
```

And then in somelib:

```
class SOMELIB_EXPORT SomeClass
{
public:
#ifndef SOMELIB_NO_DEPRECATED
  SOMELIB_DEPRECATED void oldMethod();
#endif
};
```

```
#ifndef SOMELIB_NO_DEPRECATED
void SomeClass::oldMethod() {  }
#endif
```

If PREFIX_NAME is specified, the argument will be used as a prefix to all generated macros.

For example:

```
generate_export_header(somelib PREFIX_NAME VTK_)
```

Generates the macros VTK_SOMELIB_EXPORT etc.

- **GetPrerequisites**: Functions to analyze and list executable file prerequisites.

  This module provides functions to list the .dll, .dylib or .so files that an executable or shared library file depends on. (Its

prerequisites.)

It uses various tools to obtain the list of required shared library files:

```
dumpbin (Windows)
objdump (MinGW on Windows)
ldd (Linux/Unix)
otool (Mac OSX)
```

The following functions are provided by this module:

```
get_prerequisites
list_prerequisites
list_prerequisites_by_glob
gp_append_unique
is_file_executable
gp_item_default_embedded_path
  (projects can override with gp_item_default_embedded_path_override)
gp_resolve_item
  (projects can override with gp_resolve_item_override)
gp_resolved_file_type
  (projects can override with gp_resolved_file_type_override)
gp_file_type
```

Requires CMake 2.6 or greater because it uses function, break, return and PARENT_SCOPE.

```
GET_PREREQUISITES(<target> <prerequisites_var> <exclude_system> <recurse>
                  <exepath> <dirs>)
```

Get the list of shared library files required by <target>. The list in the variable named <prerequisites_var> should be empty on first entry to this function. On exit, <prerequisites_var> will contain the list of required shared library files.

<target> is the full path to an executable file. <prerequisites_var> is the name of a CMake variable to contain the results. <exclude_system> must be 0 or 1 indicating whether to include or exclude "system" prerequisites. If <recurse> is set to 1 all prerequisites will be found recursively, if set to 0 only direct prerequisites are listed. <exepath> is the path to the top level executable used for @executable_path replacment on the Mac. <dirs> is a list of paths where libraries might be found: these paths are searched first when a target without any path info is given. Then standard system locations are also searched: PATH, Framework locations, /usr/lib...

```
LIST_PREREQUISITES(<target> [<recurse> [<exclude_system> [<verbose>]]])
```

Print a message listing the prerequisites of <target>.

<target> is the name of a shared library or executable target or the full path to a shared library or executable file. If <recurse> is set to 1 all prerequisites will be found recursively, if set to 0 only direct prerequisites are listed. <exclude_system> must be 0 or 1 indicating whether to include or exclude "system" prerequisites. With <verbose> set to 0 only the full path names of the prerequisites are printed, set to 1 extra informatin will be displayed.

```
LIST_PREREQUISITES_BY_GLOB(<glob_arg> <glob_exp>)
```

Print the prerequisites of shared library and executable files matching a globbing pattern. <glob_arg> is GLOB or GLOB_RECURSE and <glob_exp> is a globbing expression used with "file(GLOB" or "file(GLOB_RECURSE" to retrieve a list of matching files. If a matching file is executable, its prerequisites are listed.

Any additional (optional) arguments provided are passed along as the optional arguments to the list_prerequisites calls.

```
GP_APPEND_UNIQUE(<list_var> <value>)
```

Append <value> to the list variable <list_var> only if the value is not already in the list.

```
IS_FILE_EXECUTABLE(<file> <result_var>)
```

Return 1 in <result_var> if <file> is a binary executable, 0 otherwise.

```
GP_ITEM_DEFAULT_EMBEDDED_PATH(<item> <default_embedded_path_var>)
```

Return the path that others should refer to the item by when the item is embedded inside a bundle.

Override on a per-project basis by providing a project-specific gp_item_default_embedded_path_override function.

```
GP_RESOLVE_ITEM(<context> <item> <exepath> <dirs> <resolved_item_var>)
```

Resolve an item into an existing full path file.

Override on a per-project basis by providing a project-specific gp_resolve_item_override function.

```
GP_RESOLVED_FILE_TYPE(<original_file> <file> <exepath> <dirs> <type_var>)
```

Return the type of <file> with respect to <original_file>. String describing type of prerequisite is returned in variable named <type_var>.

Use <exepath> and <dirs> if necessary to resolve non-absolute <file> values -- but only for non-embedded items.

Possible types are:

```
system
local
embedded
other
```

Override on a per-project basis by providing a project-specific gp_resolved_file_type_override function.

```
GP_FILE_TYPE(<original_file> <file> <type_var>)
```

Return the type of <file> with respect to <original_file>. String describing type of prerequisite is returned in variable named <type_var>.

Possible types are:

```
system
local
embedded
other
```

- **`InstallRequiredSystemLibraries`**:

  By including this file, all library files listed in the variable CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS will be installed with install(PROGRAMS ...) into bin for WIN32 and lib for non-WIN32. If CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP is set to TRUE before including this file, then the INSTALL command is not called. The user can use the variable CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS to use a custom install command and install them however they want. If it is the MSVC compiler, then the microsoft run time libraries will be found and automatically added to the CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS, and installed. If CMAKE_INSTALL_DEBUG_LIBRARIES is set and it is the MSVC compiler, then the debug libraries are installed when available. If CMAKE_INSTALL_DEBUG_LIBRARIES_ONLY is set then only the debug libraries are installed when both debug and release are available. If CMAKE_INSTALL_MFC_LIBRARIES is set then the MFC run time libraries are installed as well as the CRT run time libraries. If CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION is set then the libraries are installed to that directory rather than the default. If CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_NO_WARNINGS is NOT set, then this file warns about required files that do not exist. You can set this variable to ON before including this file to avoid the warning. For example, the Visual Studio Express editions do not include the redistributable files, so if you include this file on a machine with only VS Express installed, you'll get the warning.

- **`MacroAddFileDependencies`**: MACRO_ADD_FILE_DEPENDENCIES(<_file> depend_files...)

  Using the macro MACRO_ADD_FILE_DEPENDENCIES() is discouraged. There are usually better ways to specify the correct dependencies.

  MACRO_ADD_FILE_DEPENDENCIES(<_file> depend_files...) is just a convenience wrapper around the OBJECT_DEPENDS source file property. You can just use set_property(SOURCE <file> APPEND PROPERTY OBJECT_DEPENDS depend_files) instead.

- **`ProcessorCount`**: ProcessorCount(var)

  Determine the number of processors/cores and save value in ${var}

  Sets the variable named ${var} to the number of physical cores available on the machine if the information can be determined. Otherwise it is set to 0. Currently this functionality is implemented for AIX, cygwin, FreeBSD, HPUX, IRIX, Linux, Mac OS X, QNX, Sun and Windows.

  This function is guaranteed to return a positive integer (>=1) if it succeeds. It returns 0 if there's a problem determining the processor count.

  Example use, in a ctest -S dashboard script:

  ```
  include(ProcessorCount)
  ProcessorCount(N)
  if(NOT N EQUAL 0)
    set(CTEST_BUILD_FLAGS -j${N})
    set(ctest_test_args ${ctest_test_args} PARALLEL_LEVEL ${N})
  endif()
  ```

  This function is intended to offer an approximation of the value of the number of compute cores available on the current machine, such that you may use that value for parallel building and parallel testing. It is meant to help utilize as much of the machine as seems reasonable. Of course, knowledge of what else might be running on the machine simultaneously should be used when deciding whether to request a machine's full capacity all for yourself.

- **`Qt4ConfigDependentSettings`**:

  This file is included by FindQt4.cmake, don't include it directly.

- **`Qt4Macros`**:

  This file is included by FindQt4.cmake, don't include it directly.

- **SelectLibraryConfigurations**:

  select_library_configurations(basename)

  This macro takes a library base name as an argument, and will choose good values for basename_LIBRARY, basename_LIBRARIES, basename_LIBRARY_DEBUG, and basename_LIBRARY_RELEASE depending on what has been found and set. If only basename_LIBRARY_RELEASE is defined, basename_LIBRARY will be set to the release value, and basename_LIBRARY_DEBUG will be set to basename_LIBRARY_DEBUG-NOTFOUND. If only basename_LIBRARY_DEBUG is defined, then basename_LIBRARY will take the debug value, and basename_LIBRARY_RELEASE will be set to basename_LIBRARY_RELEASE-NOTFOUND.

  If the generator supports configuration types, then basename_LIBRARY and basename_LIBRARIES will be set with debug and optimized flags specifying the library to be used for the given configuration. If no build type has been set or the generator in use does not support configuration types, then basename_LIBRARY and basename_LIBRARIES will take only the release value, or the debug value if the release one is not set.

- **SquishTestScript**:

  This script launches a GUI test using Squish. You should not call the script directly; instead, you should access it via the SQUISH_ADD_TEST macro that is defined in FindSquish.cmake.

  This script starts the Squish server, launches the test on the client, and finally stops the squish server. If any of these steps fail (including if the tests do not pass) then a fatal error is raised.

- **TestBigEndian**: Define macro to determine endian type

  Check if the system is big endian or little endian

  ```
  TEST_BIG_ENDIAN(VARIABLE)
  VARIABLE - variable to store the result to
  ```

- **TestCXXAcceptsFlag**: Test CXX compiler for a flag

  Check if the CXX compiler accepts a flag

  ```
  Macro CHECK_CXX_ACCEPTS_FLAG(FLAGS VARIABLE) -
     checks if the function exists
  FLAGS - the flags to try
  VARIABLE - variable to store the result
  ```

- **TestForANSIForScope**: Check for ANSI for scope support

  Check if the compiler restricts the scope of variables declared in a for-init-statement to the loop body.

  ```
  CMAKE_NO_ANSI_FOR_SCOPE - holds result
  ```

- **TestForANSIStreamHeaders**: Test for compiler support of ANSI stream headers iostream, etc.

  check if the compiler supports the standard ANSI iostream header (without the .h)

  ```
  CMAKE_NO_ANSI_STREAM_HEADERS - defined by the results
  ```

- **TestForSSTREAM**: Test for compiler support of ANSI sstream header

  check if the compiler supports the standard ANSI sstream header

  ```
  CMAKE_NO_ANSI_STRING_STREAM - defined by the results
  ```

- **TestForSTDNamespace**: Test for std:: namespace support

  check if the compiler supports std:: on stl classes

  ```
  CMAKE_NO_STD_NAMESPACE - defined by the results
  ```

- **UseEcos**: This module defines variables and macros required to build eCos application.

  This file contains the following macros: ECOS_ADD_INCLUDE_DIRECTORIES() - add the eCos include dirs ECOS_ADD_EXECUTABLE(name source1 ... sourceN) - create an eCos executable ECOS_ADJUST_DIRECTORY(VAR source1 ... sourceN) - adjusts the path of the source files and puts the result into VAR

  Macros for selecting the toolchain: ECOS_USE_ARM_ELF_TOOLS() - enable the ARM ELF toolchain for the directory where it is called ECOS_USE_I386_ELF_TOOLS() - enable the i386 ELF toolchain for the directory where it is called

ECOS_USE_PPC_EABI_TOOLS() - enable the PowerPC toolchain for the directory where it is called

It contains the following variables: ECOS_DEFINITIONS ECOSCONFIG_EXECUTABLE ECOS_CONFIG_FILE - defaults to ecos.ecc, if your eCos configuration file has a different name, adjust this variable for internal use only:

```
ECOS_ADD_TARGET_LIB
```

- **`UseJava`**: Use Module for Java

  This file provides functions for Java. It is assumed that FindJava.cmake has already been loaded. See FindJava.cmake for information on how to load Java into your CMake project.

  add_jar(target_name

```
[SOURCES] source1 [source2 ...] [resource1 ...]
[INCLUDE_JARS jar1 [jar2 ...]]
[ENTRY_POINT entry]
[VERSION version]
[OUTPUT_NAME name]
[OUTPUT_DIR dir]
)
```

This command creates a <target_name>.jar. It compiles the given source files (source) and adds the given resource files (resource) to the jar file. If only resource files are given then just a jar file is created. The list of include jars are added to the classpath when compiling the java sources and also to the dependencies of the target. INCLUDE_JARS also accepts other target names created by add_jar. For backwards compatibility, jar files listed as sources are ignored (as they have been since the first version of this module).

The default OUTPUT_DIR can also be changed by setting the variable CMAKE_JAVA_TARGET_OUTPUT_DIR.

Additional instructions:

```
To add compile flags to the target you can set these flags with
the following variable:


    set(CMAKE_JAVA_COMPILE_FLAGS -nowarn)


To add a path or a jar file to the class path you can do this
with the CMAKE_JAVA_INCLUDE_PATH variable.


    set(CMAKE_JAVA_INCLUDE_PATH /usr/share/java/shibboleet.jar)


To use a different output name for the target you can set it with:


    add_jar(foobar foobar.java OUTPUT_NAME shibboleet.jar)


To use a different output directory than CMAKE_CURRENT_BINARY_DIR
you can set it with:


    add_jar(foobar foobar.java OUTPUT_DIR ${PROJECT_BINARY_DIR}/bin)


To define an entry point in your jar you can set it with the ENTRY_POINT
named argument:


    add_jar(example ENTRY_POINT com/examples/MyProject/Main)


To add a VERSION to the target output name you can set it using
the VERSION named argument to add_jar. This will create a jar file with the
name shibboleet-1.0.0.jar and will create a symlink shibboleet.jar
pointing to the jar with the version information.


    add_jar(shibboleet shibbotleet.java VERSION 1.2.0)
```

If the target is a JNI library, utilize the following commands to create a JNI symbolic link:

```
set(CMAKE_JNI_TARGET TRUE)
add_jar(shibboleet shibbotleet.java VERSION 1.2.0)
install_jar(shibboleet ${LIB_INSTALL_DIR}/shibboleet)
install_jni_symlink(shibboleet ${JAVA_LIB_INSTALL_DIR})
```

If a single target needs to produce more than one jar from its java source code, to prevent the accumulation of duplicate class files in subsequent jars, set/reset CMAKE_JAR_CLASSES_PREFIX prior to calling the add_jar() function:

```
set(CMAKE_JAR_CLASSES_PREFIX com/redhat/foo)
add_jar(foo foo.java)

set(CMAKE_JAR_CLASSES_PREFIX com/redhat/bar)
add_jar(bar bar.java)
```

Target Properties:

The add_jar() functions sets some target properties. You can get these properties with the

```
get_property(TARGET <target_name> PROPERTY <propery_name>)
```

command.

INSTALL_FILES	The files which should be installed. This is used by install_jar().
JNI_SYMLINK	The JNI symlink which should be installed. This is used by install_jni_symlink().
JAR_FILE	The location of the jar file so that you can include it.
CLASS_DIR	The directory where the class files can be found. For example to use them with javah.

find_jar(<VAR>

```
name | NAMES name1 [name2 ...]
[PATHS path1 [path2 ... ENV var]]
[VERSIONS version1 [version2]]
[DOC "cache documentation string"]
)
```

This command is used to find a full path to the named jar. A cache entry named by <VAR> is created to stor the result of this command. If the full path to a jar is found the result is stored in the variable and the search will not repeated unless the variable is cleared. If nothing is found, the result will be <VAR>-NOTFOUND, and the search will be attempted again next time find_jar 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 NAMES argument. Additional search locations can be specified after the PATHS argument. If you require special a version of a jar file you can specify it with the VERSIONS argument. The argument after DOC will be used for the documentation string in the cache.

install_jar(TARGET_NAME DESTINATION)

This command installs the TARGET_NAME files to the given DESTINATION. It should be called in the same scope as add_jar() or it will fail.

install_jni_symlink(TARGET_NAME DESTINATION)

This command installs the TARGET_NAME JNI symlinks to the given DESTINATION. It should be called in the same scope as add_jar() or it will fail.

create_javadoc(<VAR>

```
PACKAGES pkg1 [pkg2 ...]
```

```
                [SOURCEPATH <sourcepath>]
                [CLASSPATH <classpath>]
                [INSTALLPATH <install path>]
                [DOCTITLE "the documentation title"]
                [WINDOWTITLE "the title of the document"]
                [AUTHOR TRUE|FALSE]
                [USE TRUE|FALSE]
                [VERSION TRUE|FALSE]
               )
```

Create java documentation based on files or packages. For more details please read the javadoc manpage.

There are two main signatures for create_javadoc. The first signature works with package names on a path with source files:

```
Example:
create_javadoc(my_example_doc
  PACKAGES com.exmaple.foo com.example.bar
  SOURCEPATH "${CMAKE_CURRENT_SOURCE_DIR}"
  CLASSPATH ${CMAKE_JAVA_INCLUDE_PATH}
  WINDOWTITLE "My example"
  DOCTITLE "<h1>My example</h1>"
  AUTHOR TRUE
  USE TRUE
  VERSION TRUE
)
```

The second signature for create_javadoc works on a given list of files.

```
create_javadoc(<VAR>
               FILES file1 [file2 ...]
               [CLASSPATH <classpath>]
               [INSTALLPATH <install path>]
               [DOCTITLE "the documentation title"]
               [WINDOWTITLE "the title of the document"]
               [AUTHOR TRUE|FALSE]
               [USE TRUE|FALSE]
               [VERSION TRUE|FALSE]
              )
```

Example:

```
create_javadoc(my_example_doc
  FILES ${example_SRCS}
  CLASSPATH ${CMAKE_JAVA_INCLUDE_PATH}
  WINDOWTITLE "My example"
  DOCTITLE "<h1>My example</h1>"
  AUTHOR TRUE
  USE TRUE
  VERSION TRUE
)
```

Both signatures share most of the options. These options are the same as what you can find in the javadoc manpage. Please look at the manpage for CLASSPATH, DOCTITLE, WINDOWTITLE, AUTHOR, USE and VERSION.

The documentation will be by default installed to

```
${CMAKE_INSTALL_PREFIX}/share/javadoc/<VAR>
```

if you don't set the INSTALLPATH.

- **UseJavaClassFilelist**:

  This script create a list of compiled Java class files to be added to a jar file. This avoids including cmake files which get created in the binary directory.

- **UseJavaSymlinks**:

Helper script for UseJava.cmake

- **`UsePkgConfig`**: Obsolete pkg-config module for CMake, use FindPkgConfig instead.

  This module defines the following macro:

  PKGCONFIG(package includedir libdir linkflags cflags)

  Calling PKGCONFIG will fill the desired information into the 4 given arguments, e.g. PKGCONFIG(libart-2.0 LIBART_INCLUDE_DIR LIBART_LINK_DIR LIBART_LINK_FLAGS LIBART_CFLAGS) if pkg-config was NOT found or the specified software package doesn't exist, the variable will be empty when the function returns, otherwise they will contain the respective information

- **`UseQt4`**: Use Module for QT4

  Sets up C and C++ to use Qt 4. It is assumed that FindQt.cmake has already been loaded. See FindQt.cmake for information on how to load Qt 4 into your CMake project.

- **`UseSWIG`**: SWIG module for CMake

  Defines the following macros:

  ```
  SWIG_ADD_MODULE(name language [ files ])
    - Define swig module with given name and specified language
  SWIG_LINK_LIBRARIES(name [ libraries ])
    - Link libraries to swig module
  ```

  All other macros are for internal use only. To get the actual name of the swig module, use: ${SWIG_MODULE_${name}_REAL_NAME}. Set Source files properties such as CPLUSPLUS and SWIG_FLAGS to specify special behavior of SWIG. Also global CMAKE_SWIG_FLAGS can be used to add special flags to all swig calls. Another special variable is CMAKE_SWIG_OUTDIR, it allows one to specify where to write all the swig generated module (swig -outdir option) The name-specific variable SWIG_MODULE_<name>_EXTRA_DEPS may be used to specify extra dependencies for the generated modules. If the source file generated by swig need some special flag you can use set_source_files_properties(${swig_generated_file_fullname}

  ```
          PROPERTIES COMPILE_FLAGS "-bla")
  ```

- **`Use_wxWindows`**: --

  This convenience include finds if wxWindows is installed and set the appropriate libs, incdirs, flags etc. author Jan Woetzel <jw -at- mip.informatik.uni-kiel.de> (07/2003) USAGE:

  ```
  just include Use_wxWindows.cmake
  in your projects CMakeLists.txt
  ```

  include(${CMAKE_MODULE_PATH}/Use_wxWindows.cmake)

  ```
  if you are sure you need GL then
  ```

  set(WXWINDOWS_USE_GL 1)

  ```
  *before* you include this file.
  ```

- **`UsewxWidgets`**: Convenience include for using wxWidgets library.

  Determines if wxWidgets was FOUND and sets the appropriate libs, incdirs, flags, etc. INCLUDE_DIRECTORIES and LINK_DIRECTORIES are called.

  USAGE

  ```
  # Note that for MinGW users the order of libs is important!
  find_package(wxWidgets REQUIRED net gl core base)
  include(${wxWidgets_USE_FILE})
  # and for each of your dependent executable/library targets:
  target_link_libraries(<YourTarget> ${wxWidgets_LIBRARIES})
  ```

  DEPRECATED

  ```
  LINK_LIBRARIES is not called in favor of adding dependencies per target.
  ```

  AUTHOR

  ```
  Jan Woetzel <jw -at- mip.informatik.uni-kiel.de>
  ```

- **`WriteBasicConfigVersionFile`**:

  ```
  WRITE_BASIC_CONFIG_VERSION_FILE( filename VERSION major.minor.patch COMPATIBILITY (AnyNewerVersion|SameMajorVersion) )
  ```

Deprecated, see WRITE_BASIC_PACKAGE_VERSION_FILE(), it is identical.

Policies

- **CMP0000**: A minimum required CMake version must be specified.

  CMake requires that projects specify the version of CMake to which they have been written. This policy has been put in place so users trying to build the project may be told when they need to update their CMake. Specifying a version also helps the project build with CMake versions newer than that specified. Use the cmake_minimum_required command at the top of your main CMakeLists.txt file:

  ```
  cmake_minimum_required(VERSION <major>.<minor>)
  ```

  where "<major>.<minor>" is the version of CMake you want to support (such as "2.6"). The command will ensure that at least the given version of CMake is running and help newer versions be compatible with the project. See documentation of cmake_minimum_required for details.

  Note that the command invocation must appear in the CMakeLists.txt file itself; a call in an included file is not sufficient. However, the cmake_policy command may be called to set policy CMP0000 to OLD or NEW behavior explicitly. The OLD behavior is to silently ignore the missing invocation. The NEW behavior is to issue an error instead of a warning. An included file may set CMP0000 explicitly to affect how this policy is enforced for the main CMakeLists.txt file.

  This policy was introduced in CMake version 2.6.0.

- **CMP0001**: CMAKE_BACKWARDS_COMPATIBILITY should no longer be used.

  The OLD behavior is to check CMAKE_BACKWARDS_COMPATIBILITY and present it to the user. The NEW behavior is to ignore CMAKE_BACKWARDS_COMPATIBILITY completely.

  In CMake 2.4 and below the variable CMAKE_BACKWARDS_COMPATIBILITY was used to request compatibility with earlier versions of CMake. In CMake 2.6 and above all compatibility issues are handled by policies and the cmake_policy command. However, CMake must still check CMAKE_BACKWARDS_COMPATIBILITY for projects written for CMake 2.4 and below.

  This policy was introduced in CMake version 2.6.0. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0002**: Logical target names must be globally unique.

  Targets names created with add_executable, add_library, or add_custom_target are logical build target names. Logical target names must be globally unique because:

  - Unique names may be referenced unambiguously both in CMake
    code and on make tool command lines.
  - Logical names are used by Xcode and VS IDE generators
    to produce meaningful project names for the targets.

  The logical name of executable and library targets does not have to correspond to the physical file names built. Consider using the OUTPUT_NAME target property to create two targets with the same physical name while keeping logical names distinct. Custom targets must simply have globally unique names (unless one uses the global property ALLOW_DUPLICATE_CUSTOM_TARGETS with a Makefiles generator).

This policy was introduced in CMake version 2.6.0. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0003**: Libraries linked via full path no longer produce linker search paths.

  This policy affects how libraries whose full paths are NOT known are found at link time, but was created due to a change in how CMake deals with libraries whose full paths are known. Consider the code

  ```
  target_link_libraries(myexe /path/to/libA.so)
  ```

  CMake 2.4 and below implemented linking to libraries whose full paths are known by splitting them on the link line into separate components consisting of the linker search path and the library name. The example code might have produced something like

  ```
  ... -L/path/to -lA ...
  ```

  in order to link to library A. An analysis was performed to order multiple link directories such that the linker would find library A in the desired location, but there are cases in which this does not work. CMake versions 2.6 and above use the more reliable approach of passing the full path to libraries directly to the linker in most cases. The example code now produces something like

  ```
  ... /path/to/libA.so ....
  ```

  Unfortunately this change can break code like

  ```
  target_link_libraries(myexe /path/to/libA.so B)
  ```

  where "B" is meant to find "/path/to/libB.so". This code is wrong because the user is asking the linker to find library B but has not provided a linker search path (which may be added with the link_directories command). However, with the old linking implementation the code would work accidentally because the linker search path added for library A allowed library B to be found.

  In order to support projects depending on linker search paths added by linking to libraries with known full paths, the OLD behavior for this policy will add the linker search paths even though they are not needed for their own libraries. When this policy is set to OLD, CMake will produce a link line such as

  ```
  ... -L/path/to /path/to/libA.so -lB ...
  ```

  which will allow library B to be found as it was previously. When this policy is set to NEW, CMake will produce a link line such as

  ```
  ... /path/to/libA.so -lB ...
  ```

  which more accurately matches what the project specified.

  The setting for this policy used when generating the link line is that in effect when the target is created by an add_executable or add_library command. For the example described above, the code

  ```
  cmake_policy(SET CMP0003 OLD) # or cmake_policy(VERSION 2.4)
  add_executable(myexe myexe.c)
  target_link_libraries(myexe /path/to/libA.so B)
  ```

  will work and suppress the warning for this policy. It may also be updated to work with the corrected linking approach:

  ```
  cmake_policy(SET CMP0003 NEW) # or cmake_policy(VERSION 2.6)
  link_directories(/path/to) # needed to find library B
  add_executable(myexe myexe.c)
  target_link_libraries(myexe /path/to/libA.so B)
  ```

  Even better, library B may be specified with a full path:

  ```
  add_executable(myexe myexe.c)
  target_link_libraries(myexe /path/to/libA.so /path/to/libB.so)
  ```

  When all items on the link line have known paths CMake does not check this policy so it has no effect.

  Note that the warning for this policy will be issued for at most one target. This avoids flooding users with messages for every target when setting the policy once will probably fix all targets.

  This policy was introduced in CMake version 2.6.0. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0004**: Libraries linked may not have leading or trailing whitespace.

  CMake versions 2.4 and below silently removed leading and trailing whitespace from libraries linked with code like

  ```
  target_link_libraries(myexe " A ")
  ```

  This could lead to subtle errors in user projects.

  The OLD behavior for this policy is to silently remove leading and trailing whitespace. The NEW behavior for this policy is to diagnose the existence of such whitespace as an error. The setting for this policy used when checking the library names is that in effect when the target is created by an add_executable or add_library command.

This policy was introduced in CMake version 2.6.0. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0005**: Preprocessor definition values are now escaped automatically.

  This policy determines whether or not CMake should generate escaped preprocessor definition values added via add_definitions. CMake versions 2.4 and below assumed that only trivial values would be given for macros in add_definitions calls. It did not attempt to escape non-trivial values such as string literals in generated build rules. CMake versions 2.6 and above support escaping of most values, but cannot assume the user has not added escapes already in an attempt to work around limitations in earlier versions.

  The OLD behavior for this policy is to place definition values given to add_definitions directly in the generated build rules without attempting to escape anything. The NEW behavior for this policy is to generate correct escapes for all native build tools automatically. See documentation of the COMPILE_DEFINITIONS target property for limitations of the escaping implementation.

  This policy was introduced in CMake version 2.6.0. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0006**: Installing MACOSX_BUNDLE targets requires a BUNDLE DESTINATION.

  This policy determines whether the install(TARGETS) command must be given a BUNDLE DESTINATION when asked to install a target with the MACOSX_BUNDLE property set. CMake 2.4 and below did not distinguish application bundles from normal executables when installing targets. CMake 2.6 provides a BUNDLE option to the install(TARGETS) command that specifies rules specific to application bundles on the Mac. Projects should use this option when installing a target with the MACOSX_BUNDLE property set.

  The OLD behavior for this policy is to fall back to the RUNTIME DESTINATION if a BUNDLE DESTINATION is not given. The NEW behavior for this policy is to produce an error if a bundle target is installed without a BUNDLE DESTINATION.

  This policy was introduced in CMake version 2.6.0. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0007**: list command no longer ignores empty elements.

  This policy determines whether the list command will ignore empty elements in the list. CMake 2.4 and below list commands ignored all empty elements in the list. For example, a;b;;c would have length 3 and not 4. The OLD behavior for this policy is to ignore empty list elements. The NEW behavior for this policy is to correctly count empty elements in a list.

  This policy was introduced in CMake version 2.6.0. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0008**: Libraries linked by full-path must have a valid library file name.

  In CMake 2.4 and below it is possible to write code like

  ```
  target_link_libraries(myexe /full/path/to/somelib)
  ```

  where "somelib" is supposed to be a valid library file name such as "libsomelib.a" or "somelib.lib". For Makefile generators this produces an error at build time because the dependency on the full path cannot be found. For VS IDE and Xcode generators this used to work by accident because CMake would always split off the library directory and ask the linker to search for the library by name (-lsomelib or somelib.lib). Despite the failure with Makefiles, some projects have code like this and build only with VS and/or Xcode. This version of CMake prefers to pass the full path directly to the native build tool, which will fail in this case because it does not name a valid library file.

  This policy determines what to do with full paths that do not appear to name a valid library file. The OLD behavior for this policy is to split the library name from the path and ask the linker to search for it. The NEW behavior for this policy is to trust the given path and pass it directly to the native build tool unchanged.

  This policy was introduced in CMake version 2.6.1. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0009**: FILE GLOB_RECURSE calls should not follow symlinks by default.

  In CMake 2.6.1 and below, FILE GLOB_RECURSE calls would follow through symlinks, sometimes coming up with unexpectedly large result sets because of symlinks to top level directories that contain hundreds of thousands of files.

  This policy determines whether or not to follow symlinks encountered during a FILE GLOB_RECURSE call. The OLD behavior for this policy is to follow the symlinks. The NEW behavior for this policy is not to follow the symlinks by default, but only if FOLLOW_SYMLINKS is given as an additional argument to the FILE command.

  This policy was introduced in CMake version 2.6.2. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0010**: Bad variable reference syntax is an error.

  In CMake 2.6.2 and below, incorrect variable reference syntax such as a missing close-brace ("${FOO") was reported but did not stop processing of CMake code. This policy determines whether a bad variable reference is an error. The OLD behavior for this policy is to warn about the error, leave the string untouched, and continue. The NEW behavior for this policy is to report an error.

This policy was introduced in CMake version 2.6.3. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0011**: Included scripts do automatic cmake_policy PUSH and POP.

  In CMake 2.6.2 and below, CMake Policy settings in scripts loaded by the include() and find_package() commands would affect the includer. Explicit invocations of cmake_policy(PUSH) and cmake_policy(POP) were required to isolate policy changes and protect the includer. While some scripts intend to affect the policies of their includer, most do not. In CMake 2.6.3 and above, include() and find_package() by default PUSH and POP an entry on the policy stack around an included script, but provide a NO_POLICY_SCOPE option to disable it. This policy determines whether or not to imply NO_POLICY_SCOPE for compatibility. The OLD behavior for this policy is to imply NO_POLICY_SCOPE for include() and find_package() commands. The NEW behavior for this policy is to allow the commands to do their default cmake_policy PUSH and POP.

  This policy was introduced in CMake version 2.6.3. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0012**: if() recognizes numbers and boolean constants.

  In CMake versions 2.6.4 and lower the if() command implicitly dereferenced arguments corresponding to variables, even those named like numbers or boolean constants, except for 0 and 1. Numbers and boolean constants such as true, false, yes, no, on, off, y, n, notfound, ignore (all case insensitive) were recognized in some cases but not all. For example, the code "if(TRUE)" might have evaluated as false. Numbers such as 2 were recognized only in boolean expressions like "if(NOT 2)" (leading to false) but not as a single-argument like "if(2)" (also leading to false). Later versions of CMake prefer to treat numbers and boolean constants literally, so they should not be used as variable names.

  The OLD behavior for this policy is to implicitly dereference variables named like numbers and boolean constants. The NEW behavior for this policy is to recognize numbers and boolean constants without dereferencing variables with such names.

  This policy was introduced in CMake version 2.8.0. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0013**: Duplicate binary directories are not allowed.

  CMake 2.6.3 and below silently permitted add_subdirectory() calls to create the same binary directory multiple times. During build system generation files would be written and then overwritten in the build tree and could lead to strange behavior. CMake 2.6.4 and above explicitly detect duplicate binary directories. CMake 2.6.4 always considers this case an error. In CMake 2.8.0 and above this policy determines whether or not the case is an error. The OLD behavior for this policy is to allow duplicate binary directories. The NEW behavior for this policy is to disallow duplicate binary directories with an error.

  This policy was introduced in CMake version 2.8.0. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0014**: Input directories must have CMakeLists.txt.

  CMake versions before 2.8 silently ignored missing CMakeLists.txt files in directories referenced by add_subdirectory() or subdirs(), treating them as if present but empty. In CMake 2.8.0 and above this policy determines whether or not the case is an error. The OLD behavior for this policy is to silently ignore the problem. The NEW behavior for this policy is to report an error.

  This policy was introduced in CMake version 2.8.0. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0015**: link_directories() treats paths relative to the source dir.

  In CMake 2.8.0 and lower the link_directories() command passed relative paths unchanged to the linker. In CMake 2.8.1 and above the link_directories() command prefers to interpret relative paths with respect to CMAKE_CURRENT_SOURCE_DIR, which is consistent with include_directories() and other commands. The OLD behavior for this policy is to use relative paths verbatim in the linker command. The NEW behavior for this policy is to convert relative paths to absolute paths by appending the relative path to CMAKE_CURRENT_SOURCE_DIR.

  This policy was introduced in CMake version 2.8.1. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0016**: target_link_libraries() reports error if its only argument is not a target.

  In CMake 2.8.2 and lower the target_link_libraries() command silently ignored if it was called with only one argument, and this argument wasn't a valid target. In CMake 2.8.3 and above it reports an error in this case.

  This policy was introduced in CMake version 2.8.3. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0017**: Prefer files from the CMake module directory when including from there.

  Starting with CMake 2.8.4, if a cmake-module shipped with CMake (i.e. located in the CMake module directory) calls include() or find_package(), the files located in the CMake module directory are preferred over the files in CMAKE_MODULE_PATH. This makes sure that the modules belonging to CMake always get those files included which they expect, and against which they were developed and tested. In all other cases, the files found in CMAKE_MODULE_PATH still take precedence over the ones in the CMake module directory. The OLD behaviour is to always prefer files from CMAKE_MODULE_PATH over files from the CMake modules directory.

  This policy was introduced in CMake version 2.8.4. CMake version 2.8.12 warns when the policy is not set and uses OLD

behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0018**: Ignore CMAKE_SHARED_LIBRARY_<Lang>_FLAGS variable.

  CMake 2.8.8 and lower compiled sources in SHARED and MODULE libraries using the value of the undocumented CMAKE_SHARED_LIBRARY_<Lang>_FLAGS platform variable. The variable contained platform-specific flags needed to compile objects for shared libraries. Typically it included a flag such as -fPIC for position independent code but also included other flags needed on certain platforms. CMake 2.8.9 and higher prefer instead to use the POSITION_INDEPENDENT_CODE target property to determine what targets should be position independent, and new undocumented platform variables to select flags while ignoring CMAKE_SHARED_LIBRARY_<Lang>_FLAGS completely.

  The default for either approach produces identical compilation flags, but if a project modifies CMAKE_SHARED_LIBRARY_<Lang>_FLAGS from its original value this policy determines which approach to use.

  The OLD behavior for this policy is to ignore the POSITION_INDEPENDENT_CODE property for all targets and use the modified value of CMAKE_SHARED_LIBRARY_<Lang>_FLAGS for SHARED and MODULE libraries.

  The NEW behavior for this policy is to ignore CMAKE_SHARED_LIBRARY_<Lang>_FLAGS whether it is modified or not and honor the POSITION_INDEPENDENT_CODE target property.

  This policy was introduced in CMake version 2.8.9. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0019**: Do not re-expand variables in include and link information.

  CMake 2.8.10 and lower re-evaluated values given to the include_directories, link_directories, and link_libraries commands to expand any leftover variable references at the end of the configuration step. This was for strict compatibility with VERY early CMake versions because all variable references are now normally evaluated during CMake language processing. CMake 2.8.11 and higher prefer to skip the extra evaluation.

  The OLD behavior for this policy is to re-evaluate the values for strict compatibility. The NEW behavior for this policy is to leave the values untouched.

  This policy was introduced in CMake version 2.8.11. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0020**: Automatically link Qt executables to qtmain target on Windows.

  CMake 2.8.10 and lower required users of Qt to always specify a link dependency to the qtmain.lib static library manually on Windows. CMake 2.8.11 gained the ability to evaluate generator expressions while determining the link dependencies from IMPORTED targets. This allows CMake itself to automatically link executables which link to Qt to the qtmain.lib library when using IMPORTED Qt targets. For applications already linking to qtmain.lib, this should have little impact. For applications which supply their own alternative WinMain implementation and for applications which use the QAxServer library, this automatic linking will need to be disabled as per the documentation.

  The OLD behavior for this policy is not to link executables to qtmain.lib automatically when they link to the QtCore IMPORTEDtarget. The NEW behavior for this policy is to link executables to qtmain.lib automatically when they link to QtCore IMPORTED target.

  This policy was introduced in CMake version 2.8.11. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0021**: Fatal error on relative paths in INCLUDE_DIRECTORIES target property.

  CMake 2.8.10.2 and lower allowed the INCLUDE_DIRECTORIES target property to contain relative paths. The base path for such relative entries is not well defined. CMake 2.8.12 issues a FATAL_ERROR if the INCLUDE_DIRECTORIES property contains a relative path.

  The OLD behavior for this policy is not to warn about relative paths in the INCLUDE_DIRECTORIES target property. The NEW behavior for this policy is to issue a FATAL_ERROR if INCLUDE_DIRECTORIES contains a relative path.

  This policy was introduced in CMake version 2.8.12. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0022**: INTERFACE_LINK_LIBRARIES defines the link interface.

  CMake 2.8.11 constructed the 'link interface' of a target from properties matching (IMPORTED_)?LINK_INTERFACE_LIBRARIES(_<CONFIG>)?. The modern way to specify config-sensitive content is to use generator expressions and the IMPORTED_ prefix makes uniform processing of the link interface with generator expressions impossible. The INTERFACE_LINK_LIBRARIES target property was introduced as a replacement in CMake 2.8.12. This new property is named consistently with the INTERFACE_COMPILE_DEFINITIONS, INTERFACE_INCLUDE_DIRECTORIES and INTERFACE_COMPILE_OPTIONS properties. For in-build targets, CMake will use the INTERFACE_LINK_LIBRARIES property as the source of the link interface only if policy CMP0022 is NEW. When exporting a target which has this policy set to NEW, only the INTERFACE_LINK_LIBRARIES property will be processed and generated for the IMPORTED target by default. A new option to the install(EXPORT) and export commands allows export of the old-style properties for compatibility with downstream users of CMake versions older than 2.8.12. The target_link_libraries command will no longer populate the properties matching LINK_INTERFACE_LIBRARIES(_<CONFIG>)? if this policy is NEW.

  The OLD behavior for this policy is to ignore the INTERFACE_LINK_LIBRARIES property for in-build targets. The NEW behavior for this policy is to use the INTERFACE_LINK_LIBRARIES property for in-build targets, and ignore the old properties matching

(IMPORTED_)?LINK_INTERFACE_LIBRARIES(_<CONFIG>)?.

This policy was introduced in CMake version 2.8.12. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

- **CMP0023**: Plain and keyword target_link_libraries signatures cannot be mixed.

  CMake 2.8.12 introduced the target_link_libraries signature using the PUBLIC, PRIVATE, and INTERFACE keywords to generalize the LINK_PUBLIC and LINK_PRIVATE keywords introduced in CMake 2.8.7. Use of signatures with any of these keywords sets the link interface of a target explicitly, even if empty. This produces confusing behavior when used in combination with the historical behavior of the plain target_link_libraries signature. For example, consider the code:

  ```
  target_link_libraries(mylib A)
  target_link_libraries(mylib PRIVATE B)
  ```

  After the first line the link interface has not been set explicitly so CMake would use the link implementation, A, as the link interface. However, the second line sets the link interface to empty. In order to avoid this subtle behavior CMake now prefers to disallow mixing the plain and keyword signatures of target_link_libraries for a single target.

  The OLD behavior for this policy is to allow keyword and plain target_link_libraries signatures to be mixed. The NEW behavior for this policy is to not to allow mixing of the keyword and plain signatures.

  This policy was introduced in CMake version 2.8.12. CMake version 2.8.12 warns when the policy is not set and uses OLD behavior. Use the cmake_policy command to set it to OLD or NEW explicitly.

Variables

Variables That Change Behavior

- **BUILD_SHARED_LIBS**
- **CMAKE_ABSOLUTE_DESTINATION_FILES**
- **CMAKE_AUTOMOC_RELAXED_MODE**
- **CMAKE_BACKWARDS_COMPATIBILITY**
- **CMAKE_BUILD_TYPE**
- **CMAKE_COLOR_MAKEFILE**
- **CMAKE_CONFIGURATION_TYPES**
- **CMAKE_DEBUG_TARGET_PROPERTIES**
- **CMAKE_DISABLE_FIND_PACKAGE_<PackageName>**
- **CMAKE_ERROR_DEPRECATED**
- **CMAKE_ERROR_ON_ABSOLUTE_INSTALL_DESTINATION**
- **CMAKE_FIND_LIBRARY_PREFIXES**
- **CMAKE_FIND_LIBRARY_SUFFIXES**
- **CMAKE_FIND_PACKAGE_WARN_NO_MODULE**
- **CMAKE_IGNORE_PATH**
- **CMAKE_INCLUDE_PATH**
- **CMAKE_INSTALL_DEFAULT_COMPONENT_NAME**
- **CMAKE_INSTALL_PREFIX**
- **CMAKE_LIBRARY_PATH**
- **CMAKE_MFC_FLAG**
- **CMAKE_MODULE_PATH**
- **CMAKE_NOT_USING_CONFIG_FLAGS**
- **CMAKE_POLICY_DEFAULT_CMP<NNNN>**
- **CMAKE_PREFIX_PATH**
- **CMAKE_PROGRAM_PATH**
- **CMAKE_SKIP_INSTALL_ALL_DEPENDENCY**
- **CMAKE_SYSTEM_IGNORE_PATH**
- **CMAKE_SYSTEM_INCLUDE_PATH**
- **CMAKE_SYSTEM_LIBRARY_PATH**
- **CMAKE_SYSTEM_PREFIX_PATH**
- **CMAKE_SYSTEM_PROGRAM_PATH**
- **CMAKE_USER_MAKE_RULES_OVERRIDE**
- **CMAKE_WARN_DEPRECATED**
- **CMAKE_WARN_ON_ABSOLUTE_INSTALL_DESTINATION**

- **BUILD_SHARED_LIBS**: Global flag to cause add_library to create shared libraries if on.

  If present and true, this will cause all libraries to be built shared unless the library was explicitly added as a static library. This variable is often added to projects as an OPTION so that each user of a project can decide if they want to build the project using shared or static libraries.

- **CMAKE_ABSOLUTE_DESTINATION_FILES**: List of files which have been installed using an ABSOLUTE DESTINATION path.

  This variable is defined by CMake-generated cmake_install.cmake scripts. It can be used (read-only) by programs or scripts that source those install scripts. This is used by some CPack generators (e.g. RPM).

- **CMAKE_AUTOMOC_RELAXED_MODE**: Switch between strict and relaxed automoc mode.

By default, automoc behaves exactly as described in the documentation of the AUTOMOC target property. When set to TRUE, it accepts more input and tries to find the correct input file for moc even if it differs from the documented behaviour. In this mode it e.g. also checks whether a header file is intended to be processed by moc when a "foo.moc" file has been included.

Relaxed mode has to be enabled for KDE4 compatibility.

- **`CMAKE_BACKWARDS_COMPATIBILITY`**: Version of cmake required to build project

  From the point of view of backwards compatibility, this specifies what version of CMake should be supported. By default this value is the version number of CMake that you are running. You can set this to an older version of CMake to support deprecated commands of CMake in projects that were written to use older versions of CMake. This can be set by the user or set at the beginning of a CMakeLists file.

- **`CMAKE_BUILD_TYPE`**: Specifies the build type on single-configuration generators.

  This statically specifies what build type (configuration) will be built in this build tree. Possible values are empty, Debug, Release, RelWithDebInfo and MinSizeRel. This variable is only meaningful to single-configuration generators (such as make and Ninja) i.e. those which choose a single configuration when CMake runs to generate a build tree as opposed to multi-configuration generators which offer selection of the build configuration within the generated build environment. There are many per-config properties and variables (usually following clean SOME_VAR_<CONFIG> order conventions), such as CMAKE_C_FLAGS_<CONFIG>, specified as uppercase: CMAKE_C_FLAGS_[DEBUG|RELEASE|RELWITHDEBINFO|MINSIZEREL]. For example, in a build tree configured to build type Debug, CMake will see to having CMAKE_C_FLAGS_DEBUG settings get added to the CMAKE_C_FLAGS settings. See also CMAKE_CONFIGURATION_TYPES.

- **`CMAKE_COLOR_MAKEFILE`**: Enables color output when using the Makefile generator.

  When enabled, the generated Makefiles will produce colored output. Default is ON.

- **`CMAKE_CONFIGURATION_TYPES`**: Specifies the available build types on multi-config generators.

  This specifies what build types (configurations) will be available such as Debug, Release, RelWithDebInfo etc. This has reasonable defaults on most platforms, but can be extended to provide other build types. See also CMAKE_BUILD_TYPE for details of managing configuration data, and CMAKE_CFG_INTDIR.

- **`CMAKE_DEBUG_TARGET_PROPERTIES`**: Enables tracing output for target properties.

  This variable can be populated with a list of properties to generate debug output for when evaluating target properties. Currently it can only be used when evaluating the INCLUDE_DIRECTORIES, COMPILE_DEFINITIONS and COMPILE_OPTIONS target properties. In that case, it outputs a backtrace for each entry in the target property. Default is unset.

- **`CMAKE_DISABLE_FIND_PACKAGE_<PackageName>`**: Variable for disabling find_package() calls.

  Every non-REQUIRED find_package() call in a project can be disabled by setting the variable CMAKE_DISABLE_FIND_PACKAGE_<PackageName> to TRUE. This can be used to build a project without an optional package, although that package is installed.

  This switch should be used during the initial CMake run. Otherwise if the package has already been found in a previous CMake run, the variables which have been stored in the cache will still be there. In that case it is recommended to remove the cache variables for this package from the cache using the cache editor or cmake -U

- **`CMAKE_ERROR_DEPRECATED`**: Whether to issue deprecation errors for macros and functions.

  If TRUE, this can be used by macros and functions to issue fatal errors when deprecated macros or functions are used. This variable is FALSE by default.

- **`CMAKE_ERROR_ON_ABSOLUTE_INSTALL_DESTINATION`**: Ask cmake_install.cmake script to error out as soon as a file with absolute INSTALL DESTINATION is encountered.

  The fatal error is emitted before the installation of the offending file takes place. This variable is used by CMake-generated cmake_install.cmake scripts. If one sets this variable to ON while running the script, it may get fatal error messages from the script.

- **`CMAKE_FIND_LIBRARY_PREFIXES`**: Prefixes to prepend when looking for libraries.

  This specifies what prefixes to add to library names when the find_library command looks for libraries. On UNIX systems this is typically lib, meaning that when trying to find the foo library it will look for libfoo.

- **`CMAKE_FIND_LIBRARY_SUFFIXES`**: Suffixes to append when looking for libraries.

  This specifies what suffixes to add to library names when the find_library command looks for libraries. On Windows systems this is typically .lib and .dll, meaning that when trying to find the foo library it will look for foo.dll etc.

- **`CMAKE_FIND_PACKAGE_WARN_NO_MODULE`**: Tell find_package to warn if called without an explicit mode.

  If find_package is called without an explicit mode option (MODULE, CONFIG or NO_MODULE) and no Find<pkg>.cmake module is in CMAKE_MODULE_PATH then CMake implicitly assumes that the caller intends to search for a package configuration file. If no package configuration file is found then the wording of the failure message must account for both the case that the package is really missing and the case that the project has a bug and failed to provide the intended Find module. If instead the caller specifies an explicit mode option then the failure message can be more specific.

  Set CMAKE_FIND_PACKAGE_WARN_NO_MODULE to TRUE to tell find_package to warn when it implicitly assumes Config mode. This helps developers enforce use of an explicit mode in all calls to find_package within a project.

- **CMAKE_IGNORE_PATH**: Path to be ignored by FIND_XXX() commands.

  Specifies directories to be ignored by searches in FIND_XXX() commands. This is useful in cross-compiled environments where some system directories contain incompatible but possibly linkable libraries. For example, on cross-compiled cluster environments, this allows a user to ignore directories containing libraries meant for the front-end machine that modules like FindX11 (and others) would normally search. By default this is empty; it is intended to be set by the project. Note that CMAKE_IGNORE_PATH takes a list of directory names, NOT a list of prefixes. If you want to ignore paths under prefixes (bin, include, lib, etc.), you'll need to specify them explicitly. See also CMAKE_PREFIX_PATH, CMAKE_LIBRARY_PATH, CMAKE_INCLUDE_PATH, CMAKE_PROGRAM_PATH.

- **CMAKE_INCLUDE_PATH**: Path used for searching by FIND_FILE() and FIND_PATH().

  Specifies a path which will be used both by FIND_FILE() and FIND_PATH(). Both commands will check each of the contained directories for the existence of the file which is currently searched. By default it is empty, it is intended to be set by the project. See also CMAKE_SYSTEM_INCLUDE_PATH, CMAKE_PREFIX_PATH.

- **CMAKE_INSTALL_DEFAULT_COMPONENT_NAME**: Default component used in install() commands.

  If an install() command is used without the COMPONENT argument, these files will be grouped into a default component. The name of this default install component will be taken from this variable. It defaults to "Unspecified".

- **CMAKE_INSTALL_PREFIX**: Install directory used by install.

  If "make install" is invoked or INSTALL is built, this directory is prepended onto all install directories. This variable defaults to /usr/local on UNIX and c:/Program Files on Windows.

  On UNIX one can use the DESTDIR mechanism in order to relocate the whole installation. DESTDIR means DESTination DIRectory. It is commonly used by makefile users in order to install software at non-default location. It is usually invoked like this:

  ```
  make DESTDIR=/home/john install
  ```

  which will install the concerned software using the installation prefix, e.g. "/usr/local" prepended with the DESTDIR value which finally gives "/home/john/usr/local".

  WARNING: DESTDIR may not be used on Windows because installation prefix usually contains a drive letter like in "C:/Program Files" which cannot be prepended with some other prefix.

  The installation prefix is also added to CMAKE_SYSTEM_PREFIX_PATH so that find_package, find_program, find_library, find_path, and find_file will search the prefix for other software.

- **CMAKE_LIBRARY_PATH**: Path used for searching by FIND_LIBRARY().

  Specifies a path which will be used by FIND_LIBRARY(). FIND_LIBRARY() will check each of the contained directories for the existence of the library which is currently searched. By default it is empty, it is intended to be set by the project. See also CMAKE_SYSTEM_LIBRARY_PATH, CMAKE_PREFIX_PATH.

- **CMAKE_MFC_FLAG**: Tell cmake to use MFC for an executable or dll.

  This can be set in a CMakeLists.txt file and will enable MFC in the application. It should be set to 1 for the static MFC library, and 2 for the shared MFC library. This is used in Visual Studio 6 and 7 project files. The CMakeSetup dialog used MFC and the CMakeLists.txt looks like this:

  ```
  add_definitions(-D_AFXDLL)
  set(CMAKE_MFC_FLAG 2)
  add_executable(CMakeSetup WIN32 ${SRCS})
  ```

- **CMAKE_MODULE_PATH**: List of directories to search for CMake modules.

  Commands like include() and find_package() search for files in directories listed by this variable before checking the default modules that come with CMake.

- **CMAKE_NOT_USING_CONFIG_FLAGS**: Skip _BUILD_TYPE flags if true.

  This is an internal flag used by the generators in CMake to tell CMake to skip the _BUILD_TYPE flags.

- **CMAKE_POLICY_DEFAULT_CMP<NNNN>**: Default for CMake Policy CMP<NNNN> when it is otherwise left unset.

  Commands cmake_minimum_required(VERSION) and cmake_policy(VERSION) by default leave policies introduced after the given version unset. Set CMAKE_POLICY_DEFAULT_CMP<NNNN> to OLD or NEW to specify the default for policy CMP<NNNN>, where <NNNN> is the policy number.

  This variable should not be set by a project in CMake code; use cmake_policy(SET) instead. Users running CMake may set this variable in the cache (e.g. -DCMAKE_POLICY_DEFAULT_CMP<NNNN>=<OLD|NEW>) to set a policy not otherwise set by the project. Set to OLD to quiet a policy warning while using old behavior or to NEW to try building the project with new behavior.

- **CMAKE_PREFIX_PATH**: Path used for searching by FIND_XXX(), with appropriate suffixes added.

  Specifies a path which will be used by the FIND_XXX() commands. It contains the "base" directories, the FIND_XXX() commands append appropriate subdirectories to the base directories. So FIND_PROGRAM() adds /bin to each of the directories in the path, FIND_LIBRARY() appends /lib to each of the directories, and FIND_PATH() and FIND_FILE() append /include . By default it is empty, it is intended to be set by the project. See also CMAKE_SYSTEM_PREFIX_PATH, CMAKE_INCLUDE_PATH,

CMAKE_LIBRARY_PATH, CMAKE_PROGRAM_PATH.

- **CMAKE_PROGRAM_PATH**: Path used for searching by FIND_PROGRAM().

  Specifies a path which will be used by FIND_PROGRAM(). FIND_PROGRAM() will check each of the contained directories for the existence of the program which is currently searched. By default it is empty, it is intended to be set by the project. See also CMAKE_SYSTEM_PROGRAM_PATH, CMAKE_PREFIX_PATH.

- **CMAKE_SKIP_INSTALL_ALL_DEPENDENCY**: Don't make the install target depend on the all target.

  By default, the "install" target depends on the "all" target. This has the effect, that when "make install" is invoked or INSTALL is built, first the "all" target is built, then the installation starts. If CMAKE_SKIP_INSTALL_ALL_DEPENDENCY is set to TRUE, this dependency is not created, so the installation process will start immediately, independent from whether the project has been completely built or not.

- **CMAKE_SYSTEM_IGNORE_PATH**: Path to be ignored by FIND_XXX() commands.

  Specifies directories to be ignored by searches in FIND_XXX() commands. This is useful in cross-compiled environments where some system directories contain incompatible but possibly linkable libraries. For example, on cross-compiled cluster environments, this allows a user to ignore directories containing libraries meant for the front-end machine that modules like FindX11 (and others) would normally search. By default this contains a list of directories containing incompatible binaries for the host system. See also CMAKE_SYSTEM_PREFIX_PATH, CMAKE_SYSTEM_LIBRARY_PATH, CMAKE_SYSTEM_INCLUDE_PATH, and CMAKE_SYSTEM_PROGRAM_PATH.

- **CMAKE_SYSTEM_INCLUDE_PATH**: Path used for searching by FIND_FILE() and FIND_PATH().

  Specifies a path which will be used both by FIND_FILE() and FIND_PATH(). Both commands will check each of the contained directories for the existence of the file which is currently searched. By default it contains the standard directories for the current system. It is NOT intended to be modified by the project, use CMAKE_INCLUDE_PATH for this. See also CMAKE_SYSTEM_PREFIX_PATH.

- **CMAKE_SYSTEM_LIBRARY_PATH**: Path used for searching by FIND_LIBRARY().

  Specifies a path which will be used by FIND_LIBRARY(). FIND_LIBRARY() will check each of the contained directories for the existence of the library which is currently searched. By default it contains the standard directories for the current system. It is NOT intended to be modified by the project, use CMAKE_LIBRARY_PATH for this. See also CMAKE_SYSTEM_PREFIX_PATH.

- **CMAKE_SYSTEM_PREFIX_PATH**: Path used for searching by FIND_XXX(), with appropriate suffixes added.

  Specifies a path which will be used by the FIND_XXX() commands. It contains the "base" directories, the FIND_XXX() commands append appropriate subdirectories to the base directories. So FIND_PROGRAM() adds /bin to each of the directories in the path, FIND_LIBRARY() appends /lib to each of the directories, and FIND_PATH() and FIND_FILE() append /include . By default this contains the standard directories for the current system and the CMAKE_INSTALL_PREFIX. It is NOT intended to be modified by the project, use CMAKE_PREFIX_PATH for this. See also CMAKE_SYSTEM_INCLUDE_PATH, CMAKE_SYSTEM_LIBRARY_PATH, CMAKE_SYSTEM_PROGRAM_PATH, and CMAKE_SYSTEM_IGNORE_PATH.

- **CMAKE_SYSTEM_PROGRAM_PATH**: Path used for searching by FIND_PROGRAM().

  Specifies a path which will be used by FIND_PROGRAM(). FIND_PROGRAM() will check each of the contained directories for the existence of the program which is currently searched. By default it contains the standard directories for the current system. It is NOT intended to be modified by the project, use CMAKE_PROGRAM_PATH for this. See also CMAKE_SYSTEM_PREFIX_PATH.

- **CMAKE_USER_MAKE_RULES_OVERRIDE**: Specify a CMake file that overrides platform information.

  CMake loads the specified file while enabling support for each language from either the project() or enable_language() commands. It is loaded after CMake's builtin compiler and platform information modules have been loaded but before the information is used. The file may set platform information variables to override CMake's defaults.

  This feature is intended for use only in overriding information variables that must be set before CMake builds its first test project to check that the compiler for a language works. It should not be used to load a file in cases that a normal include() will work. Use it only as a last resort for behavior that cannot be achieved any other way. For example, one may set CMAKE_C_FLAGS_INIT to change the default value used to initialize CMAKE_C_FLAGS before it is cached. The override file should NOT be used to set anything that could be set after languages are enabled, such as variables like CMAKE_RUNTIME_OUTPUT_DIRECTORY that affect the placement of binaries. Information set in the file will be used for try_compile and try_run builds too.

- **CMAKE_WARN_DEPRECATED**: Whether to issue deprecation warnings for macros and functions.

  If TRUE, this can be used by macros and functions to issue deprecation warnings. This variable is FALSE by default.

- **CMAKE_WARN_ON_ABSOLUTE_INSTALL_DESTINATION**: Ask cmake_install.cmake script to warn each time a file with absolute INSTALL DESTINATION is encountered.

  This variable is used by CMake-generated cmake_install.cmake scripts. If one sets this variable to ON while running the script, it may get warning messages from the script.

Variables That Describe the System

- **APPLE**
- **BORLAND**
- **CMAKE_CL_64**

- **CMAKE_COMPILER_2005**
- **CMAKE_HOST_APPLE**
- **CMAKE_HOST_SYSTEM**
- **CMAKE_HOST_SYSTEM_NAME**
- **CMAKE_HOST_SYSTEM_PROCESSOR**
- **CMAKE_HOST_SYSTEM_VERSION**
- **CMAKE_HOST_UNIX**
- **CMAKE_HOST_WIN32**
- **CMAKE_LIBRARY_ARCHITECTURE**
- **CMAKE_LIBRARY_ARCHITECTURE_REGEX**
- **CMAKE_OBJECT_PATH_MAX**
- **CMAKE_SYSTEM**
- **CMAKE_SYSTEM_NAME**
- **CMAKE_SYSTEM_PROCESSOR**
- **CMAKE_SYSTEM_VERSION**
- **CYGWIN**
- **ENV**
- **MSVC**
- **MSVC10**
- **MSVC11**
- **MSVC12**
- **MSVC60**
- **MSVC70**
- **MSVC71**
- **MSVC80**
- **MSVC90**
- **MSVC_IDE**
- **MSVC_VERSION**
- **UNIX**
- **WIN32**
- **XCODE_VERSION**

- **APPLE**: True if running on Mac OS X.

  Set to true on Mac OS X.

- **BORLAND**: True if the Borland compiler is being used.

  This is set to true if the Borland compiler is being used.

- **CMAKE_CL_64**: Using the 64 bit compiler from Microsoft

  Set to true when using the 64 bit cl compiler from Microsoft.

- **CMAKE_COMPILER_2005**: Using the Visual Studio 2005 compiler from Microsoft

  Set to true when using the Visual Studio 2005 compiler from Microsoft.

- **CMAKE_HOST_APPLE**: True for Apple OS X operating systems.

  Set to true when the host system is Apple OS X.

- **CMAKE_HOST_SYSTEM**: Name of system cmake is being run on.

  The same as CMAKE_SYSTEM but for the host system instead of the target system when cross compiling.

- **CMAKE_HOST_SYSTEM_NAME**: Name of the OS CMake is running on.

  The same as CMAKE_SYSTEM_NAME but for the host system instead of the target system when cross compiling.

- **CMAKE_HOST_SYSTEM_PROCESSOR**: The name of the CPU CMake is running on.

  The same as CMAKE_SYSTEM_PROCESSOR but for the host system instead of the target system when cross compiling.

- **CMAKE_HOST_SYSTEM_VERSION**: OS version CMake is running on.

  The same as CMAKE_SYSTEM_VERSION but for the host system instead of the target system when cross compiling.

- **CMAKE_HOST_UNIX**: True for UNIX and UNIX like operating systems.

  Set to true when the host system is UNIX or UNIX like (i.e. APPLE and CYGWIN).

- **CMAKE_HOST_WIN32**: True on windows systems, including win64.

  Set to true when the host system is Windows and on Cygwin.

- **CMAKE_LIBRARY_ARCHITECTURE**: Target architecture library directory name, if detected.

  This is the value of CMAKE_<lang>_LIBRARY_ARCHITECTURE as detected for one of the enabled languages.

- **CMAKE_LIBRARY_ARCHITECTURE_REGEX**: Regex matching possible target architecture library directory names.

This is used to detect CMAKE_<lang>_LIBRARY_ARCHITECTURE from the implicit linker search path by matching the <arch> name.

- **CMAKE_OBJECT_PATH_MAX**: Maximum object file full-path length allowed by native build tools.

  CMake computes for every source file an object file name that is unique to the source file and deterministic with respect to the full path to the source file. This allows multiple source files in a target to share the same name if they lie in different directories without rebuilding when one is added or removed. However, it can produce long full paths in a few cases, so CMake shortens the path using a hashing scheme when the full path to an object file exceeds a limit. CMake has a built-in limit for each platform that is sufficient for common tools, but some native tools may have a lower limit. This variable may be set to specify the limit explicitly. The value must be an integer no less than 128.

- **CMAKE_SYSTEM**: Name of system cmake is compiling for.

  This variable is the composite of CMAKE_SYSTEM_NAME and CMAKE_SYSTEM_VERSION, like this ${CMAKE_SYSTEM_NAME}-${CMAKE_SYSTEM_VERSION}. If CMAKE_SYSTEM_VERSION is not set, then CMAKE_SYSTEM is the same as CMAKE_SYSTEM_NAME.

- **CMAKE_SYSTEM_NAME**: Name of the OS CMake is building for.

  This is the name of the operating system on which CMake is targeting. On systems that have the uname command, this variable is set to the output of uname -s. Linux, Windows, and Darwin for Mac OS X are the values found on the big three operating systems.

- **CMAKE_SYSTEM_PROCESSOR**: The name of the CPU CMake is building for.

  On systems that support uname, this variable is set to the output of uname -p, on windows it is set to the value of the environment variable PROCESSOR_ARCHITECTURE

- **CMAKE_SYSTEM_VERSION**: OS version CMake is building for.

  A numeric version string for the system, on systems that support uname, this variable is set to the output of uname -r. On other systems this is set to major-minor version numbers.

- **CYGWIN**: True for Cygwin.

  Set to true when using Cygwin.

- **ENV**: Access environment variables.

  Use the syntax $ENV{VAR} to read environment variable VAR. See also the set() command to set ENV{VAR}.

- **MSVC**: True when using Microsoft Visual C

  Set to true when the compiler is some version of Microsoft Visual C.

- **MSVC10**: True when using Microsoft Visual C 10.0

  Set to true when the compiler is version 10.0 of Microsoft Visual C.

- **MSVC11**: True when using Microsoft Visual C 11.0

  Set to true when the compiler is version 11.0 of Microsoft Visual C.

- **MSVC12**: True when using Microsoft Visual C 12.0

  Set to true when the compiler is version 12.0 of Microsoft Visual C.

- **MSVC60**: True when using Microsoft Visual C 6.0

  Set to true when the compiler is version 6.0 of Microsoft Visual C.

- **MSVC70**: True when using Microsoft Visual C 7.0

  Set to true when the compiler is version 7.0 of Microsoft Visual C.

- **MSVC71**: True when using Microsoft Visual C 7.1

  Set to true when the compiler is version 7.1 of Microsoft Visual C.

- **MSVC80**: True when using Microsoft Visual C 8.0

  Set to true when the compiler is version 8.0 of Microsoft Visual C.

- **MSVC90**: True when using Microsoft Visual C 9.0

  Set to true when the compiler is version 9.0 of Microsoft Visual C.

- **MSVC_IDE**: True when using the Microsoft Visual C IDE

  Set to true when the target platform is the Microsoft Visual C IDE, as opposed to the command line compiler.

- **MSVC_VERSION**: The version of Microsoft Visual C/C++ being used if any.

  Known version numbers are:

  ```
  1200 = VS  6.0
  ```

```
1300 = VS  7.0
1310 = VS  7.1
1400 = VS  8.0
1500 = VS  9.0
1600 = VS 10.0
1700 = VS 11.0
1800 = VS 12.0
```

- **`UNIX`**: True for UNIX and UNIX like operating systems.

  Set to true when the target system is UNIX or UNIX like (i.e. APPLE and CYGWIN).

- **`WIN32`**: True on windows systems, including win64.

  Set to true when the target system is Windows.

- **`XCODE_VERSION`**: Version of Xcode (Xcode generator only).

  Under the Xcode generator, this is the version of Xcode as specified in "Xcode.app/Contents/version.plist" (such as "3.1.2").

Variables for Languages

- `CMAKE_<LANG>_ARCHIVE_APPEND`
- `CMAKE_<LANG>_ARCHIVE_CREATE`
- `CMAKE_<LANG>_ARCHIVE_FINISH`
- `CMAKE_<LANG>_COMPILER`
- `CMAKE_<LANG>_COMPILER_ABI`
- `CMAKE_<LANG>_COMPILER_ID`
- `CMAKE_<LANG>_COMPILER_LOADED`
- `CMAKE_<LANG>_COMPILER_VERSION`
- `CMAKE_<LANG>_COMPILE_OBJECT`
- `CMAKE_<LANG>_CREATE_SHARED_LIBRARY`
- `CMAKE_<LANG>_CREATE_SHARED_MODULE`
- `CMAKE_<LANG>_CREATE_STATIC_LIBRARY`
- `CMAKE_<LANG>_FLAGS`
- `CMAKE_<LANG>_FLAGS_DEBUG`
- `CMAKE_<LANG>_FLAGS_MINSIZEREL`
- `CMAKE_<LANG>_FLAGS_RELEASE`
- `CMAKE_<LANG>_FLAGS_RELWITHDEBINFO`
- `CMAKE_<LANG>_IGNORE_EXTENSIONS`
- `CMAKE_<LANG>_IMPLICIT_INCLUDE_DIRECTORIES`
- `CMAKE_<LANG>_IMPLICIT_LINK_DIRECTORIES`
- `CMAKE_<LANG>_IMPLICIT_LINK_FRAMEWORK_DIRECTORIES`
- `CMAKE_<LANG>_IMPLICIT_LINK_LIBRARIES`
- `CMAKE_<LANG>_LIBRARY_ARCHITECTURE`
- `CMAKE_<LANG>_LINKER_PREFERENCE`
- `CMAKE_<LANG>_LINKER_PREFERENCE_PROPAGATES`
- `CMAKE_<LANG>_LINK_EXECUTABLE`
- `CMAKE_<LANG>_OUTPUT_EXTENSION`
- `CMAKE_<LANG>_PLATFORM_ID`
- `CMAKE_<LANG>_SIZEOF_DATA_PTR`
- `CMAKE_<LANG>_SOURCE_FILE_EXTENSIONS`
- `CMAKE_COMPILER_IS_GNU<LANG>`
- `CMAKE_Fortran_MODDIR_DEFAULT`
- `CMAKE_Fortran_MODDIR_FLAG`
- `CMAKE_Fortran_MODOUT_FLAG`
- `CMAKE_INTERNAL_PLATFORM_ABI`
- `CMAKE_USER_MAKE_RULES_OVERRIDE_<LANG>`

- **`CMAKE_<LANG>_ARCHIVE_APPEND`**: Rule variable to append to a static archive.

  This is a rule variable that tells CMake how to append to a static archive. It is used in place of CMAKE_<LANG>_CREATE_STATIC_LIBRARY on some platforms in order to support large object counts. See also CMAKE_<LANG>_ARCHIVE_CREATE and CMAKE_<LANG>_ARCHIVE_FINISH.

- **`CMAKE_<LANG>_ARCHIVE_CREATE`**: Rule variable to create a new static archive.

  This is a rule variable that tells CMake how to create a static archive. It is used in place of CMAKE_<LANG>_CREATE_STATIC_LIBRARY on some platforms in order to support large object counts. See also CMAKE_<LANG>_ARCHIVE_APPEND and CMAKE_<LANG>_ARCHIVE_FINISH.

- **`CMAKE_<LANG>_ARCHIVE_FINISH`**: Rule variable to finish an existing static archive.

  This is a rule variable that tells CMake how to finish a static archive. It is used in place of CMAKE_<LANG>_CREATE_STATIC_LIBRARY on some platforms in order to support large object counts. See also CMAKE_<LANG>_ARCHIVE_CREATE and CMAKE_<LANG>_ARCHIVE_APPEND.

- **CMAKE_<LANG>_COMPILER**: The full path to the compiler for LANG.

  This is the command that will be used as the <LANG> compiler. Once set, you can not change this variable.

- **CMAKE_<LANG>_COMPILER_ABI**: An internal variable subject to change.

  This is used in determining the compiler ABI and is subject to change.

- **CMAKE_<LANG>_COMPILER_ID**: Compiler identification string.

  A short string unique to the compiler vendor. Possible values include:

  ```
  Absoft = Absoft Fortran (absoft.com)
  ADSP = Analog VisualDSP++ (analog.com)
  Clang = LLVM Clang (clang.llvm.org)
  Cray = Cray Compiler (cray.com)
  Embarcadero, Borland = Embarcadero (embarcadero.com)
  G95 = G95 Fortran (g95.org)
  GNU = GNU Compiler Collection (gcc.gnu.org)
  HP = Hewlett-Packard Compiler (hp.com)
  Intel = Intel Compiler (intel.com)
  MIPSpro = SGI MIPSpro (sgi.com)
  MSVC = Microsoft Visual Studio (microsoft.com)
  PGI = The Portland Group (pgroup.com)
  PathScale = PathScale (pathscale.com)
  SDCC = Small Device C Compiler (sdcc.sourceforge.net)
  SunPro = Oracle Solaris Studio (oracle.com)
  TI = Texas Instruments (ti.com)
  TinyCC = Tiny C Compiler (tinycc.org)
  Watcom = Open Watcom (openwatcom.org)
  XL, VisualAge, zOS = IBM XL (ibm.com)
  ```

  This variable is not guaranteed to be defined for all compilers or languages.

- **CMAKE_<LANG>_COMPILER_LOADED**: Defined to true if the language is enabled.

  When language <LANG> is enabled by project() or enable_language() this variable is defined to 1.

- **CMAKE_<LANG>_COMPILER_VERSION**: Compiler version string.

  Compiler version in major[.minor[.patch[.tweak]]] format. This variable is not guaranteed to be defined for all compilers or languages.

- **CMAKE_<LANG>_COMPILE_OBJECT**: Rule variable to compile a single object file.

  This is a rule variable that tells CMake how to compile a single object file for the language <LANG>.

- **CMAKE_<LANG>_CREATE_SHARED_LIBRARY**: Rule variable to create a shared library.

  This is a rule variable that tells CMake how to create a shared library for the language <LANG>.

- **CMAKE_<LANG>_CREATE_SHARED_MODULE**: Rule variable to create a shared module.

  This is a rule variable that tells CMake how to create a shared library for the language <LANG>.

- **CMAKE_<LANG>_CREATE_STATIC_LIBRARY**: Rule variable to create a static library.

  This is a rule variable that tells CMake how to create a static library for the language <LANG>.

- **CMAKE_<LANG>_FLAGS**: Flags for all build types.

  <LANG> flags used regardless of the value of CMAKE_BUILD_TYPE.

- **CMAKE_<LANG>_FLAGS_DEBUG**: Flags for Debug build type or configuration.

  <LANG> flags used when CMAKE_BUILD_TYPE is Debug.

- **CMAKE_<LANG>_FLAGS_MINSIZEREL**: Flags for MinSizeRel build type or configuration.

  <LANG> flags used when CMAKE_BUILD_TYPE is MinSizeRel.Short for minimum size release.

- **CMAKE_<LANG>_FLAGS_RELEASE**: Flags for Release build type or configuration.

  <LANG> flags used when CMAKE_BUILD_TYPE is Release

- **CMAKE_<LANG>_FLAGS_RELWITHDEBINFO**: Flags for RelWithDebInfo type or configuration.

  <LANG> flags used when CMAKE_BUILD_TYPE is RelWithDebInfo. Short for Release With Debug Information.

- **CMAKE_<LANG>_IGNORE_EXTENSIONS**: File extensions that should be ignored by the build.

  This is a list of file extensions that may be part of a project for a given language but are not compiled.

- **CMAKE_<LANG>_IMPLICIT_INCLUDE_DIRECTORIES**: Directories implicitly searched by the compiler for header files.

  CMake does not explicitly specify these directories on compiler command lines for language <LANG>. This prevents system

include directories from being treated as user include directories on some compilers.

- **CMAKE_<LANG>_IMPLICIT_LINK_DIRECTORIES**: Implicit linker search path detected for language <LANG>.

  Compilers typically pass directories containing language runtime libraries and default library search paths when they invoke a linker. These paths are implicit linker search directories for the compiler's language. CMake automatically detects these directories for each language and reports the results in this variable.

  When a library in one of these directories is given by full path to target_link_libraries() CMake will generate the -l<name> form on link lines to ensure the linker searches its implicit directories for the library. Note that some toolchains read implicit directories from an environment variable such as LIBRARY_PATH so keep its value consistent when operating in a given build tree.

- **CMAKE_<LANG>_IMPLICIT_LINK_FRAMEWORK_DIRECTORIES**: Implicit linker framework search path detected for language <LANG>.

  These paths are implicit linker framework search directories for the compiler's language. CMake automatically detects these directories for each language and reports the results in this variable.

- **CMAKE_<LANG>_IMPLICIT_LINK_LIBRARIES**: Implicit link libraries and flags detected for language <LANG>.

  Compilers typically pass language runtime library names and other flags when they invoke a linker. These flags are implicit link options for the compiler's language. CMake automatically detects these libraries and flags for each language and reports the results in this variable.

- **CMAKE_<LANG>_LIBRARY_ARCHITECTURE**: Target architecture library directory name detected for <lang>.

  If the <lang> compiler passes to the linker an architecture-specific system library search directory such as <prefix>/lib/<arch> this variable contains the <arch> name if/as detected by CMake.

- **CMAKE_<LANG>_LINKER_PREFERENCE**: Preference value for linker language selection.

  The "linker language" for executable, shared library, and module targets is the language whose compiler will invoke the linker. The LINKER_LANGUAGE target property sets the language explicitly. Otherwise, the linker language is that whose linker preference value is highest among languages compiled and linked into the target. See also the CMAKE_<LANG>_LINKER_PREFERENCE_PROPAGATES variable.

- **CMAKE_<LANG>_LINKER_PREFERENCE_PROPAGATES**: True if CMAKE_<LANG>_LINKER_PREFERENCE propagates across targets.

  This is used when CMake selects a linker language for a target. Languages compiled directly into the target are always considered. A language compiled into static libraries linked by the target is considered if this variable is true.

- **CMAKE_<LANG>_LINK_EXECUTABLE** : Rule variable to link an executable.

  Rule variable to link an executable for the given language.

- **CMAKE_<LANG>_OUTPUT_EXTENSION**: Extension for the output of a compile for a single file.

  This is the extension for an object file for the given <LANG>. For example .obj for C on Windows.

- **CMAKE_<LANG>_PLATFORM_ID**: An internal variable subject to change.

  This is used in determining the platform and is subject to change.

- **CMAKE_<LANG>_SIZEOF_DATA_PTR**: Size of pointer-to-data types for language <LANG>.

  This holds the size (in bytes) of pointer-to-data types in the target platform ABI. It is defined for languages C and CXX (C++).

- **CMAKE_<LANG>_SOURCE_FILE_EXTENSIONS**: Extensions of source files for the given language.

  This is the list of extensions for a given language's source files.

- **CMAKE_COMPILER_IS_GNU<LANG>**: True if the compiler is GNU.

  If the selected <LANG> compiler is the GNU compiler then this is TRUE, if not it is FALSE. Unlike the other per-language variables, this uses the GNU syntax for identifying languages instead of the CMake syntax. Recognized values of the <LANG> suffix are:

  ```
  CC = C compiler
  CXX = C++ compiler
  G77 = Fortran compiler
  ```

- **CMAKE_Fortran_MODDIR_DEFAULT**: Fortran default module output directory.

  Most Fortran compilers write .mod files to the current working directory. For those that do not, this is set to "." and used when the Fortran_MODULE_DIRECTORY target property is not set.

- **CMAKE_Fortran_MODDIR_FLAG**: Fortran flag for module output directory.

  This stores the flag needed to pass the value of the Fortran_MODULE_DIRECTORY target property to the compiler.

- **CMAKE_Fortran_MODOUT_FLAG**: Fortran flag to enable module output.

  Most Fortran compilers write .mod files out by default. For others, this stores the flag needed to enable module output.

- **CMAKE_INTERNAL_PLATFORM_ABI**: An internal variable subject to change.

This is used in determining the compiler ABI and is subject to change.

- **CMAKE_USER_MAKE_RULES_OVERRIDE_<LANG>**: Specify a CMake file that overrides platform information for <LANG>.

  This is a language-specific version of CMAKE_USER_MAKE_RULES_OVERRIDE loaded only when enabling language <LANG>.

Variables that Control the Build

- CMAKE_<CONFIG>_POSTFIX
- CMAKE_<LANG>_VISIBILITY_PRESET
- CMAKE_ARCHIVE_OUTPUT_DIRECTORY
- CMAKE_AUTOMOC
- CMAKE_AUTOMOC_MOC_OPTIONS
- CMAKE_BUILD_WITH_INSTALL_RPATH
- CMAKE_DEBUG_POSTFIX
- CMAKE_EXE_LINKER_FLAGS
- CMAKE_EXE_LINKER_FLAGS_<CONFIG>
- CMAKE_Fortran_FORMAT
- CMAKE_Fortran_MODULE_DIRECTORY
- CMAKE_GNUtoMS
- CMAKE_INCLUDE_CURRENT_DIR
- CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE
- CMAKE_INSTALL_NAME_DIR
- CMAKE_INSTALL_RPATH
- CMAKE_INSTALL_RPATH_USE_LINK_PATH
- CMAKE_LIBRARY_OUTPUT_DIRECTORY
- CMAKE_LIBRARY_PATH_FLAG
- CMAKE_LINK_DEF_FILE_FLAG
- CMAKE_LINK_DEPENDS_NO_SHARED
- CMAKE_LINK_INTERFACE_LIBRARIES
- CMAKE_LINK_LIBRARY_FILE_FLAG
- CMAKE_LINK_LIBRARY_FLAG
- CMAKE_MACOSX_BUNDLE
- CMAKE_MODULE_LINKER_FLAGS
- CMAKE_MODULE_LINKER_FLAGS_<CONFIG>
- CMAKE_NO_BUILTIN_CHRPATH
- CMAKE_PDB_OUTPUT_DIRECTORY
- CMAKE_POSITION_INDEPENDENT_CODE
- CMAKE_RUNTIME_OUTPUT_DIRECTORY
- CMAKE_SHARED_LINKER_FLAGS
- CMAKE_SHARED_LINKER_FLAGS_<CONFIG>
- CMAKE_SKIP_BUILD_RPATH
- CMAKE_SKIP_INSTALL_RPATH
- CMAKE_STATIC_LINKER_FLAGS
- CMAKE_STATIC_LINKER_FLAGS_<CONFIG>
- CMAKE_TRY_COMPILE_CONFIGURATION
- CMAKE_USE_RELATIVE_PATHS
- CMAKE_VISIBILITY_INLINES_HIDDEN
- CMAKE_WIN32_EXECUTABLE
- EXECUTABLE_OUTPUT_PATH
- LIBRARY_OUTPUT_PATH

- **CMAKE_<CONFIG>_POSTFIX**: Default filename postfix for libraries under configuration <CONFIG>.

  When a non-executable target is created its <CONFIG>_POSTFIX target property is initialized with the value of this variable if it is set.

- **CMAKE_<LANG>_VISIBILITY_PRESET**: Default value for <LANG>_VISIBILITY_PRESET of targets.

  This variable is used to initialize the <LANG>_VISIBILITY_PRESET property on all the targets. See that target property for additional information.

- **CMAKE_ARCHIVE_OUTPUT_DIRECTORY**: Where to put all the ARCHIVE targets when built.

  This variable is used to initialize the ARCHIVE_OUTPUT_DIRECTORY property on all the targets. See that target property for additional information.

- **CMAKE_AUTOMOC**: Whether to handle moc automatically for Qt targets.

  This variable is used to initialize the AUTOMOC property on all the targets. See that target property for additional information.

- **CMAKE_AUTOMOC_MOC_OPTIONS**: Additional options for moc when using automoc (see CMAKE_AUTOMOC).

  This variable is used to initialize the AUTOMOC_MOC_OPTIONS property on all the targets. See that target property for additional information.

- **CMAKE_BUILD_WITH_INSTALL_RPATH**: Use the install path for the RPATH

Normally CMake uses the build tree for the RPATH when building executables etc on systems that use RPATH. When the software is installed the executables etc are relinked by CMake to have the install RPATH. If this variable is set to true then the software is always built with the install path for the RPATH and does not need to be relinked when installed.

- **CMAKE_DEBUG_POSTFIX**: See variable CMAKE_<CONFIG>_POSTFIX.

  This variable is a special case of the more-general CMAKE_<CONFIG>_POSTFIX variable for the DEBUG configuration.

- **CMAKE_EXE_LINKER_FLAGS**: Linker flags to be used to create executables.

  These flags will be used by the linker when creating an executable.

- **CMAKE_EXE_LINKER_FLAGS_<CONFIG>**: Flags to be used when linking an executable.

  Same as CMAKE_C_FLAGS_* but used by the linker when creating executables.

- **CMAKE_Fortran_FORMAT**: Set to FIXED or FREE to indicate the Fortran source layout.

  This variable is used to initialize the Fortran_FORMAT property on all the targets. See that target property for additional information.

- **CMAKE_Fortran_MODULE_DIRECTORY**: Fortran module output directory.

  This variable is used to initialize the Fortran_MODULE_DIRECTORY property on all the targets. See that target property for additional information.

- **CMAKE_GNUtoMS**: Convert GNU import libraries (.dll.a) to MS format (.lib).

  This variable is used to initialize the GNUtoMS property on targets when they are created. See that target property for additional information.

- **CMAKE_INCLUDE_CURRENT_DIR**: Automatically add the current source- and build directories to the include path.

  If this variable is enabled, CMake automatically adds in each directory ${CMAKE_CURRENT_SOURCE_DIR} and ${CMAKE_CURRENT_BINARY_DIR} to the include path for this directory. These additional include directories do not propagate down to subdirectories. This is useful mainly for out-of-source builds, where files generated into the build tree are included by files located in the source tree.

  By default CMAKE_INCLUDE_CURRENT_DIR is OFF.

- **CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE**: Automatically add the current source- and build directories to the INTERFACE_INCLUDE_DIRECTORIES.

  If this variable is enabled, CMake automatically adds for each shared library target, static library target, module target and executable target, ${CMAKE_CURRENT_SOURCE_DIR} and ${CMAKE_CURRENT_BINARY_DIR} to the INTERFACE_INCLUDE_DIRECTORIES.By default CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE is OFF.

- **CMAKE_INSTALL_NAME_DIR**: Mac OS X directory name for installed targets.

  CMAKE_INSTALL_NAME_DIR is used to initialize the INSTALL_NAME_DIR property on all targets. See that target property for more information.

- **CMAKE_INSTALL_RPATH**: The rpath to use for installed targets.

  A semicolon-separated list specifying the rpath to use in installed targets (for platforms that support it). This is used to initialize the target property INSTALL_RPATH for all targets.

- **CMAKE_INSTALL_RPATH_USE_LINK_PATH**: Add paths to linker search and installed rpath.

  CMAKE_INSTALL_RPATH_USE_LINK_PATH is a boolean that if set to true will append directories in the linker search path and outside the project to the INSTALL_RPATH. This is used to initialize the target property INSTALL_RPATH_USE_LINK_PATH for all targets.

- **CMAKE_LIBRARY_OUTPUT_DIRECTORY**: Where to put all the LIBRARY targets when built.

  This variable is used to initialize the LIBRARY_OUTPUT_DIRECTORY property on all the targets. See that target property for additional information.

- **CMAKE_LIBRARY_PATH_FLAG**: The flag to be used to add a library search path to a compiler.

  The flag will be used to specify a library directory to the compiler. On most compilers this is "-L".

- **CMAKE_LINK_DEF_FILE_FLAG** : Linker flag to be used to specify a .def file for dll creation.

  The flag will be used to add a .def file when creating a dll on Windows; this is only defined on Windows.

- **CMAKE_LINK_DEPENDS_NO_SHARED**: Whether to skip link dependencies on shared library files.

  This variable initializes the LINK_DEPENDS_NO_SHARED property on targets when they are created. See that target property for additional information.

- **CMAKE_LINK_INTERFACE_LIBRARIES**: Default value for LINK_INTERFACE_LIBRARIES of targets.

  This variable is used to initialize the LINK_INTERFACE_LIBRARIES property on all the targets. See that target property for additional information.

- **CMAKE_LINK_LIBRARY_FILE_FLAG**: Flag to be used to link a library specified by a path to its file.

  The flag will be used before a library file path is given to the linker. This is needed only on very few platforms.

- **CMAKE_LINK_LIBRARY_FLAG**: Flag to be used to link a library into an executable.

  The flag will be used to specify a library to link to an executable. On most compilers this is "-l".

- **CMAKE_MACOSX_BUNDLE**: Default value for MACOSX_BUNDLE of targets.

  This variable is used to initialize the MACOSX_BUNDLE property on all the targets. See that target property for additional information.

- **CMAKE_MODULE_LINKER_FLAGS**: Linker flags to be used to create modules.

  These flags will be used by the linker when creating a module.

- **CMAKE_MODULE_LINKER_FLAGS_<CONFIG>**: Flags to be used when linking a module.

  Same as CMAKE_C_FLAGS_* but used by the linker when creating modules.

- **CMAKE_NO_BUILTIN_CHRPATH**: Do not use the builtin ELF editor to fix RPATHs on installation.

  When an ELF binary needs to have a different RPATH after installation than it does in the build tree, CMake uses a builtin editor to change the RPATH in the installed copy. If this variable is set to true then CMake will relink the binary before installation instead of using its builtin editor.

- **CMAKE_PDB_OUTPUT_DIRECTORY**: Where to put all the MS debug symbol files from linker.

  This variable is used to initialize the PDB_OUTPUT_DIRECTORY property on all the targets. See that target property for additional information.

- **CMAKE_POSITION_INDEPENDENT_CODE**: Default value for POSITION_INDEPENDENT_CODE of targets.

  This variable is used to initialize the POSITION_INDEPENDENT_CODE property on all the targets. See that target property for additional information.

- **CMAKE_RUNTIME_OUTPUT_DIRECTORY**: Where to put all the RUNTIME targets when built.

  This variable is used to initialize the RUNTIME_OUTPUT_DIRECTORY property on all the targets. See that target property for additional information.

- **CMAKE_SHARED_LINKER_FLAGS**: Linker flags to be used to create shared libraries.

  These flags will be used by the linker when creating a shared library.

- **CMAKE_SHARED_LINKER_FLAGS_<CONFIG>**: Flags to be used when linking a shared library.

  Same as CMAKE_C_FLAGS_* but used by the linker when creating shared libraries.

- **CMAKE_SKIP_BUILD_RPATH**: Do not include RPATHs in the build tree.

  Normally CMake uses the build tree for the RPATH when building executables etc on systems that use RPATH. When the software is installed the executables etc are relinked by CMake to have the install RPATH. If this variable is set to true then the software is always built with no RPATH.

- **CMAKE_SKIP_INSTALL_RPATH**: Do not include RPATHs in the install tree.

  Normally CMake uses the build tree for the RPATH when building executables etc on systems that use RPATH. When the software is installed the executables etc are relinked by CMake to have the install RPATH. If this variable is set to true then the software is always installed without RPATH, even if RPATH is enabled when building. This can be useful for example to allow running tests from the build directory with RPATH enabled before the installation step. To omit RPATH in both the build and install steps, use CMAKE_SKIP_RPATH instead.

- **CMAKE_STATIC_LINKER_FLAGS**: Linker flags to be used to create static libraries.

  These flags will be used by the linker when creating a static library.

- **CMAKE_STATIC_LINKER_FLAGS_<CONFIG>**: Flags to be used when linking a static library.

  Same as CMAKE_C_FLAGS_* but used by the linker when creating static libraries.

- **CMAKE_TRY_COMPILE_CONFIGURATION**: Build configuration used for try_compile and try_run projects.

  Projects built by try_compile and try_run are built synchronously during the CMake configuration step. Therefore a specific build configuration must be chosen even if the generated build system supports multiple configurations.

- **CMAKE_USE_RELATIVE_PATHS**: Use relative paths (May not work!).

  If this is set to TRUE, then CMake will use relative paths between the source and binary tree. This option does not work for more complicated projects, and relative paths are used when possible. In general, it is not possible to move CMake generated makefiles to a different location regardless of the value of this variable.

- **CMAKE_VISIBILITY_INLINES_HIDDEN**: Default value for VISIBILITY_INLINES_HIDDEN of targets.

  This variable is used to initialize the VISIBILITY_INLINES_HIDDEN property on all the targets. See that target property for

additional information.

- **CMAKE_WIN32_EXECUTABLE**: Default value for WIN32_EXECUTABLE of targets.

  This variable is used to initialize the WIN32_EXECUTABLE property on all the targets. See that target property for additional information.

- **EXECUTABLE_OUTPUT_PATH**: Old executable location variable.

  The target property RUNTIME_OUTPUT_DIRECTORY supercedes this variable for a target if it is set. Executable targets are otherwise placed in this directory.

- **LIBRARY_OUTPUT_PATH**: Old library location variable.

  The target properties ARCHIVE_OUTPUT_DIRECTORY, LIBRARY_OUTPUT_DIRECTORY, and RUNTIME_OUTPUT_DIRECTORY supercede this variable for a target if they are set. Library targets are otherwise placed in this directory.

Variables that Provide Information

- CMAKE_AR
- CMAKE_ARGC
- CMAKE_ARGV0
- CMAKE_BINARY_DIR
- CMAKE_BUILD_TOOL
- CMAKE_CACHEFILE_DIR
- CMAKE_CACHE_MAJOR_VERSION
- CMAKE_CACHE_MINOR_VERSION
- CMAKE_CACHE_PATCH_VERSION
- CMAKE_CFG_INTDIR
- CMAKE_COMMAND
- CMAKE_CROSSCOMPILING
- CMAKE_CTEST_COMMAND
- CMAKE_CURRENT_BINARY_DIR
- CMAKE_CURRENT_LIST_DIR
- CMAKE_CURRENT_LIST_FILE
- CMAKE_CURRENT_LIST_LINE
- CMAKE_CURRENT_SOURCE_DIR
- CMAKE_DL_LIBS
- CMAKE_EDIT_COMMAND
- CMAKE_EXECUTABLE_SUFFIX
- CMAKE_EXTRA_GENERATOR
- CMAKE_EXTRA_SHARED_LIBRARY_SUFFIXES
- CMAKE_GENERATOR
- CMAKE_GENERATOR_TOOLSET
- CMAKE_HOME_DIRECTORY
- CMAKE_IMPORT_LIBRARY_PREFIX
- CMAKE_IMPORT_LIBRARY_SUFFIX
- CMAKE_LINK_LIBRARY_SUFFIX
- CMAKE_MAJOR_VERSION
- CMAKE_MAKE_PROGRAM
- CMAKE_MINIMUM_REQUIRED_VERSION
- CMAKE_MINOR_VERSION
- CMAKE_PARENT_LIST_FILE
- CMAKE_PATCH_VERSION
- CMAKE_PROJECT_NAME
- CMAKE_RANLIB
- CMAKE_ROOT
- CMAKE_SCRIPT_MODE_FILE
- CMAKE_SHARED_LIBRARY_PREFIX
- CMAKE_SHARED_LIBRARY_SUFFIX
- CMAKE_SHARED_MODULE_PREFIX
- CMAKE_SHARED_MODULE_SUFFIX
- CMAKE_SIZEOF_VOID_P
- CMAKE_SKIP_RPATH
- CMAKE_SOURCE_DIR
- CMAKE_STANDARD_LIBRARIES
- CMAKE_STATIC_LIBRARY_PREFIX
- CMAKE_STATIC_LIBRARY_SUFFIX
- CMAKE_TWEAK_VERSION
- CMAKE_VERBOSE_MAKEFILE
- CMAKE_VERSION
- CMAKE_VS_PLATFORM_TOOLSET
- CMAKE_XCODE_PLATFORM_TOOLSET
- PROJECT_BINARY_DIR

- **PROJECT_NAME**
- **PROJECT_SOURCE_DIR**
- **[Project name]_BINARY_DIR**
- **[Project name]_SOURCE_DIR**

variables defined by cmake, that give information about the project, and cmake

- **CMAKE_AR**: Name of archiving tool for static libraries.

  This specifies the name of the program that creates archive or static libraries.

- **CMAKE_ARGC**: Number of command line arguments passed to CMake in script mode.

  When run in -P script mode, CMake sets this variable to the number of command line arguments. See also CMAKE_ARGV0, 1, 2 ...

- **CMAKE_ARGV0**: Command line argument passed to CMake in script mode.

  When run in -P script mode, CMake sets this variable to the first command line argument. It then also sets CMAKE_ARGV1, CMAKE_ARGV2, ... and so on, up to the number of command line arguments given. See also CMAKE_ARGC.

- **CMAKE_BINARY_DIR**: The path to the top level of the build tree.

  This is the full path to the top level of the current CMake build tree. For an in-source build, this would be the same as CMAKE_SOURCE_DIR.

- **CMAKE_BUILD_TOOL**: Tool used for the actual build process.

  This variable is set to the program that will be needed to build the output of CMake. If the generator selected was Visual Studio 6, the CMAKE_BUILD_TOOL will be set to msdev, for Unix Makefiles it will be set to make or gmake, and for Visual Studio 7 it set to devenv. For NMake Makefiles the value is nmake. This can be useful for adding special flags and commands based on the final build environment.

- **CMAKE_CACHEFILE_DIR**: The directory with the CMakeCache.txt file.

  This is the full path to the directory that has the CMakeCache.txt file in it. This is the same as CMAKE_BINARY_DIR.

- **CMAKE_CACHE_MAJOR_VERSION**: Major version of CMake used to create the CMakeCache.txt file

  This stores the major version of CMake used to write a CMake cache file. It is only different when a different version of CMake is run on a previously created cache file.

- **CMAKE_CACHE_MINOR_VERSION**: Minor version of CMake used to create the CMakeCache.txt file

  This stores the minor version of CMake used to write a CMake cache file. It is only different when a different version of CMake is run on a previously created cache file.

- **CMAKE_CACHE_PATCH_VERSION**: Patch version of CMake used to create the CMakeCache.txt file

  This stores the patch version of CMake used to write a CMake cache file. It is only different when a different version of CMake is run on a previously created cache file.

- **CMAKE_CFG_INTDIR**: Build-time reference to per-configuration output subdirectory.

  For native build systems supporting multiple configurations in the build tree (such as Visual Studio and Xcode), the value is a reference to a build-time variable specifying the name of the per-configuration output subdirectory. On Makefile generators this evaluates to "." because there is only one configuration in a build tree. Example values:

  ```
  $(IntDir)        = Visual Studio 6
  $(OutDir)        = Visual Studio 7, 8, 9
  $(Configuration) = Visual Studio 10
  $(CONFIGURATION) = Xcode
  .                = Make-based tools
  ```

  Since these values are evaluated by the native build system, this variable is suitable only for use in command lines that will be evaluated at build time. Example of intended usage:

  ```
  add_executable(mytool mytool.c)
  add_custom_command(
    OUTPUT out.txt
    COMMAND ${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_CFG_INTDIR}/mytool
            ${CMAKE_CURRENT_SOURCE_DIR}/in.txt out.txt
    DEPENDS mytool in.txt
    )
  add_custom_target(drive ALL DEPENDS out.txt)
  ```

  Note that CMAKE_CFG_INTDIR is no longer necessary for this purpose but has been left for compatibility with existing projects. Instead add_custom_command() recognizes executable target names in its COMMAND option, so "${CMAKE_CURRENT_BINARY_DIR}/${CMAKE_CFG_INTDIR}/mytool" can be replaced by just "mytool".

  This variable is read-only. Setting it is undefined behavior. In multi-configuration build systems the value of this variable is passed as the value of preprocessor symbol "CMAKE_INTDIR" to the compilation of all source files.

- **CMAKE_COMMAND**: The full path to the cmake executable.

  This is the full path to the CMake executable cmake which is useful from custom commands that want to use the cmake -E option for portable system commands. (e.g. /usr/local/bin/cmake

- **CMAKE_CROSSCOMPILING**: Is CMake currently cross compiling.

  This variable will be set to true by CMake if CMake is cross compiling. Specifically if the build platform is different from the target platform.

- **CMAKE_CTEST_COMMAND**: Full path to ctest command installed with cmake.

  This is the full path to the CTest executable ctest which is useful from custom commands that want to use the cmake -E option for portable system commands.

- **CMAKE_CURRENT_BINARY_DIR**: The path to the binary directory currently being processed.

  This the full path to the build directory that is currently being processed by cmake. Each directory added by add_subdirectory will create a binary directory in the build tree, and as it is being processed this variable will be set. For in-source builds this is the current source directory being processed.

- **CMAKE_CURRENT_LIST_DIR**: Full directory of the listfile currently being processed.

  As CMake processes the listfiles in your project this variable will always be set to the directory where the listfile which is currently being processed (CMAKE_CURRENT_LIST_FILE) is located. The value has dynamic scope. When CMake starts processing commands in a source file it sets this variable to the directory where this file is located. When CMake finishes processing commands from the file it restores the previous value. Therefore the value of the variable inside a macro or function is the directory of the file invoking the bottom-most entry on the call stack, not the directory of the file containing the macro or function definition.

  See also CMAKE_CURRENT_LIST_FILE.

- **CMAKE_CURRENT_LIST_FILE**: Full path to the listfile currently being processed.

  As CMake processes the listfiles in your project this variable will always be set to the one currently being processed. The value has dynamic scope. When CMake starts processing commands in a source file it sets this variable to the location of the file. When CMake finishes processing commands from the file it restores the previous value. Therefore the value of the variable inside a macro or function is the file invoking the bottom-most entry on the call stack, not the file containing the macro or function definition.

  See also CMAKE_PARENT_LIST_FILE.

- **CMAKE_CURRENT_LIST_LINE**: The line number of the current file being processed.

  This is the line number of the file currently being processed by cmake.

- **CMAKE_CURRENT_SOURCE_DIR**: The path to the source directory currently being processed.

  This the full path to the source directory that is currently being processed by cmake.

- **CMAKE_DL_LIBS**: Name of library containing dlopen and dlcose.

  The name of the library that has dlopen and dlclose in it, usually -ldl on most UNIX machines.

- **CMAKE_EDIT_COMMAND**: Full path to cmake-gui or ccmake.

  This is the full path to the CMake executable that can graphically edit the cache. For example, cmake-gui, ccmake, or cmake -i.

- **CMAKE_EXECUTABLE_SUFFIX**: The suffix for executables on this platform.

  The suffix to use for the end of an executable filename if any, .exe on Windows.

  CMAKE_EXECUTABLE_SUFFIX_<LANG> overrides this for language <LANG>.

- **CMAKE_EXTRA_GENERATOR**: The extra generator used to build the project.

  When using the Eclipse, CodeBlocks or KDevelop generators, CMake generates Makefiles (CMAKE_GENERATOR) and additionally project files for the respective IDE. This IDE project file generator is stored in CMAKE_EXTRA_GENERATOR (e.g. "Eclipse CDT4").

- **CMAKE_EXTRA_SHARED_LIBRARY_SUFFIXES**: Additional suffixes for shared libraries.

  Extensions for shared libraries other than that specified by CMAKE_SHARED_LIBRARY_SUFFIX, if any. CMake uses this to recognize external shared library files during analysis of libraries linked by a target.

- **CMAKE_GENERATOR**: The generator used to build the project.

  The name of the generator that is being used to generate the build files. (e.g. "Unix Makefiles", "Visual Studio 6", etc.)

- **CMAKE_GENERATOR_TOOLSET**: Native build system toolset name specified by user.

  Some CMake generators support a toolset name to be given to the native build system to choose a compiler. If the user specifies a toolset name (e.g. via the cmake -T option) the value will be available in this variable.

- **CMAKE_HOME_DIRECTORY**: Path to top of source tree.

This is the path to the top level of the source tree.

- **CMAKE_IMPORT_LIBRARY_PREFIX**: The prefix for import libraries that you link to.

  The prefix to use for the name of an import library if used on this platform.

  CMAKE_IMPORT_LIBRARY_PREFIX_<LANG> overrides this for language <LANG>.

- **CMAKE_IMPORT_LIBRARY_SUFFIX**: The suffix for import libraries that you link to.

  The suffix to use for the end of an import library filename if used on this platform.

  CMAKE_IMPORT_LIBRARY_SUFFIX_<LANG> overrides this for language <LANG>.

- **CMAKE_LINK_LIBRARY_SUFFIX**: The suffix for libraries that you link to.

  The suffix to use for the end of a library filename, .lib on Windows.

- **CMAKE_MAJOR_VERSION**: The Major version of cmake (i.e. the 2 in 2.X.X)

  This specifies the major version of the CMake executable being run.

- **CMAKE_MAKE_PROGRAM**: See CMAKE_BUILD_TOOL.

  This variable is around for backwards compatibility, see CMAKE_BUILD_TOOL.

- **CMAKE_MINIMUM_REQUIRED_VERSION**: Version specified to cmake_minimum_required command

  Variable containing the VERSION component specified in the cmake_minimum_required command.

- **CMAKE_MINOR_VERSION**: The Minor version of cmake (i.e. the 4 in X.4.X).

  This specifies the minor version of the CMake executable being run.

- **CMAKE_PARENT_LIST_FILE**: Full path to the CMake file that included the current one.

  While processing a CMake file loaded by include() or find_package() this variable contains the full path to the file including it.
  The top of the include stack is always the CMakeLists.txt for the current directory. See also CMAKE_CURRENT_LIST_FILE.

- **CMAKE_PATCH_VERSION**: The patch version of cmake (i.e. the 3 in X.X.3).

  This specifies the patch version of the CMake executable being run.

- **CMAKE_PROJECT_NAME**: The name of the current project.

  This specifies name of the current project from the closest inherited PROJECT command.

- **CMAKE_RANLIB**: Name of randomizing tool for static libraries.

  This specifies name of the program that randomizes libraries on UNIX, not used on Windows, but may be present.

- **CMAKE_ROOT**: Install directory for running cmake.

  This is the install root for the running CMake and the Modules directory can be found here. This is commonly used in this format: ${CMAKE_ROOT}/Modules

- **CMAKE_SCRIPT_MODE_FILE**: Full path to the -P script file currently being processed.

  When run in -P script mode, CMake sets this variable to the full path of the script file. When run to configure a CMakeLists.txt file, this variable is not set.

- **CMAKE_SHARED_LIBRARY_PREFIX**: The prefix for shared libraries that you link to.

  The prefix to use for the name of a shared library, lib on UNIX.

  CMAKE_SHARED_LIBRARY_PREFIX_<LANG> overrides this for language <LANG>.

- **CMAKE_SHARED_LIBRARY_SUFFIX**: The suffix for shared libraries that you link to.

  The suffix to use for the end of a shared library filename, .dll on Windows.

  CMAKE_SHARED_LIBRARY_SUFFIX_<LANG> overrides this for language <LANG>.

- **CMAKE_SHARED_MODULE_PREFIX**: The prefix for loadable modules that you link to.

  The prefix to use for the name of a loadable module on this platform.

  CMAKE_SHARED_MODULE_PREFIX_<LANG> overrides this for language <LANG>.

- **CMAKE_SHARED_MODULE_SUFFIX**: The suffix for shared libraries that you link to.

  The suffix to use for the end of a loadable module filename on this platform

  CMAKE_SHARED_MODULE_SUFFIX_<LANG> overrides this for language <LANG>.

- **CMAKE_SIZEOF_VOID_P**: Size of a void pointer.

  This is set to the size of a pointer on the machine, and is determined by a try compile. If a 64 bit size is found, then the library search path is modified to look for 64 bit libraries first.

- **CMAKE_SKIP_RPATH**: If true, do not add run time path information.

  If this is set to TRUE, then the rpath information is not added to compiled executables. The default is to add rpath information if the platform supports it. This allows for easy running from the build tree. To omit RPATH in the install step, but not the build step, use CMAKE_SKIP_INSTALL_RPATH instead.

- **CMAKE_SOURCE_DIR**: The path to the top level of the source tree.

  This is the full path to the top level of the current CMake source tree. For an in-source build, this would be the same as CMAKE_BINARY_DIR.

- **CMAKE_STANDARD_LIBRARIES**: Libraries linked into every executable and shared library.

  This is the list of libraries that are linked into all executables and libraries.

- **CMAKE_STATIC_LIBRARY_PREFIX**: The prefix for static libraries that you link to.

  The prefix to use for the name of a static library, lib on UNIX.

  CMAKE_STATIC_LIBRARY_PREFIX_<LANG> overrides this for language <LANG>.

- **CMAKE_STATIC_LIBRARY_SUFFIX**: The suffix for static libraries that you link to.

  The suffix to use for the end of a static library filename, .lib on Windows.

  CMAKE_STATIC_LIBRARY_SUFFIX_<LANG> overrides this for language <LANG>.

- **CMAKE_TWEAK_VERSION**: The tweak version of cmake (i.e. the 1 in X.X.X.1).

  This specifies the tweak version of the CMake executable being run. Releases use tweak < 20000000 and development versions use the date format CCYYMMDD for the tweak level.

- **CMAKE_VERBOSE_MAKEFILE**: Create verbose makefiles if on.

  This variable defaults to false. You can set this variable to true to make CMake produce verbose makefiles that show each command line as it is used.

- **CMAKE_VERSION**: The full version of cmake in major.minor.patch[.tweak[-id]] format.

  This specifies the full version of the CMake executable being run. This variable is defined by versions 2.6.3 and higher. See variables CMAKE_MAJOR_VERSION, CMAKE_MINOR_VERSION, CMAKE_PATCH_VERSION, and CMAKE_TWEAK_VERSION for individual version components. The [-id] component appears in non-release versions and may be arbitrary text.

- **CMAKE_VS_PLATFORM_TOOLSET**: Visual Studio Platform Toolset name.

  VS 10 and above use MSBuild under the hood and support multiple compiler toolchains. CMake may specify a toolset explicitly, such as "v110" for VS 11 or "Windows7.1SDK" for 64-bit support in VS 10 Express. CMake provides the name of the chosen toolset in this variable.

- **CMAKE_XCODE_PLATFORM_TOOLSET**: Xcode compiler selection.

  Xcode supports selection of a compiler from one of the installed toolsets. CMake provides the name of the chosen toolset in this variable, if any is explicitly selected (e.g. via the cmake -T option).

- **PROJECT_BINARY_DIR**: Full path to build directory for project.

  This is the binary directory of the most recent PROJECT command.

- **PROJECT_NAME**: Name of the project given to the project command.

  This is the name given to the most recent PROJECT command.

- **PROJECT_SOURCE_DIR**: Top level source directory for the current project.

  This is the source directory of the most recent PROJECT command.

- **[Project name]_BINARY_DIR**: Top level binary directory for the named project.

  A variable is created with the name used in the PROJECT command, and is the binary directory for the project. This can be useful when SUBDIR is used to connect several projects.

- **[Project name]_SOURCE_DIR**: Top level source directory for the named project.

  A variable is created with the name used in the PROJECT command, and is the source directory for the project. This can be useful when add_subdirectory is used to connect several projects.

Copyright

Neither the names of Kitware, Inc., the Insight Software Consortium, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

See Also

- **Home Page**
- **Frequently Asked Questions**
- **Online Documentation**
- **Mailing List**

The following resources are available to get help using CMake:

- **Home Page**: http://www.cmake.org

  The primary starting point for learning about CMake.

- **Frequently Asked Questions**: http://www.cmake.org/Wiki/CMake_FAQ

  A Wiki is provided containing answers to frequently asked questions.

- **Online Documentation**: http://www.cmake.org/HTML/Documentation.html

  Links to available documentation may be found on this web page.

- **Mailing List**: http://www.cmake.org/HTML/MailingLists.html

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