NAME

cmake-variables − CMake Variables Reference

This page documents variables that are provided by CMake or have meaning to CMake when set by project code.

For general information on variables, see the *Variables* section in the cmake−language manual.

NOTE:

CMake reserves identifiers that:

- begin with **CMAKE_** (upper−, lower−, or mixed−case), or
- begin with **_CMAKE_** (upper−, lower−, or mixed−case), or
- begin with **_** followed by the name of any *CMake Command*.

VARIABLES THAT PROVIDE INFORMATION

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*.

When run in *cmake −P* script mode, CMake sets the variables **CMAKE_BINARY_DIR**, *CMAKE_SOURCE_DIR*, *CMAKE_CURRENT_BINARY_DIR* and *CMAKE_CURRENT_SOURCE_DIR* to the current working directory.

CMAKE_BUILD_TOOL

This variable exists only for backwards compatibility. It contains the same value as *CMAKE_MAKE_PROGRAM*. Use that variable instead.

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_CACHEFILE_DIR
This variable is used internally by CMake, and may not be set during the first configuration of a build tree. When it is set, it has the same value as *CMAKE_BINARY_DIR*. Use that variable instead.

CMAKE_CFG_INTDIR
Deprecated since version 3.21: This variable has poor support on *Ninja Multi−Config*, and predates the existence of the *$<CONFIG>* generator expression. Use **$<CONFIG>** instead.

Build−time reference to per−configuration output subdirectory.

For native build systems supporting multiple configurations in the build tree (such as *Visual Studio Generators* and *Xcode*), the value is a reference to a build−time variable specifying the name of the per−configuration output subdirectory. On*Mak efile Generators* this evaluates to **.** because there is only one configuration in a build tree. Example values:

$(Configuration)	Visual Studio
$(CONFIGURATION)	Xcode
.	Make−based tools
.	Ninja
${CONFIGURATION}	Ninja Multi−Config

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(1)* executable.

This is the full path to the CMake executable *cmake(1)* 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_CPACK_COMMAND
Added in version 3.13.

Full path to *cpack(1)* command installed with CMake.

This is the full path to the CPack executable *cpack(1)* that can be used for custom commands or tests to invoke CPack commands.

CMAKE_CROSSCOMPILING
This variable is set by CMake to indicate whether it is cross compiling, but note limitations discussed below.

This variable will be set to true by CMake if the *CMAKE_SYSTEM_NAME* variable has been set manually (i.e. in a toolchain file or as a cache entry from the *cmake* command line). In most cases, manually setting *CMAKE_SYSTEM_NAME* will only be done when cross compiling since, if not manually set, it will be given the same value as *CMAKE_HOST_SYSTEM_NAME*, which is correct for the non−cross−compiling case. In the event that *CMAKE_SYSTEM_NAME* is manually set to the same value as *CMAKE_HOST_SYSTEM_NAME*, then **CMAKE_CROSSCOMPILING** will still be set to true.

Another case to be aware of is that builds targeting Apple platforms other than macOS are handled differently to other cross compiling scenarios. Rather than relying on *CMAKE_SYSTEM_NAME* to select the target platform, Apple device builds use *CMAKE_OSX_SYSROOT* to select the appropriate SDK, which indirectly determines the target platform. Furthermore, when using the *Xcode* generator, developers can switch between device and simulator builds at build time rather than having a single choice at configure time, so the concept of whether the build is cross compiling or not is more complex. Therefore, the use of **CMAKE_CROSSCOMPILING** is not recommended for projects targeting Apple devices.

CMAKE_CROSSCOMPILING_EMULATOR
Added in version 3.3.

This variable is only used when *CMAKE_CROSSCOMPILING* is on. It should point to a command on the host system that can run executable built for the target system.

Added in version 3.15: If this variable contains a *semicolon−separated list*, then the first value is the command and remaining values are its arguments.

Added in version 3.28: This variable can be initialized via an *CMAKE_CROSSCOMPILING_EMULATOR* environment variable.

The command will be used to run *try_run()* generated executables, which avoids manual population of the **TryRunResults.cmake** file.

This variable is also used as the default value for the *CROSSCOMPILING_EMULATOR* target property of executables. However, while *generator expressions* are supported by the target property (since CMake 3.29), they are *not* supported by this variable's *try_run()* functionality.

CMAKE_CTEST_COMMAND
Full path to *ctest(1)* command installed with CMake.

This is the full path to the CTest executable *ctest(1)* that can be used for custom commands or tests to invoke CTest commands.

CMAKE_CURRENT_BINARY_DIR
The path to the binary directory currently being processed.

This is 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.

When run in *cmake −P* script mode, CMake sets the variables *CMAKE_BINARY_DIR*, *CMAKE_SOURCE_DIR*, **CMAKE_CURRENT_BINARY_DIR** and *CMAKE_CURRENT_SOURCE_DIR* to the current working directory.

CMAKE_CURRENT_FUNCTION
Added in version 3.17.

When executing code inside a *function()*, this variable contains the name of the current function. It can be useful for diagnostic or debug messages.

See also *CMAKE_CURRENT_FUNCTION_LIST_DIR*, *CMAKE_CURRENT_FUNCTION_LIST_FILE* and *CMAKE_CURRENT_FUNCTION_LIST_LINE*.

CMAKE_CURRENT_FUNCTION_LIST_DIR
Added in version 3.17.

When executing code inside a *function()*, this variable contains the full directory of the listfile that defined the current function.

It is quite common practice in CMake for modules to use some additional files, such as templates to be copied in after substituting CMake variables. In such cases, a function needs to know where to locate those files in a way that doesn't depend on where the function is called. Without **CMAKE_CURRENT_FUNCTION_LIST_DIR**, the code to do that would typically use the following pattern:

```
set(_THIS_MODULE_BASE_DIR "${CMAKE_CURRENT_LIST_DIR}")

function(foo)
  configure_file(
    "${_THIS_MODULE_BASE_DIR}/some.template.in"
    some.output
  )
endfunction()
```

Using **CMAKE_CURRENT_FUNCTION_LIST_DIR** inside the function instead eliminates the need for the extra variable which would otherwise be visible outside the function's scope. The above example can be written in the more concise and more robust form:

```
function(foo)
  configure_file(
    "${CMAKE_CURRENT_FUNCTION_LIST_DIR}/some.template.in"
    some.output
  )
endfunction()
```

See also *CMAKE_CURRENT_FUNCTION*, *CMAKE_CURRENT_FUNCTION_LIST_FILE*, *CMAKE_CURRENT_FUNCTION_LIST_LINE* and *CMAKE_CURRENT_LIST_DIR*.

CMAKE_CURRENT_FUNCTION_LIST_FILE
Added in version 3.17.

When executing code inside a *function()*, this variable contains the full path to the listfile that defined the current function.

See also *CMAKE_CURRENT_FUNCTION*, *CMAKE_CURRENT_FUNCTION_LIST_DIR*, *CMAKE_CURRENT_FUNCTION_LIST_LINE* and *CMAKE_CURRENT_LIST_FILE*.

CMAKE_CURRENT_FUNCTION_LIST_LINE
Added in version 3.17.

When executing code inside a *function()*, this variable contains the line number in the listfile where the current function was defined.

See also *CMAKE_CURRENT_FUNCTION*, *CMAKE_CURRENT_FUNCTION_LIST_DIR*, *CMAKE_CURRENT_FUNCTION_LIST_FILE* and *CMAKE_CURRENT_LIST_LINE*.

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* and *CMAKE_CURRENT_FUNCTION_LIST_DIR*.

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* and *CMAKE_CURRENT_FUNCTION_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.

If CMake is currently processing deferred calls scheduled by the *cmake_language(DEFER)* command, this variable evaluates to **DEFERRED** instead of a specific line number.

See also *CMAKE_CURRENT_FUNCTION_LIST_LINE*.

CMAKE_CURRENT_SOURCE_DIR
The path to the source directory currently being processed.

This is the full path to the source directory that is currently being processed by cmake.

When run in *cmake −P* script mode, CMake sets the variables *CMAKE_BINARY_DIR*, *CMAKE_SOURCE_DIR*, *CMAKE_CURRENT_BINARY_DIR* and **CMAKE_CURRENT_SOURCE_DIR** to the current working directory.

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:

- *AUTOUIC_OPTIONS*
- *COMPILE_DEFINITIONS*
- *COMPILE_FEATURES*
- *COMPILE_OPTIONS*
- *INCLUDE_DIRECTORIES*
- *LINK_DIRECTORIES*
- *LINK_OPTIONS*
- *POSITION_INDEPENDENT_CODE*
- *SOURCES*

target properties and any other property listed in *COMPATIBLE_INTERFACE_STRING* and other **COMPATIBLE_INTERFACE_** properties. It outputs an origin for each entry in the target property. Default is unset.

CMAKE_DIRECTORY_LABELS
Added in version 3.10.

Specify labels for the current directory.

This is used to initialize the *LABELS* directory property.

CMAKE_DL_LIBS
Name of library containing **dlopen** and **dlclose**.

The name of the library that has **dlopen** and **dlclose** in it, usually **−ldl** on most UNIX machines.

CMAKE_DOTNET_SDK
Added in version 3.23.

Default value for *DOTNET_SDK* property of targets.

This variable is used to initialize the *DOTNET_SDK* property on all targets. See that target property for additional information.

CMAKE_DOTNET_TARGET_FRAMEWORK
Added in version 3.17.

Default value for *DOTNET_TARGET_FRAMEWORK* property of targets.

This variable is used to initialize the *DOTNET_TARGET_FRAMEWORK* property on all targets. See that

target property for additional information.

Setting **CMAKE_DOTNET_TARGET_FRAMEWORK** may be necessary when working with **C#** and newer .NET framework versions to avoid referencing errors with the **ALL_BUILD** CMake target.

This variable is only evaluated for *Visual Studio Generators* VS 2010 and above.

CMAKE_DOTNET_TARGET_FRAMEWORK_VERSION
Added in version 3.12.

Default value for *DOTNET_TARGET_FRAMEWORK_VERSION* property of targets.

This variable is used to initialize the *DOTNET_TARGET_FRAMEWORK_VERSION* property on all targets. See that target property for additional information. When set, *CMAKE_DOTNET_TARGET_FRAMEWORK* takes precednece over this variable. See that variable or the associated target property *DOTNET_TARGET_FRAMEWORK* for additional information.

Setting **CMAKE_DOTNET_TARGET_FRAMEWORK_VERSION** may be necessary when working with **C#** and newer .NET framework versions to avoid referencing errors with the **ALL_BUILD** CMake target.

This variable is only evaluated for *Visual Studio Generators* VS 2010 and above.

CMAKE_EDIT_COMMAND
Full path to *cmake−gui(1)* or *ccmake(1)*. Defined only for*Mak efile Generators* and *Ninja Generators* when not using any *Extra Generators*.

This is the full path to the CMake executable that can graphically edit the cache. For example, *cmake−gui(1)* or *ccmake(1)*.

CMAKE_EXECUTABLE_SUFFIX
The suffix for executables on the target 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>**.

See the *CMAKE_HOST_EXECUTABLE_SUFFIX* variable for the executable suffix on the host platform.

CMAKE_EXECUTABLE_SUFFIX_<LANG>
The suffix to use for the end of an executable filename of **<LANG>** compiler target architecture, if any.

It overrides *CMAKE_EXECUTABLE_SUFFIX* for language **<LANG>**.

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_FIND_DEBUG_MODE
Added in version 3.17.

Print extra find call information for the following commands to standard error:

- *find_program()*
- *find_library()*
- *find_file()*
- *find_path()*
- *find_package()*

Output is designed for human consumption and not for parsing. Enabling this variable is equivalent to using *cmake −−debug−find* with the added ability to enable debugging for a subset of find calls.

```
set(CMAKE_FIND_DEBUG_MODE TRUE)
find_program(...)
set(CMAKE_FIND_DEBUG_MODE FALSE)
```

Default is unset.

CMAKE_FIND_PACKAGE_NAME
Added in version 3.1.1.

Defined by the *find_package()* command while loading a find module to record the caller−specified package name. See command documentation for details.

CMAKE_FIND_PACKAGE_REDIRECTS_DIR
Added in version 3.24.

This read−only variable specifies a directory that the *find_package()* command will check first before searching anywhere else for a module or config package file. A config package file in this directory will always be found in preference to any other Find module file or config package file.

The primary purpose of this variable is to facilitate integration between *find_package()* and *FetchContent_MakeAvailable()*. The latter command may create files in the**CMAKE_FIND_P ACKAGE_REDIRECTS_DIR** directory when it populates a dependency. This allows subsequent calls to *find_package()* for the same dependency to reuse the populated contents instead of trying to satisfy the dependency from somewhere external to the build. Projects may also want to write files into this directory in some situations (see *Integrating With find_package()* for examples).

The directory that **CMAKE_FIND_PACKAGE_REDIRECTS_DIR** points to will always be erased and recreated empty at the start of every CMake run. Any files written into this directory during the CMake run will be lost the next time CMake configures the project.

CMAKE_FIND_PACKAGE_REDIRECTS_DIR is only set in CMake project mode. It is not set when CMake is run in script mode (i.e. *cmake −P*).

CMAKE_FIND_PACKAGE_SORT_DIRECTION
Added in version 3.7.

The sorting direction used by *CMAKE_FIND_PACKAGE_SORT_ORDER*. It can assume one of the following values:

DEC Default. Ordering is done in descending mode. The highest folder found will be tested first.

ASC Ordering is done in ascending mode. The lowest folder found will be tested first.

If *CMAKE_FIND_PACKAGE_SORT_ORDER* is not set or is set to **NONE** this variable has no effect.

CMAKE_FIND_PACKAGE_SORT_ORDER
Added in version 3.7.

The default order for sorting directories which match a search path containing a glob expression found using *find_package()*. It can assume one of the following values:

NONE Default. No attempt is done to sort directories. The first valid package found will be selected.

NAME Sort directories lexicographically before searching.

NATURAL
Sort directories using natural order (see **strverscmp(3)** manual), i.e. such that contiguous digits are compared as whole numbers.

Natural sorting can be employed to return the highest version when multiple versions of the same library are available to be found by *find_package()*. For example suppose that the following libraries have package configuration files on disk, in a directory of the same name, with all such directories residing in the same parent directory:

- libX−1.1.0
- libX−1.2.9
- libX−1.2.10

By setting **NATURAL** order we can select the one with the highest version number **libX−1.2.10**.

```
set(CMAKE_FIND_PACKAGE_SORT_ORDER NATURAL)
find_package(libX CONFIG)
```

The sort direction can be controlled using the *CMAKE_FIND_PACKAGE_SORT_DIRECTION* variable (by default descending, e.g. lib−B will be tested before lib−A).

CMAKE_GENERATOR
The generator used to build the project. See *cmake−generators(7)*.

The name of the generator that is being used to generate the build files. (e.g. **Unix Makefiles**, **Ninja**, etc.)

The value of this variable should never be modified by project code. A generator may be selected via the *cmake −G* option, interactively in *cmake−gui(1)*, or via the *CMAKE_GENERATOR* environment variable.

CMAKE_GENERATOR_INSTANCE
Added in version 3.11.

Generator−specific instance specification provided by user.

Some CMake generators support selection of an instance of the native build system when multiple instances are available. If the user specifies an instance (e.g. by setting this cache entry or via the *CMAKE_GENERATOR_INSTANCE* environment variable), or after a default instance is chosen when a build tree is first configured, the value will be available in this variable.

The value of this variable should never be modified by project code. A toolchain file specified by the *CMAKE_TOOLCHAIN_FILE* variable may initialize **CMAKE_GENERATOR_INSTANCE** as a cache entry. Once a given build tree has been initialized with a particular value for this variable, changing the value has undefined behavior.

Instance specification is supported only on specific generators.

Visual Studio Instance Selection

Visual Studio Generators support instance specification for Visual Studio 2017 and above. The **CMAKE_GENERATOR_INSTANCE** variable may be set as a cache entry selecting an instance of Visual Studio via one of the following forms:

- **location**
- **location[,key=value]***
- **key=value[,key=value]***

The **location** specifies the absolute path to the top−level directory of the VS installation.

The **key=value** pairs form a comma−separated list of options to specify details of the instance selection. Supported pairs are:

version=<major>.<minor>.<date>.<build>
Added in version 3.23.

Specify the 4−component VS Build Version, a.k.a. Build Number.

The components are:

<major>.<minor>
The VS major and minor version numbers. These are the same as the release version numbers.

<date>
A build date in the format **MMMDD**, where **MMM** is a month index since an epoch used by Microsoft, and **DD** is a day in that month.

<build>
A build index on the day represented by **<date>**.

The build number is reported by **vswhere** as **installationVersion**. For example, VS 16.11.10 has build number **16.11.32126.315**.

Added in version 3.23: A portable VS instance, which is not known to the Visual Studio Installer, may be specified by providing both **location** and **version=**.

If the value of **CMAKE_GENERATOR_INSTANCE** is not specified explicitly by the user or a toolchain file, CMake queries the Visual Studio Installer to locate VS instances, chooses one, and sets the variable as a cache entry to hold the value persistently. If an environment variable of the form **VS##0COMNTOOLS**, where **##** the Visual Studio major version number, is set and points to the **Common7/Tools** directory within one of the VS instances, that instance will be used. Otherwise, if more than one VS instance is installed we do not define which one is chosen by default.

The VS version build number of the selected VS instance is provided in the *CMAKE_VS_VERSION_BUILD_NUMBER* variable.

CMAKE_GENERATOR_PLATFORM
Added in version 3.1.

Generator−specific target platform specification provided by user.

Some CMake generators support a target platform name to be given to the native build system to choose a compiler toolchain. If the user specifies a platform name (e.g. via the *cmake −A* option or via the *CMAKE_GENERATOR_PLATFORM* environment variable) the value will be available in this variable.

The value of this variable should never be modified by project code. A toolchain file specified by the *CMAKE_TOOLCHAIN_FILE* variable may initialize **CMAKE_GENERATOR_PLATFORM**. Once a given build tree has been initialized with a particular value for this variable, changing the value has undefined behavior.

Platform specification is supported only on specific generators:

- For *Visual Studio Generators* with VS 2005 and above this specifies the target architecture.
- For *Green Hills MULTI* this specifies the target architecture.

See native build system documentation for allowed platform names.

Visual Studio Platform Selection

The *Visual Studio Generators* support platform specification using one of these forms:

- **platform**
- **platform[,key=value]***
- **key=value[,key=value]***

The **platform** specifies the target platform (VS target architecture), such as **x64**, **ARM64**, or **Win32**. The selected platform name is provided in the *CMAKE_VS_PLATFORM_NAME* variable.

The **key=value** pairs form a comma−separated list of options to specify generator−specific details of the platform selection. Supported pairs are:

version=<version>

Added in version 3.27.

Specify the Windows SDK version to use. This is supported by VS 2015 and above when targeting Windows or Windows Store. CMake will set the *CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION* variable to the selected SDK version.

The **<version>** may be one of:

10.0 Specify that any 10.0 SDK version may be used, and let Visual Studio pick one. This is supported by VS 2019 and above.

10.0.<build>.<increment>

Specify the exact 4−component SDK version, e.g., **10.0.19041.0**. The specified version of the SDK must be installed. It may not exceed the value of *CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION_MAXIMUM*, if that variable is set.

8.1 Specify the 8.1 SDK version. This is always supported by VS 2015. On VS 2017 and above the 8.1 SDK must be installed.

If the **version** field is not specified, CMake selects a version as described in the *CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION* variable documentation.

CMAKE_GENERATOR_TOOLSET

Native build system toolset specification provided by user.

Some CMake generators support a toolset specification to tell the native build system how to choose a

compiler. If the user specifies a toolset (e.g. via the*cmak e −T* option or via the *CMAKE_GENERATOR_TOOLSET* environment variable) the value will be available in this variable.

The value of this variable should never be modified by project code. A toolchain file specified by the *CMAKE_TOOLCHAIN_FILE* variable may initialize **CMAKE_GENERATOR_TOOLSET**. Once a given build tree has been initialized with a particular value for this variable, changing the value has undefined behavior.

Toolset specification is supported only on specific generators:

- *Visual Studio Generators* for VS 2010 and above
- The *Xcode* generator for Xcode 3.0 and above
- The *Green Hills MULTI* generator

See native build system documentation for allowed toolset names.

Visual Studio Toolset Selection

The *Visual Studio Generators* support toolset specification using one of these forms:

- **toolset**
- **toolset[,key=value]***
- **key=value[,key=value]***

The **toolset** specifies the toolset name. The selected toolset name is provided in the *CMAKE_VS_PLATFORM_TOOLSET* variable.

The **key=value** pairs form a comma−separated list of options to specify generator−specific details of the toolset selection. Supported pairs are:

cuda=<version>|<path>
: Specify the CUDA toolkit version to use or the path to a standalone CUDA toolkit directory. Supported by VS 2010 and above. The version can only be used with the CUDA toolkit VS integration globally installed. See the *CMAKE_VS_PLATFORM_TOOLSET_CUDA* and *CMAKE_VS_PLATFORM_TOOLSET_CUDA_CUSTOM_DIR* variables.

fortran=<compiler>
: Added in version 3.29.

  Specify the Fortran compiler to use, among those that have the required Visual Studio Integration feature installed. The value may be one of:

  ifort Intel classic Fortran compiler.

  ifx Intel oneAPI Fortran compiler.

  See the *CMAKE_VS_PLATFORM_TOOLSET_FORTRAN* variable.

host=<arch>
: Specify the host tools architecture as **x64** or **x86**. Supported by VS 2013 and above. See the *CMAKE_VS_PLATFORM_TOOLSET_HOST_ARCHITECTURE* variable.

version=<version>
: Specify the toolset version to use. Supported by VS 2017 and above with the specified toolset installed. See the*CMAKE_VS_PLA TFORM_TOOLSET_VERSION* variable.

VCTargetsPath=<path>
: Specify an alternative **VCTargetsPath** value for Visual Studio project files. This allows use of VS platform extension configuration files (**.props** and **.targets**) that are not installed with VS.

Visual Studio Toolset Customization

These are unstable interfaces with no compatibility guarantees because they hook into undocumented internal CMake implementation details. Institutions may use these to internally maintain support for non−public Visual Studio platforms and toolsets, but must accept responsibility to make updates as changes are made to CMake.

Additional **key=value** pairs are available:

customFlagTableDir=<path>

Added in version 3.21.

Specify the absolute path to a directory from which to load custom flag tables stored as JSON documents with file names of the form **<platform>_<toolset>_<tool>.json** or **<platform>_<tool>.json**, where **<platform>** is the *CMAKE_VS_PLATFORM_NAME*, **<toolset>** is the *CMAKE_VS_PLATFORM_TOOLSET*, and **<tool>** is the tool for which the flag table is meant. **This naming pattern is an internal CMake implementation detail.** The **<tool>** names are undocumented. The format of the**.json** flag table files is undocumented.

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_JOB_POOL_COMPILE

This variable is used to initialize the *JOB_POOL_COMPILE* property on all the targets. See *JOB_POOL_COMPILE* for additional information.

CMAKE_JOB_POOL_LINK

This variable is used to initialize the *JOB_POOL_LINK* property on all the targets. See *JOB_POOL_LINK* for additional information.

CMAKE_JOB_POOL_PRECOMPILE_HEADER

Added in version 3.17.

This variable is used to initialize the *JOB_POOL_PRECOMPILE_HEADER* property on all the targets. See *JOB_POOL_PRECOMPILE_HEADER* for additional information.

CMAKE_JOB_POOLS

Added in version 3.11.

If the *JOB_POOLS* global property is not set, the value of this variable is used in its place. See *JOB_POOLS* for additional information.

CMAKE_<LANG>_COMPILER_AR

Added in version 3.9.

A wrapper around **ar** adding the appropriate **−−plugin** option for the compiler.

See also *CMAKE_AR*.

CMAKE_<LANG>_COMPILER_FRONTEND_VARIANT
Added in version 3.14.

Identification string of the compiler frontend variant.

Some compilers have multiple, different frontends for accepting command line options. (For example **Clang** originally only had a frontend compatible with the **GNU** compiler but since its port to Windows (**Clang−Cl**) it now also supports a frontend compatible with **MSVC**.) When CMake detects such a compiler it sets this variable to what would have been the *CMAKE_<LANG>_COMPILER_ID* for the compiler whose frontend it resembles.

NOTE:
In other words, this variable describes what command line options and language extensions the compiler frontend expects.

Changed in version 3.26: This variable is set for **GNU**, **MSVC**, and **AppleClang** compilers that have only one frontend variant.

CMAKE_<LANG>_COMPILER_LINKER
Added in version 3.29.

The full path to the linker for **LANG**.

This is the command that will be used as the **<LANG>** linker.

This variable is not guaranteed to be defined for all linkers or languages.

NOTE:
This variable is read−only. It must not be set by the user. To select a specific linker, use the *CMAKE_LINKER_TYPE* variable or the *LINKER_TYPE* target property.

CMAKE_<LANG>_COMPILER_LINKER_FRONTEND_VARIANT
Added in version 3.29.

Identification string of the linker frontend variant.

Some linkers have multiple, different frontends for accepting command line options. For example, **LLVM LLD** originally only had a frontend compatible with the **GNU** compiler, but since its port to Windows (**lld−link**), it now also supports a frontend compatible with **MSVC**. When CMake detects such a linker, it sets this variable to what would have been the *CMAKE_<LANG>_COMPILER_LINKER_ID* for the linker whose frontend it resembles.

NOTE:
In other words, this variable describes what command line options and language extensions the linker frontend expects.

CMAKE_<LANG>_COMPILER_LINKER_ID
Added in version 3.29.

Linker identification string.

A short string unique to the linker vendor. Possible values include:

Value	Name
AppleClang	Apple Clang
LLD	*LLVM LLD*
GNU	*GNU Binutils − ld linker* (also known as **bfd**)
GNUgold	*GNU Binutils − gold linker*
MSVC	*Microsoft Visual Studio*
MOLD	*mold: A Modern Linker*, or on Apple the *sold* linker
AIX	AIX system linker
Solaris	SunOS system linker

This variable is not guaranteed to be defined for all linkers or languages.

CMAKE_<LANG>_COMPILER_LINKER_VERSION
Added in version 3.29.

Linker version string.

Linker version in major[.minor[.patch[.tweak]]] format. This variable is not guaranteed to be defined for all linkers or languages.

CMAKE_<LANG>_COMPILER_RANLIB
Added in version 3.9.

A wrapper around **ranlib** adding the appropriate **−−plugin** option for the compiler.

See also *CMAKE_RANLIB*.

CMAKE_<LANG>_DEVICE_LINK_MODE
Added in version 4.0.

Defines how the device link step is done. The possible values are:

DRIVER
The compiler is used as driver for the device link step.

LINKER
The linker is used directly for the device link step.

CMAKE_<LANG>_LINK_LIBRARY_SUFFIX
Added in version 3.16.

Language−specific suffix for libraries that you link to.

The suffix to use for the end of a library filename, **.lib** on Windows.

CMAKE_<LANG>_LINK_MODE
Added in version 4.0.

Defines how the link step is done. The possible values are:

DRIVER
The compiler is used as driver for the link step.

LINKER
The linker is used directly for the link step.

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_LINK_SEARCH_END_STATIC
Added in version 3.4.

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.

This variable is used to initialize the target property *LINK_SEARCH_END_STATIC* for all targets. If set, its value is also used by the *try_compile()* command.

See also *CMAKE_LINK_SEARCH_START_STATIC*.

CMAKE_LINK_SEARCH_START_STATIC
Added in version 3.4.

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).

This variable is used to initialize the target property *LINK_SEARCH_START_STATIC* for all targets. If set, its value is also used by the *try_compile()* command.

See also *CMAKE_LINK_SEARCH_END_STATIC*.

CMAKE_LIST_FILE_NAME
Added in version 4.0.

The name of the CMake project files. This determines the top−level file processed when CMake is configured, and the file processed by *add_subdirectory()*.

By default, this is **CMakeLists.txt**. If set to anything else, **CMakeLists.txt** will be used as a fallback whenever the specified file cannot be found within a project subdirectory.

This variable reports the value set via the *cmake −−project−file* option. The value of this variable should never be set directly by projects or users.

WARNING:
The use of alternate project file names is intended for temporary use by developers during an incremental transition and not for publication of a final product. CMake will always emit a warning when the project file is anything other than **CMakeLists.txt**.

CMAKE_MAJOR_VERSION
First version number component of the *CMAKE_VERSION* variable.

CMAKE_MAKE_PROGRAM
Tool that can launch the native build system. The value may be the full path to an executable or just the tool name if it is expected to be in the **PATH**.

The tool selected depends on the *CMAKE_GENERATOR* used to configure the project:

- The *Makefile Generators* set this to **make**, **gmake**, or a generator−specific tool (e.g. **nmake** for *NMake Makefiles*).

  These generators store **CMAKE_MAKE_PROGRAM** in the CMake cache so that it may be edited by the user.

- The *Ninja* generator sets this to **ninja**.

  This generator stores **CMAKE_MAKE_PROGRAM** in the CMake cache so that it may be edited by the user.

- The *Xcode* generator sets this to **xcodebuild**.

  This generator prefers to lookup the build tool at build time rather than to store **CMAKE_MAKE_PROGRAM** in the CMake cache ahead of time. This is because **xcodebuild** is easy to find.

  For compatibility with versions of CMake prior to 3.2, if a user or project explicitly adds **CMAKE_MAKE_PROGRAM** to the CMake cache then CMake will use the specified value.

- The *Visual Studio Generators* set this to the full path to **MSBuild.exe** or **devenv.com**. (See also variables *CMAKE_VS_MSBUILD_COMMAND* and *CMAKE_VS_DEVENV_COMMAND*.

  These generators prefer to lookup the build tool at build time rather than to store **CMAKE_MAKE_PROGRAM** in the CMake cache ahead of time. This is because the tools are version−specific and can be located using the Visual Studio Installer. It is also necessary because the proper build tool may depend on the project content (e.g. the Intel Fortran plugin to Visual Studio requires **devenv.com** to build its **.vfproj** project files even though **MSBuild.exe** is normally preferred to support the *CMAKE_GENERATOR_TOOLSET*).

  For compatibility with versions of CMake prior to 3.0, if a user or project explicitly adds **CMAKE_MAKE_PROGRAM** to the CMake cache then CMake will use the specified value if possible.

- The *Green Hills MULTI* generator sets this to the full path to **gbuild.exe(Windows)** or **gbuild(Linux)** based upon the toolset being used.

  Once the generator has initialized a particular value for this variable, changing the value has undefined behavior.

The **CMAKE_MAKE_PROGRAM** variable is set for use by project code. The value is also used by the *cmake −−build* and *ctest −−build−and−test* tools to launch the native build process.

CMAKE_MATCH_COUNT
Added in version 3.2.

The number of matches with the last regular expression.

When a regular expression match is used, CMake fills in *CMAKE_MATCH_<n>* variables with the match contents. The**CMAKE_MA TCH_COUNT** variable holds the number of match expressions when these are filled.

CMAKE_MATCH_<n>
Capture group **<n>** matched by the last regular expression, for groups 0 through 9. Group 0 is the entire match. Groups 1 through 9 are the subexpressions captured by **()** syntax.

When a regular expression match is used, CMake fills in **CMAKE_MATCH_<n>** variables with the match contents. The*CMAKE_MA TCH_COUNT* variable holds the number of match expressions when these are filled.

CMAKE_MINIMUM_REQUIRED_VERSION
The **<min>** version of CMake given to the most recent call to the *cmake_minimum_required(VERSION)* command in the current variable scope or any parent variable scope.

CMAKE_MINOR_VERSION
Second version number component of the *CMAKE_VERSION* variable.

CMAKE_NETRC
Added in version 3.11.

This variable is used to initialize the **NETRC** option for the *file(DOWNLOAD)* and *file(UPLOAD)* commands. See those commands for additional information.

This variable is also used by the *ExternalProject* and *FetchContent* modules for internal calls to *file(DOWNLOAD)*.

The local option takes precedence over this variable.

CMAKE_NETRC_FILE
Added in version 3.11.

This variable is used to initialize the **NETRC_FILE** option for the *file(DOWNLOAD)* and *file(UPLOAD)* commands. See those commands for additional information.

This variable is also used by the *ExternalProject* and *FetchContent* modules for internal calls to *file(DOWNLOAD)*.

The local option takes precedence over this variable.

CMAKE_OBJDUMP
Path to the **objdump** executable on the host system. This tool, typically part of the Binutils collection on Unix−like systems, provides information about compiled object files.

This cache variable may be populated by CMake when project languages are enabled using the *project()* or *enable_language()* commands.

See Also

- The *file(GET_RUNTIME_DEPENDENCIES)* command provides a more general way to get information from runtime binaries.
- The *CPACK_OBJDUMP_EXECUTABLE* variable.

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
Third version number component of the *CMAKE_VERSION* variable.

CMAKE_PROJECT_DESCRIPTION
Added in version 3.9.

The description of the top level project.

This variable holds the description of the project as specified in the top level CMakeLists.txt file by a *project()* command. In the event that the top level CMakeLists.txt contains multiple *project()* calls, the most recently called one from that top level CMakeLists.txt will determine the value that **CMAKE_PROJECT_DESCRIPTION** contains. For example, consider the following top level CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(First DESCRIPTION "I am First")
project(Second DESCRIPTION "I am Second")
add_subdirectory(sub)
project(Third DESCRIPTION "I am Third")
```

And **sub/CMakeLists.txt** with the following contents:

```
project(SubProj DESCRIPTION "I am SubProj")
message("CMAKE_PROJECT_DESCRIPTION = ${CMAKE_PROJECT_DESCRIPTION}")
```

The most recently seen *project()* command from the top level CMakeLists.txt would be **project(Second ...)**, so this will print:

```
CMAKE_PROJECT_DESCRIPTION = I am Second
```

To obtain the description from the most recent call to *project()* in the current directory scope or above, see the *PROJECT_DESCRIPTION* variable.

CMAKE_PROJECT_HOMEPAGE_URL
Added in version 3.12.

The homepage URL of the top level project.

This variable holds the homepage URL of the project as specified in the top level CMakeLists.txt file by a *project()* command. In the event that the top level CMakeLists.txt contains multiple *project()* calls, the most recently called one from that top level CMakeLists.txt will determine the value that **CMAKE_PROJECT_HOMEPAGE_URL** contains. For example, consider the following top level CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(First HOMEPAGE_URL "https://first.example.com")
project(Second HOMEPAGE_URL "https://second.example.com")
add_subdirectory(sub)
project(Third HOMEPAGE_URL "https://third.example.com")
```

And **sub/CMakeLists.txt** with the following contents:

```
project(SubProj HOMEPAGE_URL "https://subproj.example.com")
message("CMAKE_PROJECT_HOMEPAGE_URL = ${CMAKE_PROJECT_HOMEPAGE_URL}")
```

The most recently seen *project()* command from the top level CMakeLists.txt would be **project(Second ...)**, so this will print:

```
CMAKE_PROJECT_HOMEPAGE_URL = https://second.example.com
```

To obtain the homepage URL from the most recent call to *project()* in the current directory scope or above, see the *PROJECT_HOMEPAGE_URL* variable.

CMAKE_PROJECT_NAME
The name of the top level project.

This variable holds the name of the project as specified in the top level CMakeLists.txt file by a *project()* command. In the event that the top level CMakeLists.txt contains multiple *project()* calls, the most recently called one from that top level CMakeLists.txt will determine the name that **CMAKE_PROJECT_NAME** contains. For example, consider the following top level CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(First)
project(Second)
add_subdirectory(sub)
project(Third)
```

And **sub/CMakeLists.txt** with the following contents:

```
project(SubProj)
message("CMAKE_PROJECT_NAME = ${CMAKE_PROJECT_NAME}")
```

The most recently seen *project()* command from the top level CMakeLists.txt would be **project(Second)**, so this will print:

```
CMAKE_PROJECT_NAME = Second
```

To obtain the name from the most recent call to *project()* in the current directory scope or above, see the *PROJECT_NAME* variable.

CMAKE_PROJECT_VERSION
Added in version 3.12.

The version of the top level project.

This variable holds the version of the project as specified in the top level CMakeLists.txt file by a *project()* command. In the event that the top level CMakeLists.txt contains multiple *project()* calls, the most recently called one from that top level CMakeLists.txt will determine the value that **CMAKE_PROJECT_VERSION** contains. For example, consider the following top level CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(First VERSION 1.2.3)
project(Second VERSION 3.4.5)
add_subdirectory(sub)
project(Third VERSION 6.7.8)
```

And **sub/CMakeLists.txt** with the following contents:

```
project(SubProj VERSION 1)
message("CMAKE_PROJECT_VERSION = ${CMAKE_PROJECT_VERSION}")
```

The most recently seen *project()* command from the top level CMakeLists.txt would be **project(Second ...)**, so this will print:

```
CMAKE_PROJECT_VERSION = 3.4.5
```

To obtain the version from the most recent call to *project()* in the current directory scope or above, see the *PROJECT_VERSION* variable.

CMAKE_PROJECT_VERSION_MAJOR
Added in version 3.12.

The major version of the top level project.

This variable holds the major version of the project as specified in the top level CMakeLists.txt file by a *project()* command. Please see *CMAKE_PROJECT_VERSION* documentation for the behavior when multiple *project()* commands are used in the sources.

CMAKE_PROJECT_VERSION_MINOR
Added in version 3.12.

The minor version of the top level project.

This variable holds the minor version of the project as specified in the top level CMakeLists.txt file by a *project()* command. Please see *CMAKE_PROJECT_VERSION* documentation for the behavior when multiple *project()* commands are used in the sources.

CMAKE_PROJECT_VERSION_PATCH
Added in version 3.12.

The patch version of the top level project.

This variable holds the patch version of the project as specified in the top level CMakeLists.txt file by a *project()* command. Please see *CMAKE_PROJECT_VERSION* documentation for the behavior when multiple *project()* commands are used in the sources.

CMAKE_PROJECT_VERSION_TWEAK
Added in version 3.12.

The tweak version of the top level project.

This variable holds the tweak version of the project as specified in the top level CMakeLists.txt file by a *project()* command. Please see *CMAKE_PROJECT_VERSION* documentation for the behavior when

multiple *project()* commands are used in the sources.

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_RULE_MESSAGES
Added in version 3.13.

Specify whether to report a message for each make rule.

If set in the cache it is used to initialize the value of the *RULE_MESSAGES* property. Users may disable the option in their local build tree to disable granular messages and report only as each target completes in Makefile builds.

CMAKE_SCRIPT_MODE_FILE
Full path to the *cmake −P* script file currently being processed.

When run in *cmake −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.

See Also
- The *CMAKE_ROLE* global property provides the current running mode.

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_LIBRARY_ARCHIVE_SUFFIX
Added in version 3.31.

The suffix for archived shared libraries that you link to.

The suffix to use for the end of a archive containing a shared library, **.a** on AIX.

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 target machine, and is determined when a compiled language is enabled. If a 64−bit size is found, then the library search path is modified to look for 64−bit libraries first.

CMAKE_SKIP_INSTALL_RULES
Whether to disable generation of installation rules.

If **TRUE**, CMake will neither generate installation rules nor will it generate **cmake_install.cmake** files. This variable is **FALSE** by default.

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. To omit RPATH in the build step, use *CMAKE_SKIP_BUILD_RPATH*.

For more information on RPATH handling see the *INSTALL_RPATH* and *BUILD_RPATH* target properties.

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*.

When run in *cmake −P* script mode, CMake sets the variables *CMAKE_BINARY_DIR*, **CMAKE_SOURCE_DIR**, *CMAKE_CURRENT_BINARY_DIR* and *CMAKE_CURRENT_SOURCE_DIR* to the current working directory.

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_Swift_COMPILATION_MODE
Added in version 3.29.

Specify how Swift compiles a target. This variable is used to initialize the *Swift_COMPILATION_MODE* property on targets as they are created.

The allowed values are:

incremental
Compiles each Swift source in the module separately, resulting in better parallelism in the build. The compiler emits additional information into the build directory improving rebuild performance when small changes are made to the source between rebuilds. This is the best option to use while iterating on changes in a project.

wholemodule
Whole−module optimizations are slowest to compile, but results in the most optimized library. The entire context is loaded into once instance of the compiler, so there is no parallelism across source files in the module.

singlefile
Compiles each source in a Swift modules separately, resulting in better parallelism. Unlike the **incremental** build mode, no additional information is emitted by the compiler during the build, so rebuilding after making small changes to the source file will not run faster. This option should be used sparingly, preferring **incremental** builds, unless working around a compiler bug.

Use *generator expressions* to support per−configuration specification. For example, the code:

```
set(CMAKE_Swift_COMPILATION_MODE
  "$<IF:$<CONFIG:Release>,wholemodule,incremental>")
```

sets the default Swift compilation mode to wholemodule mode when building a release configuration and to incremental mode in other configurations.

If this variable is not set then the *Swift_COMPILATION_MODE* target property will not be set automatically. If that property is unset then CMake uses the default value **incremental** to build the Swift source files.

NOTE:
This property only has effect when policy *CMP0157* is set to **NEW** prior to the first *project()* or *enable_language()* command that enables the Swift language.

CMAKE_Swift_MODULE_DIRECTORY
Added in version 3.15.

Swift module output directory.

This variable is used to initialize the *Swift_MODULE_DIRECTORY* property on all the targets. See the target property for additional information.

CMAKE_Swift_NUM_THREADS
Added in version 3.15.1.

Number of threads for parallel compilation for Swift targets.

This variable controls the number of parallel jobs that the swift driver creates for building targets. If not specified, it will default to the number of logical CPUs on the host.

CMAKE_TEST_LAUNCHER
Added in version 3.29.

This variable is used to initialize the *TEST_LAUNCHER* target property of executable targets as they are created. It is used to specify a launcher for running tests, added by the*add_test()* command, that run an e xecutable target.

If this variable contains a *semicolon−separated list*, then the first value is the command and remaining values are its arguments.

This variable can be initialized via an *CMAKE_TEST_LAUNCHER* environment variable.

CMAKE_TOOLCHAIN_FILE
Path to toolchain file supplied to *cmake(1)*.

This variable is specified on the command line when cross−compiling with CMake. It is the path to a file which is read early in the CMake run and which specifies locations for compilers and toolchain utilities, and other target platform and compiler related information.

Relative paths are allowed and are interpreted first as relative to the build directory, and if not found, relative to the source directory.

This is initialized by the *CMAKE_TOOLCHAIN_FILE* environment variable if it is set when a new build tree is first created.

See the *CMAKE_PROJECT_TOP_LEVEL_INCLUDES* variable for setting other things not directly related to the toolchain.

CMAKE_TWEAK_VERSION
Defined to **0** for compatibility with code written for older CMake versions that may have defined higher values.

NOTE:
In CMake versions 2.8.2 through 2.8.12, this variable holds the fourth version number component of the *CMAKE_VERSION* variable.

CMAKE_VERBOSE_MAKEFILE
Enable verbose output from Makefile builds.

This variable is a cache entry initialized (to **FALSE**) by the *project()* command. Users may enable the option in their local build tree to get more verbose output from Makefile builds and show each command line as it is launched.

CMAKE_VERSION
The CMake version string as three non−negative integer components separated by **.** and possibly followed by **−** and other information. The first two components represent the feature level and the third component represents either a bug−fix level or development date.

Release versions and release candidate versions of CMake use the format:

```
<major>.<minor>.<patch>[-rc<n>]
```

where the <**patch**> component is less than **20000000**. Development versions of CMake use the format:

```
<major>.<minor>.<date>[-<id>]
```

where the **<date>** component is of format **CCYYMMDD** and **<id>** may contain arbitrary text. This represents development as of a particular date following the **<major>.<minor>** feature release.

Individual component values are also available in variables:

- *CMAKE_MAJOR_VERSION*
- *CMAKE_MINOR_VERSION*
- *CMAKE_PATCH_VERSION*
- *CMAKE_TWEAK_VERSION*

Use the *if()* command **VERSION_LESS**, **VERSION_GREATER**, **VERSION_EQUAL**, **VERSION_LESS_EQUAL**, or **VERSION_GREATER_EQUAL** operators to compare version string values against **CMAKE_VERSION** using a component−wise test. Version component values may be 10 or larger so do not attempt to compare version strings as floating−point numbers.

NOTE:

CMake versions 2.8.2 through 2.8.12 used three components for the feature level. Release versions represented the bug−fix level in a fourth component, i.e. **<major>.<minor>.<patch>[.<tweak>][−rc<n>]**. Development versions represented the development date in the fourth component, i.e. **<major>.<minor>.<patch>.<date>[−<id>]**.

CMake versions prior to 2.8.2 used three components for the feature level and had no bug−fix component. Release versions used an even−valued second component, i.e. **<major>.<e ven−minor>.<patch>[−rc<n>]**. Development versions used an odd−valued second component with the development date as the third component, i.e. **<major>.<odd−minor>.<date>**.

The **CMAKE_VERSION** variable is defined by CMake 2.6.3 and higher. Earlier versions defined only the individual component variables.

CMAKE_VS_DEVENV_COMMAND

The *Visual Studio Generators* set this variable to the **devenv.com** command installed with the corresponding Visual Studio version.

This variable is not defined by other generators even if **devenv.com** is installed on the computer.

See also the *CMAKE_VS_MSBUILD_COMMAND* and *CMAKE_MAKE_PROGRAM* variables.

CMAKE_VS_MSBUILD_COMMAND

The *Visual Studio Generators* set this variable to the **MSBuild.exe** command installed with the corresponding Visual Studio version.

This variable is not defined by other generators even if **MSBuild.exe** is installed on the computer.

See also the *CMAKE_VS_DEVENV_COMMAND* and *CMAKE_MAKE_PROGRAM* variables.

CMAKE_VS_NsightTegra_VERSION

Added in version 3.1.

When using a Visual Studio generator with the *CMAKE_SYSTEM_NAME* variable set to **Android**, this variable contains the version number of the installed NVIDIA Nsight Tegra Visual Studio Edition.

CMAKE_VS_NUGET_PACKAGE_RESTORE

Added in version 3.23.

When using a Visual Studio generator, this cache variable controls if msbuild should automatically attempt to restore NuGet packages prior to a build. NuGet packages can be defined using the *VS_PACKAGE_REFERENCES* property on a target. If no package references are defined, this setting will do nothing.

The command line option *−−resolve−package−references* can be used alternatively to control the resolve behavior globally. This option will take precedence over the cache variable.

Targets that use the *DOTNET_SDK* are required to run a restore before building. Disabling this option may cause the build to fail in such projects.

This setting is stored as a cache entry. Default value is **ON**.

See also the *VS_PACKAGE_REFERENCES* property.

CMAKE_VS_PLATFORM_NAME
Added in version 3.1.

Visual Studio target platform name used by the current generator.

VS 8 and above allow project files to specify a target platform. CMake provides the name of the chosen platform in this variable. See the*CMAKE_GENERA TOR_PLATFORM* variable for details.

See also the *CMAKE_VS_PLATFORM_NAME_DEFAULT* variable.

CMAKE_VS_PLATFORM_NAME_DEFAULT
Added in version 3.14.3.

Default for the Visual Studio target platform name for the current generator without considering the value of the *CMAKE_GENERATOR_PLATFORM* variable. For *Visual Studio Generators* for VS 2017 and below this is always **Win32**. For VS 2019 and above this is based on the host platform.

See also the *CMAKE_VS_PLATFORM_NAME* variable.

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.

See the *CMAKE_GENERATOR_TOOLSET* variable for details.

CMAKE_VS_PLATFORM_TOOLSET_CUDA
Added in version 3.9.

NVIDIA CUDA Toolkit version whose Visual Studio toolset to use.

The *Visual Studio Generators* for VS 2010 and above support using a CUDA toolset provided by a CUDA Toolkit. The toolset version number may be specified by a field in *CMAKE_GENERATOR_TOOLSET* of the form **cuda=8.0**. Or it is automatically detected if a path to a standalone CUDA directory is specified in the form **cuda=C:\path\to\cuda**. If none is specified CMake will choose a default version. CMake provides the selected CUDA toolset version in this variable. The value may be empty if no CUDA Toolkit with Visual Studio integration is installed.

CMAKE_VS_PLATFORM_TOOLSET_CUDA_CUSTOM_DIR
Added in version 3.16.

Path to standalone NVIDIA CUDA Toolkit (eg. extracted from installer).

The *Visual Studio Generators* for VS 2010 and above support using a standalone (non−installed) NVIDIA CUDA toolkit. The path may be specified by a field in *CMAKE_GENERATOR_TOOLSET* of the form **cuda=C:\path\to\cuda**. The given directory must at least contain the nvcc compiler in path **.\bin** and must provide Visual Studio integration files in path **.\extras\visual_studio_integration\ MSBuildExtensions**. One can create a standalone CUDA toolkit directory by either opening a installer with 7zip or copying the files that are extracted by the running installer. The value may be empty if no path to a standalone CUDA Toolkit was specified.

CMAKE_VS_PLATFORM_TOOLSET_FORTRAN
Added in version 3.29.

Fortran compiler to be used by Visual Studio projects.

Visual Studio Generators support selecting among Fortran compilers that have the required Visual Studio Integration feature installed. The compiler may be specified by a field in *CMAKE_GENERATOR_TOOLSET* of the form **fortran=...**. CMake provides the selected Fortran compiler in this variable. The value may be empty if the field was not specified.

CMAKE_VS_PLATFORM_TOOLSET_HOST_ARCHITECTURE
Added in version 3.8.

Visual Studio preferred tool architecture.

The *Visual Studio Generators* for VS 2013 and above support using either the 32−bit or 64−bit host toolchains by specifying a **host=x86** or **host=x64** value in the *CMAKE_GENERATOR_TOOLSET* option. CMake provides the selected toolchain architecture preference in this variable (**x86**, **x64**, **ARM64** or empty).

CMAKE_VS_PLATFORM_TOOLSET_VERSION
Added in version 3.12.

Visual Studio Platform Toolset version.

The *Visual Studio Generators* for VS 2017 and above allow to select minor versions of the same toolset. The toolset version number may be specified by a field in *CMAKE_GENERATOR_TOOLSET* of the form **version=14.11**. If none is specified CMake will choose a default toolset. The value may be empty if no minor version was selected and the default is used.

If the value is not empty, it is the version number that MSBuild uses in its **Microsoft.VCToolsVersion.*.props** file names.

Added in version 3.19.7: VS 16.9's toolset may also be specified as **14.28.16.9** because VS 16.10 uses the file name **Microsoft.VCToolsVersion.14.28.16.9.props**.

Three−Component MSVC Toolset Versions
Added in version 3.19.7.

The **version=** field may be given a three−component toolset version such as **14.28.29910**, and CMake will convert it to the name used by MSBuild **Microsoft.VCToolsVersion.*.props** files. This is useful to distinguish between VS 16.8's **14.28.29333** toolset and VS 16.9's **14.28.29910** toolset. It also matches **vcvarsall**'s **−vcvars_ver=** behavior.

CMAKE_VS_TARGET_FRAMEWORK_IDENTIFIER
Added in version 3.22.

Visual Studio target framework identifier.

In some cases, the *Visual Studio Generators* may use an explicit value for the MSBuild **TargetFrameworkIdentifier** setting in **.csproj** files. CMake provides the chosen value in this variable.

See also *CMAKE_VS_TARGET_FRAMEWORK_VERSION* and *CMAKE_VS_TARGET_FRAMEWORK_TARGETS_VERSION*.

CMAKE_VS_TARGET_FRAMEWORK_TARGETS_VERSION
Added in version 3.22.

Visual Studio target framework targets version.

In some cases, the *Visual Studio Generators* may use an explicit value for the MSBuild **TargetFrameworkTargetsVersion** setting in **.csproj** files. CMake provides the chosen value in this variable.

See also *CMAKE_VS_TARGET_FRAMEWORK_VERSION* and *CMAKE_VS_TARGET_FRAMEWORK_IDENTIFIER*.

CMAKE_VS_TARGET_FRAMEWORK_VERSION
Added in version 3.22.

Visual Studio target framework version.

In some cases, the *Visual Studio Generators* may use an explicit value for the MSBuild **TargetFrameworkVersion** setting in **.csproj** files. CMake provides the chosen value in this variable.

See the *CMAKE_DOTNET_TARGET_FRAMEWORK_VERSION* variable and *DOTNET_TARGET_FRAMEWORK_VERSION* target property to specify custom **TargetFrameworkVersion** values for project targets.

See also *CMAKE_VS_TARGET_FRAMEWORK_IDENTIFIER* and *CMAKE_VS_TARGET_FRAMEWORK_TARGETS_VERSION*.

CMAKE_VS_USE_DEBUG_LIBRARIES
Added in version 3.30.

Indicate to *Visual Studio Generators* what configurations are considered debug configurations. This controls the **UseDebugLibraries** setting in each configuration of a **.vcxproj** file.

The "Use Debug Libraries" setting in Visual Studio projects, despite its specific−sounding name, is a general−purpose indicator of what configurations are considered debug configurations. In standalone projects, this may affect MSBuild's default selection of MSVC runtime library, optimization flags, runtime checks, and similar settings. In CMake projects those settings are typically generated explicitly based on the project's specification, e.g., the MSVC runtime library is controlled by

CMAKE_MSVC_RUNTIME_LIBRARY. However, the **UseDebugLibraries** indicator is useful for reference by both humans and tools, and may also affect the behavior of platform−specific SDKs.

Set **CMAKE_VS_USE_DEBUG_LIBRARIES** to a true or false value to indicate whether each configuration is considered a debug configuration. The value may also be the empty string (**""**) in which case no **UseDebugLibraries** will be added explicitly by CMake, and MSBuild will use its default value, **false**.

Use *generator expressions* for per−configuration specification. For example, the code:

```
set(CMAKE_VS_USE_DEBUG_LIBRARIES "$<CONFIG:Debug,Custom>")
```

indicates that all following targets consider their "Debug" and "Custom" configurations to be debug configurations, and their other configurations to be non−debug configurations.

This variable is used to initialize the *VS_USE_DEBUG_LIBRARIES* property on all targets as they are created. It is also propagated by calls to the *try_compile()* command into its test project.

If this variable is not set then the *VS_USE_DEBUG_LIBRARIES* property will not be set automatically. If that property is not set then CMake generates **UseDebugLibraries** using heuristics to determine which configurations are debug configurations. See policy *CMP0162*.

CMAKE_VS_VERSION_BUILD_NUMBER
Added in version 3.26.

Visual Studio version.

Visual Studio Generators for VS 2017 and above set this variable to the Visual Studio version build number in the format **<major>.<minor>.<date>.<build>**.

The components are:

<major>.<minor>
The VS major and minor version numbers. These are the same as the release version numbers.

<date>
A build date in the format **MMMDD**, where **MMM** is a month index since an epoch used by Microsoft, and **DD** is a day in that month.

<build>
A build index on the day represented by **<date>**.

The build number is reported by **vswhere** as **installationVersion**. For example, VS 16.11.10 has build number **16.11.32126.315**.

See also the *CMAKE_GENERATOR_INSTANCE* variable.

CMAKE_VS_WINDOWS_TARGET_PLATFORM_MIN_VERSION
Added in version 3.27.

Tell *Visual Studio Generators* to use the given Windows Target Platform Minimum Version.

This variable is used to initialize the *VS_WINDOWS_TARGET_PLATFORM_MIN_VERSION* property on all targets when they are created. See that target property for additional information.

CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION
Added in version 3.4.

Visual Studio Windows Target Platform Version.

When targeting Windows 10 and above, *Visual Studio Generators* for VS 2015 and above support specification of a Windows SDK version:

- If *CMAKE_GENERATOR_PLATFORM* specifies a **version=** field, as documented by *Visual Studio Platform Selection*, that SDK version is selected.
- Otherwise, if the **WindowsSDKVersion** environment variable is set to an available SDK version, that version is selected. This is intended for use in environments established by **vcvarsall.bat** or similar scripts.

  Added in version 3.27: This is enabled by policy *CMP0149*.

- Otherwise, if *CMAKE_SYSTEM_VERSION* is set to an available SDK version, that version is selected.

  Changed in version 3.27: This is disabled by policy *CMP0149*.

- Otherwise, CMake uses the latest Windows SDK version available.

The chosen Windows target version number is provided in **CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION**. If no Windows 10 SDK is available this value will be empty.

One may set a **CMAKE_WINDOWS_KITS_10_DIR** *environment variable* to an absolute path to tell CMake to look for Windows 10 SDKs in a custom location. The specified directory is expected to contain **Include/10.0.*** directories.

See also *CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION_MAXIMUM*.

CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION_MAXIMUM
Added in version 3.19.

Override the *Windows 10 SDK Maximum Version for VS 2015* and beyond.

The **CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION_MAXIMUM** variable may be set to a false value (e.g. **OFF**, **FALSE**, or **0**) or the SDK version to use as the maximum (e.g. **10.0.14393.0**). If unset, the default depends on which version of Visual Studio is targeted by the current generator.

This can be used to exclude Windows SDK versions from consideration for *CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION*.

CMAKE_WINDOWS_KMDF_VERSION
Added in version 3.31.

Specify the *Kernel−Mode Drive Framework* target version.

A *toolchain file* that sets *CMAKE_SYSTEM_NAME* to **WindowsKernelModeDriver** must also set **CMAKE_WINDOWS_KMDF_VERSION** to specify the KMDF target version.

CMAKE_XCODE_BUILD_SYSTEM
Added in version 3.19.

Xcode build system selection.

The *Xcode* generator defines this variable to indicate which variant of the Xcode build system will be used. The value is the version of Xcode in which the corresponding build system first became mature enough for use by CMake. The possible values are:

1 The original Xcode build system. This is the default when using Xcode 11.x or below and supported up to Xcode 13.x.

12 The Xcode "new build system" introduced by Xcode 10. It became mature enough for use by CMake in Xcode 12. This is the default when using Xcode 12.x or above.

The **CMAKE_XCODE_BUILD_SYSTEM** variable is informational and should not be modified by project code. See the *Toolset and Build System Selection* documentation section to select the Xcode build system.

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−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*add_subdir ectory()* is used to connect several projects.

<PROJECT−NAME>_DESCRIPTION
Added in version 3.12.

Value given to the **DESCRIPTION** option of the most recent call to the *project()* command with project name **<PROJECT−NAME>**, if any.

<PROJECT−NAME>_HOMEPAGE_URL
Added in version 3.12.

Value given to the **HOMEPAGE_URL** option of the most recent call to the *project()* command with project name **<PROJECT−NAME>**, if any.

<PROJECT−NAME>_IS_TOP_LEVEL
Added in version 3.21.

A boolean variable indicating whether the named project was called in a top level **CMakeLists.txt** file.

To obtain the value from the most recent call to *project()* in the current directory scope or above, see the *PROJECT_IS_TOP_LEVEL* variable.

The variable value will be true in:

- the top−level directory of the project
- the top−level directory of an external project added by *ExternalProject*
- a directory added by *add_subdirectory()* that does not also contain a *project()* call
- a directory added by *FetchContent_MakeAvailable()*, if the fetched content does not contain a *project()* call

The variable value will be false in:

- a directory added by *add_subdirectory()* that also contains a *project()* call
- a directory added by *FetchContent_MakeAvailable()*, if the fetched content contains a *project()* call

<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_subdir ectory()* is used to connect several projects.

<PROJECT−NAME>_VERSION
Value given to the **VERSION** option of the most recent call to the *project()* command with project name **<PROJECT−NAME>**, if any.

See also the component−wise version variables *<PROJECT−NAME>_VERSION_MAJOR*, *<PROJECT−NAME>_VERSION_MINOR*, *<PROJECT−NAME>_VERSION_PATCH*, and *<PROJECT−NAME>_VERSION_TWEAK*.

<PROJECT−NAME>_VERSION_MAJOR
First version number component of the *<PROJECT−NAME>_VERSION* variable as set by the *project()* command.

<PROJECT−NAME>_VERSION_MINOR
Second version number component of the *<PROJECT−NAME>_VERSION* variable as set by the *project()* command.

<PROJECT−NAME>_VERSION_PATCH
Third version number component of the *<PROJECT−NAME>_VERSION* variable as set by the *project()* command.

<PROJECT−NAME>_VERSION_TWEAK
Fourth version number component of the *<PROJECT−NAME>_VERSION* variable as set by the *project()* command.

PROJECT_BINARY_DIR
Full path to build directory for project.

This is the binary directory of the most recent *project()* command.

PROJECT_DESCRIPTION
Added in version 3.9.

Short project description given to the project command.

This is the description given to the most recently called *project()* command in the current directory scope or above. To obtain the description of the top level project, see the *CMAKE_PROJECT_DESCRIPTION* variable.

PROJECT_HOMEPAGE_URL
Added in version 3.12.

The homepage URL of the project.

This is the homepage URL given to the most recently called *project()* command in the current directory scope or above. To obtain the homepage URL of the top level project, see the *CMAKE_PROJECT_HOMEPAGE_URL* variable.

PROJECT_IS_TOP_LEVEL
Added in version 3.21.

A boolean variable indicating whether the most recently called *project()* command in the current scope or above was in the top level **CMakeLists.txt** file.

Some modules should only be included as part of the top level **CMakeLists.txt** file to not cause unintended side effects in the build tree, and this variable can be used to conditionally execute such code. For example, consider the *CTest* module, which creates targets and options:

```
project(MyProject)
...
if(PROJECT_IS_TOP_LEVEL)
  include(CTest)
endif()
```

The variable value will be true in:

- the top−level directory of the project
- the top−level directory of an external project added by *ExternalProject*
- a directory added by *add_subdirectory()* that does not also contain a *project()* call
- a directory added by *FetchContent_MakeAvailable()*, if the fetched content does not contain a *project()* call

The variable value will be false in:

- a directory added by *add_subdirectory()* that also contains a *project()* call
- a directory added by *FetchContent_MakeAvailable()*, if the fetched content contains a *project()* call

PROJECT_NAME
Name of the project given to the project command.

This is the name given to the most recently called *project()* command in the current directory scope or above. To obtain the name of the top level project, see the *CMAKE_PROJECT_NAME* variable.

PROJECT_SOURCE_DIR
This is the source directory of the last call to the *project()* command made in the current directory scope or one of its parents. Note, it is not affected by calls to *project()* made within a child directory scope (i.e. from within a call to *add_subdirectory()* from the current scope).

PROJECT_VERSION
Value given to the **VERSION** option of the most recent call to the *project()* command, if any.

See also the component−wise version variables *PROJECT_VERSION_MAJOR*, *PROJECT_VERSION_MINOR*, *PROJECT_VERSION_PATCH*, and *PROJECT_VERSION_TWEAK*.

PROJECT_VERSION_MAJOR
First version number component of the *PROJECT_VERSION* variable as set by the *project()* command.

PROJECT_VERSION_MINOR
Second version number component of the *PROJECT_VERSION* variable as set by the *project()* command.

PROJECT_VERSION_PATCH
Third version number component of the *PROJECT_VERSION* variable as set by the *project()* command.

PROJECT_VERSION_TWEAK
Fourth version number component of the *PROJECT_VERSION* variable as set by the *project()* command.

VARIABLES THAT CHANGE BEHAVIOR

BUILD_SHARED_LIBS
Tell *add_library()* to default to **SHARED** libraries, instead of **STATIC** libraries, when called with no explicit library type.

Calls to *add_library()* without any explicit library type check the current **BUILD_SHARED_LIBS** variable value. If it is true, then the default library type is **SHARED**. Otherwise, the default is **STATIC**.

For example, the code:

```
add_library(example ${sources})
```

behaves as if written

```
if(BUILD_SHARED_LIBS)
  add_library(example SHARED ${sources})
else()
  add_library(example STATIC ${sources})
endif()
```

CMake does not define **BUILD_SHARED_LIBS** by default, but projects often create a cache entry for it using the *option()* command:

```
option(BUILD_SHARED_LIBS "Build using shared libraries" ON)
```

This provides a switch that users can control, e.g., with *cmake −D*. If adding such an option to the project, do so in the top level **CMakeLists.txt** file, before any *add_library()* calls. Note that if bringing external dependencies directly into the build, such as with *FetchContent* or a direct call to *add_subdirectory()*, and one of those dependencies has such a call to *option(BUILD_SHARED_LIBS ...)*, the top level project must also call *option(BUILD_SHARED_LIBS ...)* before bringing in its dependencies. Failure to do so can lead to different behavior between the first and subsequent CMake runs.

BUILD_TESTING
Control whether the *CTest* module invokes *enable_testing()*.

The *CTest* module, when loaded by **include(CTest)**, runs code of the form:

```
option(BUILD_TESTING "..." ON)
if (BUILD_TESTING)
   # ...
   enable_testing()
   # ...
endif()
```

This creates a **BUILD_TESTING** option that controls whether the *enable_testing()* command is invoked to enable generation of tests to run using *ctest(1)*. See the*add_test()* command to create tests.

NOTE:

Call **include(CTest)** in the top−level source directory since *ctest(1)* expects to find a test file in the top−level build directory.

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_ADD_CUSTOM_COMMAND_DEPENDS_EXPLICIT_ONLY
Added in version 3.27.

Whether to enable the **DEPENDS_EXPLICIT_ONLY** option by default in *add_custom_command()*.

This variable affects the default behavior of the *add_custom_command()* command. Setting this variable to **ON** is equivalent to using the **DEPENDS_EXPLICIT_ONLY** option in all uses of that command.

See also *CMAKE_OPTIMIZE_DEPENDENCIES*.

CMAKE_APPBUNDLE_PATH
Semicolon−separated list of directories specifying a search path for macOS application bundles used by the *find_program()*, and *find_package()* commands.

There is also an environment variable *CMAKE_APPBUNDLE_PATH*, which is used as an additional list of search directories.

CMAKE_BUILD_TYPE
Specifies the build type on single−configuration generators (e.g. *Mak efile Generators* or *Ninja*). Typical values include **Debug**, **Release**, **RelWithDebInfo** and **MinSizeRel**, but custom build types can also be defined.

This variable is initialized by the first *project()* or *enable_language()* command called in a project when a new build tree is first created. If the *CMAKE_BUILD_TYPE* environment variable is set, its value is used. Otherwise, a toolchain−specific default is chosen when a language is enabled. The default value is often an empty string, but this is usually not desirable and one of the other standard build types is usually more appropriate.

Depending on the situation, the value of this variable may be treated case−sensitively or case−insensitively. See *Build Configurations* for discussion of this and other related topics.

For multi−config generators, see *CMAKE_CONFIGURATION_TYPES*.

CMAKE_CLANG_VFS_OVERLAY
Added in version 3.19.

When cross compiling for windows with clang−cl, this variable can be an absolute path pointing to a clang virtual file system yaml file, which will enable clang−cl to resolve windows header names on a case sensitive file system.

CMAKE_CODEBLOCKS_COMPILER_ID
Added in version 3.11.

Change the compiler id in the generated CodeBlocks project files.

CodeBlocks uses its own compiler id string which differs from *CMAKE_<LANG>_COMPILER_ID*. If

this variable is left empty, CMake tries to recognize the CodeBlocks compiler id automatically. Otherwise the specified string is used in the CodeBlocks project file. See the CodeBlocks documentation for valid compiler id strings.

Other IDEs like QtCreator that also use the CodeBlocks generator may ignore this setting.

CMAKE_CODEBLOCKS_EXCLUDE_EXTERNAL_FILES
Added in version 3.10.

Change the way the CodeBlocks generator creates project files.

If this variable evaluates to **ON** the generator excludes from the project file any files that are located outside the project root.

CMAKE_CODELITE_USE_TARGETS
Added in version 3.7.

Change the way the CodeLite generator creates projectfiles.

If this variable evaluates to **ON** at the end of the top−level **CMakeLists.txt** file, the generator creates projectfiles based on targets rather than projects.

CMAKE_COLOR_DIAGNOSTICS
Added in version 3.24.

Enable color diagnostics throughout the generated build system.

This variable uses three states: **ON**, **OFF** and not defined.

When not defined:

- *Makefile Generators* initialize the *CMAKE_COLOR_MAKEFILE* variable to **ON**. It controls color buildsystem messages.
- GNU/Clang compilers are not invoked with any color diagnostics flag.

When **ON**:

- *Makefile Generators* produce color buildsystem messages by default. *CMAKE_COLOR_MAKEFILE* is not initialized, but may be explicitly set to **OFF** to disable color buildsystem messages.
- GNU/Clang compilers are invoked with a flag enabling color diagnostics (**−fcolor−diagnostics**).

When **OFF**:

- *Makefile Generators* do not produce color buildsystem messages by default. *CMAKE_COLOR_MAKEFILE* is not initialized, but may be explicitly set to **ON** to enable color buildsystem messages.
- GNU/Clang compilers are invoked with a flag disabling color diagnostics (**−fno−color−diagnostics**).

If the *CMAKE_COLOR_DIAGNOSTICS* environment variable is set, its value is used. Otherwise, **CMAKE_COLOR_DIAGNOSTICS** is not defined by default.

See the *CLICOLOR* and *CLICOLOR_FORCE* environment variables to control color output from CMake command−line tools.

CMAKE_COLOR_MAKEFILE
Enables color output when using the *Makefile Generators*.

When enabled, the generated Makefiles will produce colored output. Default is **ON**.

CMAKE_CONFIGURATION_TYPES
Specifies the available build types (configurations) on multi−config generators (e.g. *Visual Studio*, *Xcode*, or *Ninja Multi−Config*) as a *semicolon−separated list*. Typical entries include **Debug**, **Release**, **RelWithDebInfo** and **MinSizeRel**, but custom build types can also be defined.

This variable is initialized by the first *project()* or *enable_language()* command called in a project when a new build tree is first created. If the *CMAKE_CONFIGURATION_TYPES* environment variable is set, its value is used. Otherwise, the default value is generator−specific.

Depending on the situation, the values in this variable may be treated case−sensitively or case−insensitively. See*Build Configur ations* for discussion of this and other related topics.

For single−config generators, see *CMAKE_BUILD_TYPE*.

CMAKE_DEPENDS_IN_PROJECT_ONLY
Added in version 3.6.

When set to **TRUE** in a directory, the build system produced by the *Makefile Generators* is set up to only consider dependencies on source files that appear either in the source or in the binary directories. Changes to source files outside of these directories will not cause rebuilds.

This should be used carefully in cases where some source files are picked up through external headers during the build.

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*.

Note that this variable can lead to inconsistent results within the project. Consider the case where a dependency is requested via *find_package()* from two different places within the project. If the first call does not have the **REQUIRED** keyword, it will not find the dependency when **CMAKE_DISABLE_FIND_PACKAGE_<PackageName>** is set to true for that dependency. The project will proceed under the assumption that the dependency isn't available. If the second call elsewhere in the project *does* have the **REQUIRED** keyword, it can succeed. Two different parts of the same project have then seen opposite results for the same dependency.

See also the *CMAKE_REQUIRE_FIND_PACKAGE_<PackageName>* variable.

CMAKE_ECLIPSE_GENERATE_LINKED_RESOURCES
Added in version 3.6.

This cache variable is used by the Eclipse project generator. See*cmak e−generators(7)*.

The Eclipse project generator generates so−called linked resources e.g. to the subproject root dirs in the source tree or to the source files of targets. This can be disabled by setting this variable to FALSE.

CMAKE_ECLIPSE_GENERATE_SOURCE_PROJECT
Added in version 3.6.

This cache variable is used by the Eclipse project generator. See*cmak e−generators(7)*.

If this variable is set to TRUE, the Eclipse project generator will generate an Eclipse project in *CMAKE_SOURCE_DIR* . This project can then be used in Eclipse e.g. for the version control functionality. **CMAKE_ECLIPSE_GENERATE_SOURCE_PROJECT** defaults to **FALSE**; so nothing is written into the source directory.

CMAKE_ECLIPSE_MAKE_ARGUMENTS
Added in version 3.6.

This cache variable is used by the Eclipse project generator. See*cmak e−generators(7)*.

This variable holds arguments which are used when Eclipse invokes the make tool. By default it is initialized to hold flags to enable parallel builds (using −j typically).

CMAKE_ECLIPSE_RESOURCE_ENCODING
Added in version 3.16.

This cache variable tells the *Eclipse CDT4* project generator to set the resource encoding to the given value in generated project files. If no value is given, no encoding will be set.

CMAKE_ECLIPSE_VERSION
Added in version 3.6.

This cache variable is used by the Eclipse project generator. See*cmak e−generators(7)*.

When using the Eclipse project generator, CMake tries to find the Eclipse executable and detect the version of it. Depending on the version it finds, some features are enabled or disabled. If CMake doesn't find Eclipse, it assumes the oldest supported version, Eclipse Callisto (3.2).

CMAKE_ERROR_DEPRECATED
Whether to issue errors for deprecated functionality.

If **TRUE**, use of deprecated functionality will issue fatal errors. If this variable is not set, CMake behaves as if it were set to **FALSE**.

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_EXECUTE_PROCESS_COMMAND_ECHO
Added in version 3.15.

If this variable is set to **STDERR**, **STDOUT** or **NONE** then commands in *execute_process()* calls will be

printed to either stderr or stdout or not at all.

CMAKE_EXECUTE_PROCESS_COMMAND_ERROR_IS_FATAL
Added in version 4.0.

Specify a default for the *execute_process()* command's **COMMAND_ERROR_IS_FATAL** option. This variable is ignored when a **RESULTS_VARIABLE** or **RESULT_VARIABLE** keyword is supplied to the command.

CMAKE_EXPORT_BUILD_DATABASE
Added in version 3.31.

NOTE:
This variable is meaningful only when experimental support for build databases has been enabled by the **CMAKE_EXPERIMENTAL_EXPORT_BUILD_DATABASE** gate.

Enable/Disable output of module compile commands during the build.

If enabled, generates a **build_database.json** file containing the information necessary to compile a target's C++ module sources with any tooling. The format of the JSON file looks like:

```
{
  "version": 1,
  "revision": 0,
  "sets": [
    {
      "family-name" : "export_build_database",
      "name" : "export_build_database@Debug",
      "translation-units" : [
        {
          "arguments": [
            "/path/to/compiler",
            "...",
          ],
          "baseline-arguments" :
          [
            "...",
          ],
          "local-arguments" :
          [
            "...",
          ],
          "object": "CMakeFiles/target.dir/source.cxx.o",
          "private": true,
          "provides": {
            "importable": "path/to/bmi"
          },
          "requires" : [],
          "source": "path/to/source.cxx",
          "work-directory": "/path/to/working/directory"
        }
      ],
      "visible-sets" : []
    }
```

```
  ]
}
```

This is initialized by the *CMAKE_EXPORT_BUILD_DATABASE* environment variable, and initializes the *EXPORT_BUILD_DATABASE* target property for all targets.

NOTE:
This option is implemented only by the *Ninja Generators*. It is ignored on other generators.

When supported and enabled, numerous targets are created in order to make it possible to build a file containing just the commands that are needed for the tool in question.

cmake_build_database-<CONFIG>
Writes **build_database_<CONFIG>.json**. Writes a build database for the entire build for the given configuration and all languages. Not available if the configuration name is the empty string.

cmake_build_database-<LANG>-<CONFIG>
Writes **build_database_<LANG>_<CONFIG>.json**. Writes build database for the entire build for the given configuration and language. Not available if the configuration name is the empty string.

cmake_build_database-<LANG>
Writes **build_database_<LANG>.json**. Writes build database for the entire build for the given language and all configurations. In a multi-config generator, other build configuration database may be assumed to exist.

cmake_build_database
Writes to **build_database.json**. Writes build database for all languages and configurations. In a multi-config generator, other build configuration database may be assumed to exist.

CMAKE_EXPORT_COMPILE_COMMANDS
Added in version 3.5.

Enable/Disable output of compile commands during generation.

If enabled, generates a **compile_commands.json** file containing the exact compiler calls for all translation units of the project in machine-readable form. The format of the JSON file looks like:

```
[
  {
    "directory": "/home/user/development/project",
    "command": "/usr/bin/c++ ... -c ../foo/foo.cc",
    "file": "../foo/foo.cc",
    "output": "../foo.dir/foo.cc.o"
  },

  ...

  {
    "directory": "/home/user/development/project",
    "command": "/usr/bin/c++ ... -c ../foo/bar.cc",
    "file": "../foo/bar.cc",
    "output": "../foo.dir/bar.cc.o"
  }
]
```

This is initialized by the *CMAKE_EXPORT_COMPILE_COMMANDS* environment variable, and initializes the *EXPORT_COMPILE_COMMANDS* target property for all targets.

NOTE:
This option is implemented only by *Makefile Generators* and *Ninja Generators*. It is ignored on other generators.

This option currently does not work well in combination with the *UNITY_BUILD* target property or the *CMAKE_UNITY_BUILD* variable.

CMAKE_EXPORT_SARIF
Added in version 4.0.

Enable or disable CMake diagnostics output in SARIF format for a project.

If enabled, CMake will generate a SARIF log file containing diagnostic messages output by CMake when running in a project. By default, the log file is written to **.cmake/sarif/cmake.sarif**, but the location can be changed by setting the command−line option *cmake −−sarif−output* to the desired path.

The Static Analysis Results Interchange Format (SARIF) is a JSON−based standard format for static analysis tools (including build tools like CMake) to record and communicate diagnostic messages. CMake generates a SARIF log entry for warnings and errors produced while running CMake on a project (e.g. *message()* calls). Each log entry includes the message, severity, and location information if available.

An example of CMake's SARIF output is:

```
{
  "version" : "2.1.0",
  "$schema" : "https://schemastore.azurewebsites.net/schemas/json/sarif−2.1.0−rtm
  "runs" :
  [
    {
      "tool" :
      {
        "driver" :
        {
          "name" : "CMake",
          "rules" :
          [
            {
              "id" : "CMake.Warning",
              "messageStrings" :
              {
                "default" :
                {
                  "text" : "CMake Warning: {0}"
                }
              },
              "name" : "CMake Warning"
            }
          ]
        }
      },
      "results" :
```

```
            [
              {
                "level" : "warning",
                "locations" :
                [
                  {
                    "physicalLocation" :
                    {
                      "artifactLocation" :
                      {
                        "uri" : "/home/user/development/project/CMakeLists.txt"
                      },
                      "region" :
                      {
                        "startLine" : 5
                      }
                    }
                  }
                ],
                "message" :
                {
                  "text" : "An example warning"
                },
                "ruleId" : "CMake.Warning",
                "ruleIndex" : 0
              }
            ]
          }
        ]
      }
```

CMAKE_EXPORT_PACKAGE_REGISTRY
Added in version 3.15.

Enables the *export(PACKAGE)* command when *CMP0090* is set to **NEW**.

The *export(PACKAGE)* command does nothing by default. In some cases it is desirable to write to the user package registry, so the **CMAKE_EXPORT_PACKAGE_REGISTRY** variable may be set to enable it.

If *CMP0090* is *not* set to **NEW** this variable does nothing, and the *CMAKE_EXPORT_NO_PACKAGE_REGISTRY* variable controls the behavior instead.

See also *Disabling the Package Registry*.

CMAKE_EXPORT_NO_PACKAGE_REGISTRY
Added in version 3.1.

Disable the *export(PACKAGE)* command when *CMP0090* is not set to **NEW**.

In some cases, for example for packaging and for system wide installations, it is not desirable to write the user package registry. If the**CMAKE_EXPOR T_NO_PACKAGE_REGISTRY** variable is enabled, the *export(PACKAGE)* command will do nothing.

If *CMP0090* is set to **NEW** this variable does nothing, and the *CMAKE_EXPORT_PACKAGE_REGISTRY*

variable controls the behavior instead.

See also *Disabling the Package Registry*.

CMAKE_FIND_APPBUNDLE
Added in version 3.4.

This variable affects how **find_*** commands choose between macOS Application Bundles and unix−style package components.

On Darwin or systems supporting macOS Application Bundles, the **CMAKE_FIND_APPBUNDLE** variable 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.

CMAKE_FIND_FRAMEWORK
Added in version 3.4.

This variable affects how **find_*** commands choose between macOS Frameworks and unix−style package components.

On Darwin or systems supporting macOS Frameworks, the **CMAKE_FIND_FRAMEWORK** variable 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.

CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX
Added in version 3.9.

Specify a **<suffix>** to tell the *find_library()* command to search in a **lib<suffix>** directory before each **lib** directory that would normally be searched.

This overrides the behavior of related global properties:

- *FIND_LIBRARY_USE_LIB32_PATHS*
- *FIND_LIBRARY_USE_LIB64_PATHS*
- *FIND_LIBRARY_USE_LIBX32_PATHS*

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, depending on the compiler, **.dll.lib**, **.dll.a**, **.a** (e.g. rustc, GCC, or Clang), so when it tries to find the **foo** library, it will look for **[<prefix>]foo[.dll].lib** and/or **[<prefix>]foo[.dll].a**, depending on the compiler used and the **<prefix>** specified in the *CMAKE_FIND_LIBRARY_PREFIXES*.

CMAKE_FIND_NO_INSTALL_PREFIX
Exclude the values of the *CMAKE_INSTALL_PREFIX* and *CMAKE_STAGING_PREFIX* variables from *CMAKE_SYSTEM_PREFIX_PATH*. CMake adds these project−destination prefixes to *CMAKE_SYSTEM_PREFIX_PATH* by default in order to support building a series of dependent packages and installing them into a common prefix. Set **CMAKE_FIND_NO_INSTALL_PREFIX** to **TRUE** to suppress this behavior.

The *CMAKE_SYSTEM_PREFIX_PATH* is initialized on the first call to a *project()* or *enable_language()* command. Therefore one must set**CMAKE_FIND_NO_INST ALL_PREFIX** before this in order to take effect. A user may set the variable as a cache entry on the command line to achieve this.

Note that the prefix(es) may still be searched for other reasons, such as being the same prefix as the CMake installation, or for being a built−in system prefix.

CMAKE_FIND_PACKAGE_PREFER_CONFIG
Added in version 3.15.

Tell *find_package()* to try "Config" mode before "Module" mode if no mode was specified.

The command *find_package()* operates without an explicit mode when the reduced signature is used without the **MODULE** option. In this case, by default, CMake first tries Module mode by searching for a **Find<pkg>.cmake** module. If it fails, CMake then searches for the package using Config mode.

Set **CMAKE_FIND_PACKAGE_PREFER_CONFIG** to **TRUE** to tell *find_package()* to first search using Config mode before falling back to Module mode.

This variable may be useful when a developer has compiled a custom version of a common library and wishes to link it to a dependent project. If this variable is set to **TRUE**, it would prevent a dependent project's call to *find_package()* from selecting the default library located by the system's **Find<pkg>.cmake** module before finding the developer's custom built library.

Once this variable is set, it is the responsibility of the exported **<pkg>Config.cmake** files to provide the same result variables as the **Find<pkg>.cmake** modules so that dependent projects can use them interchangeably.

CMAKE_FIND_PACKAGE_RESOLVE_SYMLINKS
Added in version 3.14.

Set to **TRUE** to tell *find_package()* calls to resolve symbolic links in the value of **<PackageName>_DIR**.

This is helpful in use cases where the package search path points at a proxy directory in which symlinks to the real package locations appear. This is not enabled by default because there are also common use cases in which the symlinks should be preserved.

CMAKE_FIND_PACKAGE_TARGETS_GLOBAL
Added in version 3.24.

Setting to **TRUE** promotes all *IMPORTED* targets discovered by *find_package()* to a **GLOBAL** scope.

Setting this to **TRUE** is akin to specifying **GLOBAL** as an argument to *find_package()*. Default value is **OFF**.

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.

This variable has no effect if *CMAKE_FIND_PACKAGE_PREFER_CONFIG* is set to **TRUE**.

CMAKE_FIND_ROOT_PATH
Semicolon−separated list of root paths to search on the filesystem.

This variable is most useful when cross−compiling. CMake uses the paths in this list as alternative roots to find filesystem items with *find_package()*, *find_library()* etc.

CMAKE_FIND_ROOT_PATH_MODE_INCLUDE
This variable controls whether the *CMAKE_FIND_ROOT_PATH* and *CMAKE_SYSROOT* are used by *find_file()* and *find_path()*.

If set to **ONLY**, then only the roots in *CMAKE_FIND_ROOT_PATH* will be searched. If set to **NEVER**, then the roots in *CMAKE_FIND_ROOT_PATH* will be ignored and only the host system root will be used. If set to **BOTH**, then the host system paths and the paths in *CMAKE_FIND_ROOT_PATH* will be searched.

CMAKE_FIND_ROOT_PATH_MODE_LIBRARY
This variable controls whether the *CMAKE_FIND_ROOT_PATH* and *CMAKE_SYSROOT* are used by *find_library()*.

If set to **ONLY**, then only the roots in *CMAKE_FIND_ROOT_PATH* will be searched. If set to **NEVER**, then the roots in *CMAKE_FIND_ROOT_PATH* will be ignored and only the host system root will be used. If set to **BOTH**, then the host system paths and the paths in *CMAKE_FIND_ROOT_PATH* will be searched.

CMAKE_FIND_ROOT_PATH_MODE_PACKAGE
This variable controls whether the *CMAKE_FIND_ROOT_PATH* and *CMAKE_SYSROOT* are used by *find_package()*.

If set to **ONLY**, then only the roots in *CMAKE_FIND_ROOT_PATH* will be searched. If set to **NEVER**, then the roots in *CMAKE_FIND_ROOT_PATH* will be ignored and only the host system root will be used. If set to **BOTH**, then the host system paths and the paths in *CMAKE_FIND_ROOT_PATH* will be searched.

CMAKE_FIND_ROOT_PATH_MODE_PROGRAM
This variable controls whether the *CMAKE_FIND_ROOT_PATH* and *CMAKE_SYSROOT* are used by *find_program()*.

If set to **ONLY**, then only the roots in *CMAKE_FIND_ROOT_PATH* will be searched. If set to **NEVER**, then the roots in *CMAKE_FIND_ROOT_PATH* will be ignored and only the host system root will be used. If set to **BOTH**, then the host system paths and the paths in *CMAKE_FIND_ROOT_PATH* will be searched.

CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH
Added in version 3.16.

Controls the default behavior of the following commands for whether or not to search paths provided by cmake−specific environment variables:

- *find_program()*
- *find_library()*
- *find_file()*
- *find_path()*
- *find_package()*

This is useful in cross−compiling environments.

By default this variable is not set, which is equivalent to it having a value of **TRUE**. Explicit options given to the above commands take precedence over this variable.

See also the *CMAKE_FIND_USE_CMAKE_PATH*, *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH*, *CMAKE_FIND_USE_INSTALL_PREFIX*, *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY*, *CMAKE_FIND_USE_PACKAGE_REGISTRY*, and *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* variables.

CMAKE_FIND_USE_CMAKE_PATH
Added in version 3.16.

Controls the default behavior of the following commands for whether or not to search paths provided by cmake−specific cache variables:

- *find_program()*
- *find_library()*
- *find_file()*
- *find_path()*
- *find_package()*

This is useful in cross−compiling environments.

By default this variable is not set, which is equivalent to it having a value of **TRUE**. Explicit options given to the above commands take precedence over this variable.

See also the *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH*, *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY*, *CMAKE_FIND_USE_PACKAGE_REGISTRY*, and *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* variables.

CMAKE_FIND_USE_CMAKE_SYSTEM_PATH
Added in version 3.16.

Controls the default behavior of the following commands for whether or not to search paths provided by platform−specific cmake variables:

- *find_program()*
- *find_library()*
- *find_file()*
- *find_path()*
- *find_package()*

This is useful in cross−compiling environments.

By default this variable is not set, which is equivalent to it having a value of **TRUE**. Explicit options given to the above commands take precedence over this variable.

See also the *CMAKE_FIND_USE_CMAKE_PATH*, *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_INSTALL_PREFIX*, *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY*, *CMAKE_FIND_USE_PACKAGE_REGISTRY*, and *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* variables.

CMAKE_FIND_USE_INSTALL_PREFIX
Added in version 3.24.

Controls the default behavior of the following commands for whether or not to search the locations in the *CMAKE_INSTALL_PREFIX* and *CMAKE_STAGING_PREFIX* variables.

- *find_program()*
- *find_library()*
- *find_file()*
- *find_path()*
- *find_package()*

This is useful in cross−compiling environments.

Due to backwards compatibility with *CMAKE_FIND_NO_INSTALL_PREFIX*, the behavior of the find command change based on if this variable exists.

CMAKE_FIND_USE_INSTALL_PREFIX	CMAKE_FIND_NO_INSTALL_PREFIX	Search
Not Defined	On	NO
Not Defined	Off || Not Defined	YES
Off	On	NO
Off	Off || Not Defined	NO
On	On	YES
On	Off || Not Defined	YES

By default this variable is not defined. Explicit options given to the above commands take precedence over this variable.

See also the *CMAKE_FIND_USE_CMAKE_PATH*, *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH*,

CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY, *CMAKE_FIND_USE_PACKAGE_REGISTRY*, and *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* variables.

CMAKE_FIND_USE_PACKAGE_REGISTRY
Added in version 3.16.

Controls the default behavior of the *find_package()* command for whether or not to search paths provided by the *User Package Registry*.

By default this variable is not set and the behavior will fall back to that determined by the deprecated *CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY* variable. If that is also not set, then *find_package()* will use the *User Package Registry* unless the **NO_CMAKE_PACKAGE_REGISTRY** option is provided.

This variable takes precedence over *CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY* when both are set.

In some cases, for example to locate only system wide installations, it is not desirable to use the *User Package Registry* when searching for packages. If the **CMAKE_FIND_USE_PACKAGE_REGISTRY** variable is **FALSE**, all the *find_package()* commands will skip the *User Package Registry* as if they were called with the **NO_CMAKE_PACKAGE_REGISTRY** argument.

See also *Disabling the Package Registry* and the *CMAKE_FIND_USE_CMAKE_PATH*, *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_INSTALL_PREFIX*, *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH*, *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY*, and *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* variables.

CMAKE_FIND_USE_PACKAGE_ROOT_PATH
Added in version 3.16.

Controls the default behavior of the following commands for whether or not to search paths provided by *<PackageName>_ROOT* variables:

- *find_program()*
- *find_library()*
- *find_file()*
- *find_path()*
- *find_package()*

By default this variable is not set, which is equivalent to it having a value of **TRUE**. Explicit options given to the above commands take precedence over this variable.

See also the *CMAKE_FIND_USE_CMAKE_PATH*, *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_INSTALL_PREFIX*, *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH*, *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY*, and *CMAKE_FIND_USE_PACKAGE_REGISTRY* variables.

CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH
Added in version 3.16.

Controls the default behavior of the following commands for whether or not to search paths provided by standard system environment variables:

- *find_program()*
- *find_library()*
- *find_file()*
- *find_path()*
- *find_package()*

This is useful in cross−compiling environments.

By default this variable is not set, which is equivalent to it having a value of **TRUE**. Explicit options given to the above commands take precedence over this variable.

See also the *CMAKE_FIND_USE_CMAKE_PATH*, *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_INSTALL_PREFIX*, *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH*, *CMAKE_FIND_USE_PACKAGE_REGISTRY*, *CMAKE_FIND_USE_PACKAGE_ROOT_PATH*, and *CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY* variables.

CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY
Added in version 3.16.

Controls searching the *System Package Registry* by the *find_package()* command.

By default this variable is not set and the behavior will fall back to that determined by the deprecated *CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY* variable. If that is also not set, then *find_package()* will use the *System Package Registry* unless the **NO_CMAKE_SYSTEM_PACKAGE_REGISTRY** option is provided.

This variable takes precedence over *CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY* when both are set.

In some cases, for example to locate only user specific installations, it is not desirable to use the *System Package Registry* when searching for packages. If the **CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY** variable is **FALSE**, all the *find_package()* commands will skip the *System Package Registry* as if they were called with the **NO_CMAKE_SYSTEM_PACKAGE_REGISTRY** argument.

See also *Disabling the Package Registry*.

See also the *CMAKE_FIND_USE_CMAKE_PATH*, *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_INSTALL_PREFIX*, *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH*, *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH*, *CMAKE_FIND_USE_PACKAGE_REGISTRY*, and *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* variables.

CMAKE_FRAMEWORK_PATH
Semicolon−separated list of directories specifying a search path for macOS frameworks used by the *find_library()*, *find_package()*, *find_path()*, and *find_file()* commands.

There is also an environment variable *CMAKE_FRAMEWORK_PATH*, which is used as an additional list of search directories.

CMAKE_IGNORE_PATH
Semicolon−separated list of directories to be ignored by the various **find...()** commands.

For *find_program()*, *find_library()*, *find_file()*, and *find_path()*, any file found in one of the listed directories will be ignored. The listed directories do not apply recursively, so any subdirectories to be ignored must also be explicitly listed. **CMAKE_IGNORE_P ATH** does not affect the search *prefixes* used by these four commands. To ignore individual paths under a search prefix (e.g. **bin**, **include**, **lib**, etc.), each path must be listed in **CMAKE_IGNORE_PATH** as a full absolute path. *CMAKE_IGNORE_PREFIX_PATH* provides a more appropriate way to ignore a whole search prefix.

find_package() is also affected by **CMAKE_IGNORE_PATH**, but only for *Config mode* searches. Any **<Name>Config.cmake** or **<name>−config.cmake** file found in one of the specified directories will be ignored. In addition, any search *prefix* found in **CMAKE_IGNORE_PATH** will be skipped for backward compatibility reasons, but new code should prefer to use *CMAKE_IGNORE_PREFIX_PATH* to ignore prefixes instead.

Ignoring search locations can be useful in cross−compiling 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.

By default, **CMAKE_IGNORE_PATH** is empty. It is intended to be set by the project or the end user.

See also the following variables:

- *CMAKE_IGNORE_PREFIX_PATH*
- *CMAKE_SYSTEM_IGNORE_PATH*
- *CMAKE_PREFIX_PATH*
- *CMAKE_LIBRARY_PATH*
- *CMAKE_INCLUDE_PATH*
- *CMAKE_PROGRAM_PATH*

CMAKE_IGNORE_PREFIX_PATH
Added in version 3.23.

Semicolon−separated list of search *prefixes* to be ignored by the *find_program()*, *find_library()*, *find_file()*, and *find_path()* commands. The prefixes are also ignored by the *Config mode* of the *find_package()* command (*Module mode* is unaffected). To ignore specific directories instead, see *CMAKE_IGNORE_PATH*.

Ignoring search locations can be useful in cross−compiling 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.

By default, **CMAKE_IGNORE_PREFIX_PATH** is empty. It is intended to be set by the project or the end user.

See also the following variables:

- *CMAKE_IGNORE_PATH*
- *CMAKE_SYSTEM_IGNORE_PREFIX_PATH*
- *CMAKE_PREFIX_PATH*

- *CMAKE_LIBRARY_PATH*
- *CMAKE_INCLUDE_PATH*
- *CMAKE_PROGRAM_PATH*

CMAKE_INCLUDE_DIRECTORIES_BEFORE
Whether to append or prepend directories by default in *include_directories()*.

This variable affects the default behavior of the *include_directories()* command. Setting this variable to **ON** is equivalent to using the **BEFORE** option in all uses of that command.

CMAKE_INCLUDE_DIRECTORIES_PROJECT_BEFORE
Whether to force prepending of project include directories.

This variable affects the order of include directories generated in compiler command lines. If set to **ON**, it causes the *CMAKE_SOURCE_DIR* and the *CMAKE_BINARY_DIR* to appear first.

CMAKE_INCLUDE_PATH
Semicolon−separated list of directories specifying a search path for the *find_file()* and *find_path()* commands. By default it is empty, it is intended to be set by the project.

There is also an environment variable *CMAKE_INCLUDE_PATH*, which is used as an additional list of search directories.

See also *CMAKE_SYSTEM_INCLUDE_PATH* and *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_DEFAULT_DIRECTORY_PERMISSIONS
Added in version 3.11.

Default permissions for directories created implicitly during installation of files by *install()* and *file(INSTALL)*.

If **make install** is invoked and directories are implicitly created they get permissions set by **CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS** variable or platform specific default permissions if the variable is not set.

Implicitly created directories are created if they are not explicitly installed by *install()* command but are needed to install a file on a certain path. Example of such locations are directories created due to the setting of *CMAKE_INSTALL_PREFIX*.

Expected content of the **CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS** variable is a list of permissions that can be used by *install()* command **PERMISSIONS** section.

Example usage:

```
set(CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS
     OWNER_READ
     OWNER_WRITE
     OWNER_EXECUTE
     GROUP_READ
```

)

CMAKE_INSTALL_MESSAGE
Added in version 3.1.

Specify verbosity of installation script code generated by the *install()* command (using the *file(INSTALL)* command). For paths that are newly installed or updated, installation may print lines like:

```
-- Installing: /some/destination/path
```

For paths that are already up to date, installation may print lines like:

```
-- Up-to-date: /some/destination/path
```

The **CMAKE_INSTALL_MESSAGE** variable may be set to control which messages are printed:

ALWAYS
Print both **Installing** and **Up-to-date** messages.

LAZY Print **Installing** but not **Up-to-date** messages.

NEVER
Print neither **Installing** nor **Up-to-date** messages.

Other values have undefined behavior and may not be diagnosed.

If this variable is not set, the default behavior is **ALWAYS**.

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 as follows:

- Added in version 3.29: If the *CMAKE_INSTALL_PREFIX* environment variable is set, its value is used as default for this variable.

- **c:/Program Files/${PROJECT_NAME}** on Windows.
- **/usr/local** on UNIX platforms.

See *CMAKE_INSTALL_PREFIX_INITIALIZED_TO_DEFAULT* for how a project might choose its own default.

On UNIX one can use the **DESTDIR** mechanism in order to relocate the whole installation to a staging area. See the*DESTDIR* en vironment variable for more information.

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. This behavior can be disabled by setting the *CMAKE_FIND_NO_INSTALL_PREFIX* to **TRUE** before the first *project()* invocation.

NOTE:
Use the *GNUInstallDirs* module to provide GNU-style options for the layout of directories within the installation.

The **CMAKE_INSTALL_PREFIX** may be defined when configuring a build tree to set its installation prefix. Or, when using the *cmake(1)* command−line tool's *−−install* mode, one may specify a different prefix using the *−−prefix* option:

```
cmake --install . --prefix /my/install/prefix
```

CMAKE_INSTALL_PREFIX_INITIALIZED_TO_DEFAULT
Added in version 3.7.1.

CMake sets this variable to a **TRUE** value when the *CMAKE_INSTALL_PREFIX* has just been initialized to its default value, typically on the first run of CMake within a new build tree and the *CMAKE_INSTALL_PREFIX* environment variable is not set on the first run of CMake. This can be used by project code to change the default without overriding a user−provided value:

```
if(CMAKE_INSTALL_PREFIX_INITIALIZED_TO_DEFAULT)
  set_property(CACHE CMAKE_INSTALL_PREFIX PROPERTY VALUE "/my/default")
endif()
```

CMAKE_KATE_FILES_MODE
Added in version 3.27.

This cache variable is used by the Kate project generator and controls to what mode the **files** entry in the project file will be set. See *cmake−generators(7)*.

Possible values are **AUTO**, **SVN**, **GIT**, **HG**, **FOSSIL** and **LIST**.

When set to **LIST**, CMake will put the list of source files known to CMake in the project file. When set to **SVN**, **GIT**, **HG** or **FOSSIL**, CMake will set the generated project accordingly to Subversion, git, Mercurial or Fossil, and Kate will then use the respective command line tool to retrieve the list of files in the project. When unset or set to**A UTO**, CMake will try to detect whether the source directory is part of a git or svn checkout or not, and put the respective entry into the project file.

CMAKE_KATE_MAKE_ARGUMENTS
Added in version 3.0.

This cache variable is used by the Kate project generator. See*cmak e−generators(7)*.

This variable holds arguments which are used when Kate invokes the make tool. By default it is initialized to hold flags to enable parallel builds (using −j typically).

CMAKE_LIBRARY_PATH
Semicolon−separated list of directories specifying a search path for the *find_library()* command. By default it is empty, it is intended to be set by the project.

There is also an environment variable *CMAKE_LIBRARY_PATH*, which is used as an additional list of search directories.

See also *CMAKE_SYSTEM_LIBRARY_PATH* and *CMAKE_PREFIX_PATH*.

CMAKE_LINK_DIRECTORIES_BEFORE
Added in version 3.13.

Whether to append or prepend directories by default in *link_directories()*.

This variable affects the default behavior of the *link_directories()* command. Setting this variable to **ON** is equivalent to using the **BEFORE** option in all uses of that command.

CMAKE_LINK_LIBRARIES_ONLY_TARGETS
Added in version 3.23.

Set this variable to initialize the *LINK_LIBRARIES_ONLY_TARGETS* property of non−imported targets when they are created. Setting it to true enables an additional check that all items named by *target_link_libraries()* that can be target names are actually names of existing targets. See the target property documentation for details.

CMAKE_MAXIMUM_RECURSION_DEPTH
Added in version 3.14.

Maximum recursion depth for CMake scripts. It is intended to be set on the command line with **−DCMAKE_MAXIMUM_RECURSION_DEPTH=<x>**, or within **CMakeLists.txt** by projects that require a large recursion depth. Projects that set this variable should provide the user with a way to override it. For example:

```
# About to perform deeply recursive actions
if(NOT CMAKE_MAXIMUM_RECURSION_DEPTH)
  set(CMAKE_MAXIMUM_RECURSION_DEPTH 2000)
endif()
```

If it is not set, or is set to a non−integer value, a sensible default limit is used. If the recursion limit is reached, the script terminates immediately with a fatal error.

Calling any of the following commands increases the recursion depth:

- *include()*
- *find_package()*
- *add_subdirectory()*
- *try_compile()*
- *ctest_read_custom_files()*
- *ctest_run_script()* (unless **NEW_PROCESS** is specified)
- User−defined *function()*'s and *macro()*'s (note that *function()* and *macro()* themselves don't increase recursion depth)
- Reading or writing variables that are being watched by a *variable_watch()*

See also the *CMAKE_MAXIMUM_RECURSION_DEPTH* environment variable.

CMAKE_MESSAGE_CONTEXT
Added in version 3.17.

When enabled by the *cmake −−log−context* command line option or the *CMAKE_MESSAGE_CONTEXT_SHOW* variable, the *message()* command converts the **CMAKE_MESSAGE_CONTEXT** list into a dot−separated string surrounded by square brackets and prepends it to each line for messages of log levels **NOTICE** and below.

For logging contexts to work effectively, projects should generally **APPEND** and **POP_BACK** an item to the current value of **CMAKE_MESSAGE_CONTEXT** rather than replace it. Projects should not assume

the message context at the top of the source tree is empty, as there are scenarios where the context might have already been set (e.g. hierarchical projects).

WARNING:
Valid context names are restricted to anything that could be used as a CMake variable name. All names that begin with an underscore or the string **cmake_** are also reserved for use by CMake and should not be used by projects.

Example:

```
function(bar)
  list(APPEND CMAKE_MESSAGE_CONTEXT "bar")
  message(VERBOSE "bar VERBOSE message")
endfunction()

function(baz)
  list(APPEND CMAKE_MESSAGE_CONTEXT "baz")
  message(DEBUG "baz DEBUG message")
endfunction()

function(foo)
  list(APPEND CMAKE_MESSAGE_CONTEXT "foo")
  bar()
  message(TRACE "foo TRACE message")
  baz()
endfunction()

list(APPEND CMAKE_MESSAGE_CONTEXT "top")

message(VERBOSE "Before `foo`")
foo()
message(VERBOSE "After `foo`")

list(POP_BACK CMAKE_MESSAGE_CONTEXT)
```

Which results in the following output:

```
-- [top] Before `foo`
-- [top.foo.bar] bar VERBOSE message
-- [top.foo] foo TRACE message
-- [top.foo.baz] baz DEBUG message
-- [top] After `foo`
```

CMAKE_MESSAGE_CONTEXT_SHOW
Added in version 3.17.

Setting this variable to true enables showing a context with each line logged by the *message()* command (see *CMAKE_MESSAGE_CONTEXT* for how the context itself is specified).

This variable is an alternative to providing the **--log-context** option on the *cmake* command line. Whereas the command line option will apply only to that one CMake run, setting **CMAKE_MESSAGE_CONTEXT_SHOW** to true as a cache variable will ensure that subsequent CMake runs will continue to show the message context.

Projects should not set **CMAKE_MESSAGE_CONTEXT_SHOW**. It is intended for users so that they may control whether or not to include context with messages.

CMAKE_MESSAGE_INDENT
Added in version 3.16.

The *message()* command joins the strings from this list and for log levels of **NOTICE** and below, it prepends the resultant string to each line of the message.

Example:

```
list(APPEND listVar one two three)

message(VERBOSE [[Collected items in the "listVar":]])
list(APPEND CMAKE_MESSAGE_INDENT "  ")

foreach(item IN LISTS listVar)
  message(VERBOSE ${item})
endforeach()

list(POP_BACK CMAKE_MESSAGE_INDENT)
message(VERBOSE "No more indent")
```

Which results in the following output:

```
-- Collected items in the "listVar":
--   one
--   two
--   three
-- No more indent
```

CMAKE_MESSAGE_LOG_LEVEL
Added in version 3.17.

When set, this variable specifies the logging level used by the *message()* command. Valid values are the same as those for the *--log-level* command line option of the *cmake(1)* program. If this variable is set and the *--log-level* command line option is given, the command line option takes precedence.

The main advantage to using this variable is to make a log level persist between CMake runs. Setting it as a cache variable will ensure that subsequent CMake runs will continue to use the chosen log level.

Projects should not set this variable, it is intended for users so that they may control the log level according to their own needs.

Added in version 3.25: See the *cmake_language() cmake_language* command for a way to query the current message logging level.

CMAKE_MFC_FLAG
Use the MFC library for an executable or dll.

Enables the use of the Microsoft Foundation Classes (MFC). It should be set to **1** for the static MFC library, and **2** for the shared MFC library. This is used in Visual Studio project files.

Usage example:

```
add_definitions(-D_AFXDLL)
set(CMAKE_MFC_FLAG 2)
add_executable(CMakeSetup WIN32 ${SRCS})
```

Contents of **CMAKE_MFC_FLAG** may use *generator expressions*.

CMAKE_MODULE_PATH

Semicolon-separated list of directories, represented using forward slashes, specifying a search path for CMake modules to be loaded by the *include()* or *find_package()* commands before checking the default modules that come with CMake. By default it is empty. It is intended to be set by the project.

It's fairly common for a project to have a directory containing various ***.cmake** files to assist in development. Adding the directory to the *CMAKE_MODULE_PATH* simplifies loading them. For example, a project's top-level **CMakeLists.txt** file may contain:

```
list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/cmake")

include(Foo) # Loads ${CMAKE_CURRENT_SOURCE_DIR}/cmake/Foo.cmake

find_package(Bar) # Loads ${CMAKE_CURRENT_SOURCE_DIR}/cmake/FindBar.cmake
```

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 as a way to set its own policies; use *cmake_policy(SET)* instead. This variable is meant to externally set policies for which a project has not itself been updated:

- Users running CMake may set this variable in the cache (e.g. **-DCMAKE_POLICY_DEFAULT_CMP<NNNN>=<OLD|NEW>**). Set it to**OLD** to quiet a polic y warning while using old behavior or to **NEW** to try building the project with new behavior.
- Projects may set this variable before a call to *add_subdirectory()* that adds a third-party project in order to set its policies without modifying third-party code.

See *CMAKE_POLICY_VERSION_MINIMUM* set policies to **NEW** based on the version of CMake that introduced them.

CMAKE_POLICY_VERSION_MINIMUM

Added in version 4.0.

Specify a minimum *Policy Version* for a project without modifying its calls to *cmake_minimum_required(VERSION)* and *cmake_policy(VERSION)*.

This variable should not be set by a project in CMake code as a way to set its own policy version. Use *cmake_minimum_required(VERSION)* and/or *cmake_policy(VERSION)* for that. This variable is meant to externally set policies for which a project has not itself been updated:

- Users running CMake may set this variable in the cache, e.g., **-DCMAKE_POLICY_VERSION_MINIMUM=3.5**, to try configuring a project that has not been updated to set at least that policy version itself.

- Projects may set this variable before a call to *add_subdirectory()* that adds a third−party project in order to set its policy version without modifying third−party code.

See *CMAKE_POLICY_DEFAULT_CMP<NNNN>* to set individual policies.

CMAKE_POLICY_WARNING_CMP<NNNN>
Explicitly enable or disable the warning when CMake Policy **CMP<NNNN>** has not been set explicitly by *cmake_policy()* or implicitly by *cmake_minimum_required()*. This is meaningful only for the policies that do not warn by default:

- **CMAKE_POLICY_WARNING_CMP0025** controlled the warning for policy *CMP0025* in CMake versions before 4.0.
- **CMAKE_POLICY_WARNING_CMP0047** controlled the warning for policy *CMP0047* in CMake versions before 4.0.
- **CMAKE_POLICY_WARNING_CMP0056** controlled the warning for policy *CMP0056* in CMake versions before 4.0.
- **CMAKE_POLICY_WARNING_CMP0060** controlled the warning for policy *CMP0060* in CMake versions before 4.0.
- **CMAKE_POLICY_WARNING_CMP0065** controlled the warning for policy *CMP0065* in CMake versions before 4.0.
- **CMAKE_POLICY_WARNING_CMP0066** controls the warning for policy *CMP0066*.
- **CMAKE_POLICY_WARNING_CMP0067** controls the warning for policy *CMP0067*.
- **CMAKE_POLICY_WARNING_CMP0082** controls the warning for policy *CMP0082*.
- **CMAKE_POLICY_WARNING_CMP0089** controls the warning for policy *CMP0089*.
- **CMAKE_POLICY_WARNING_CMP0102** controls the warning for policy *CMP0102*.
- **CMAKE_POLICY_WARNING_CMP0112** controls the warning for policy *CMP0112*.
- **CMAKE_POLICY_WARNING_CMP0116** controls the warning for policy *CMP0116*.
- **CMAKE_POLICY_WARNING_CMP0126** controls the warning for policy *CMP0126*.
- **CMAKE_POLICY_WARNING_CMP0128** controls the warning for policy *CMP0128*.
- **CMAKE_POLICY_WARNING_CMP0129** controls the warning for policy *CMP0129*.
- **CMAKE_POLICY_WARNING_CMP0133** controls the warning for policy *CMP0133*.
- **CMAKE_POLICY_WARNING_CMP0172** controls the warning for policy *CMP0172*.

This variable should not be set by a project in CMake code. Project developers running CMake may set this variable in their cache to enable the warning (e.g. **−DCMAKE_POLICY_WARNING_CMP<NNNN>=ON**). Alternatively, running *cmake(1)* with the *−−debug−output*, *−−trace*, or *−−trace−expand* option will also enable the warning.

CMAKE_PREFIX_PATH
Semicolon−separated list of directories specifying installation *prefixes* to be searched by the *find_package()*, *find_program()*, *find_library()*, *find_file()*, and *find_path()* commands. Each command will add appropriate subdirectories (like **bin**, **lib**, or **include**) as specified in its own documentation.

By default this is empty. It is intended to be set by the project.

There is also an environment variable *CMAKE_PREFIX_PATH*, which is used as an additional list of search prefixes.

See also *CMAKE_SYSTEM_PREFIX_PATH*, *CMAKE_INCLUDE_PATH*, *CMAKE_LIBRARY_PATH*, *CMAKE_PROGRAM_PATH*, and *CMAKE_IGNORE_PATH*.

CMAKE_PROGRAM_PATH
Semicolon−separated list of directories specifying a search path for the *find_program()* command. By default it is empty, it is intended to be set by the project.

There is also an environment variable *CMAKE_PROGRAM_PATH*, which is used as an additional list of search directories.

See also *CMAKE_SYSTEM_PROGRAM_PATH* and *CMAKE_PREFIX_PATH*.

CMAKE_PROJECT_INCLUDE
Added in version 3.15.

A CMake language file to be included as the last step of all *project()* command calls. This is intended for injecting custom code into project builds without modifying their source. See *Code Injection* for a more detailed discussion of files potentially included during a *project()* call.

Added in version 3.29: This variable can be a *semicolon−separated list* of CMake language files to be included sequentially. It can also now refer to module names to be found in *CMAKE_MODULE_PATH* or as a builtin CMake module.

See also the *CMAKE_PROJECT_<PROJECT−NAME>_INCLUDE*, *CMAKE_PROJECT_<PROJECT−NAME>_INCLUDE_BEFORE*, *CMAKE_PROJECT_INCLUDE_BEFORE*, and *CMAKE_PROJECT_TOP_LEVEL_INCLUDES* variables.

CMAKE_PROJECT_INCLUDE_BEFORE
Added in version 3.15.

A CMake language file to be included as the first step of all *project()* command calls. This is intended for injecting custom code into project builds without modifying their source. See *Code Injection* for a more detailed discussion of files potentially included during a *project()* call.

Added in version 3.29: This variable can be a *semicolon−separated list* of CMake language files to be included sequentially. It can also now refer to module names to be found in *CMAKE_MODULE_PATH* or as a builtin CMake module.

See also the *CMAKE_PROJECT_<PROJECT−NAME>_INCLUDE*, *CMAKE_PROJECT_<PROJECT−NAME>_INCLUDE_BEFORE*, *CMAKE_PROJECT_INCLUDE*, and *CMAKE_PROJECT_TOP_LEVEL_INCLUDES* variables.

CMAKE_PROJECT_<PROJECT−NAME>_INCLUDE
A CMake language file to be included as the last step of any *project()* command calls that specify **<PROJECT−NAME>** as the project name. This is intended for injecting custom code into project builds without modifying their source. See *Code Injection* for a more detailed discussion of files potentially included during a *project()* call.

Added in version 3.29: This variable can be a *semicolon−separated list* of CMake language files to be included sequentially. It can also now refer to module names to be found in *CMAKE_MODULE_PATH* or as a builtin CMake module.

See also the *CMAKE_PROJECT_<PROJECT−NAME>_INCLUDE_BEFORE*, *CMAKE_PROJECT_INCLUDE*, *CMAKE_PROJECT_INCLUDE_BEFORE*, and

CMAKE_PROJECT_TOP_LEVEL_INCLUDES variables.

CMAKE_PROJECT_<PROJECT-NAME>_INCLUDE_BEFORE
Added in version 3.17.

A CMake language file to be included as the first step of any *project()* command calls that specify **<PROJECT-NAME>** as the project name. This is intended for injecting custom code into project builds without modifying their source. See *Code Injection* for a more detailed discussion of files potentially included during a *project()* call.

Added in version 3.29: This variable can be a *semicolon-separated list* of CMake language files to be included sequentially. It can also now refer to module names to be found in *CMAKE_MODULE_PATH* or as a builtin CMake module.

See also the *CMAKE_PROJECT_<PROJECT-NAME>_INCLUDE*, *CMAKE_PROJECT_INCLUDE*, *CMAKE_PROJECT_INCLUDE_BEFORE*, and *CMAKE_PROJECT_TOP_LEVEL_INCLUDES* variables.

CMAKE_PROJECT_TOP_LEVEL_INCLUDES
Added in version 3.24.

Semicolon-separated list of CMake language files to include as part of the very first *project()* call. The files will be included immediately after the toolchain file has been read (if one is specified) and platform variables have been set, but before any languages have been enabled. Therefore, language-specific variables, including things like *CMAKE_<LANG>_COMPILER*, might not be set. See *Code Injection* for a more detailed discussion of files potentially included during a *project()* call.

Added in version 3.29: This variable can also now refer to module names to be found in *CMAKE_MODULE_PATH* or builtin to CMake.

This variable is intended for specifying files that perform one-time setup for the build. It provides an injection point for things like configuring package managers, adding logic the user shares between projects (e.g. defining their own custom build types), and so on. It is primarily for users to add things specific to their environment, but not for specifying the toolchain details (use *CMAKE_TOOLCHAIN_FILE* for that).

By default, this variable is empty. It is intended to be set by the user.

See also:

- *CMAKE_PROJECT_INCLUDE*
- *CMAKE_PROJECT_INCLUDE_BEFORE*
- *CMAKE_PROJECT_<PROJECT-NAME>_INCLUDE*
- *CMAKE_PROJECT_<PROJECT-NAME>_INCLUDE_BEFORE*
- *PROPAGATE_TOP_LEVEL_INCLUDES_TO_TRY_COMPILE*

CMAKE_REQUIRE_FIND_PACKAGE_<PackageName>
Added in version 3.22.

Variable for making *find_package()* call **REQUIRED**.

Every non-**REQUIRED** *find_package()* call in a project can be turned into **REQUIRED** by setting the

variable **CMAKE_REQUIRE_FIND_PACKAGE_<PackageName>** to **TRUE**. This can be used to assert assumptions about build environment and to ensure the build will fail early if they do not hold.

Note that setting this variable to true breaks some commonly used patterns. Multiple calls to *find_package()* are sometimes used to obtain a different search order to the default. For example, projects can force checking a known path for a particular package first before searching any of the other default search paths:

```
find_package(something PATHS /some/local/path NO_DEFAULT_PATH)
find_package(something)
```

In the above, the first call looks for the **something** package in a specific directory. If**CMAKE_REQUIRE_FIND_PACKAGE_something** is set to true, then this first call must succeed, otherwise a fatal error occurs. The second call never gets a chance to provide a fall−back to using the default search locations.

A similar pattern is used even by some of CMake's own Find modules to search for a config package first:

```
find_package(something CONFIG QUIET)
if(NOT something_FOUND)
  # Fall back to searching using typical Find module logic...
endif()
```

Again, if **CMAKE_REQUIRE_FIND_PACKAGE_something** is true, the first call must succeed. It effectively means a config package must be found for the dependency, and the Find module logic is never used.

See also the *CMAKE_DISABLE_FIND_PACKAGE_<PackageName>* variable.

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.

See also *CMAKE_SKIP_TEST_ALL_DEPENDENCY*.

CMAKE_SKIP_TEST_ALL_DEPENDENCY
Added in version 3.29.

Control whether the **test** target depends on the **all** target.

If this variable is not defined, or is set to **TRUE**, then the **test** (or **RUN_TESTS**) target does not depend on the **all** (or **ALL_BUILD**) target. When the**test** tar get is built, e.g., via **make test**, the test process will start immediately, regardless of whether the project has been completely built or not.

If **CMAKE_SKIP_TEST_ALL_DEPENDENCY** is explicitly set to **FALSE**, then the **test** target will depend on the **all** target. When the**test** tar get is built, e.g., via **make test**, the **all** target will be built first, and then the tests will run.

See also *CMAKE_SKIP_INSTALL_ALL_DEPENDENCY*.

CMAKE_STAGING_PREFIX
This variable may be set to a path to install to when cross−compiling. This can be useful if the path in *CMAKE_SYSROOT* is read−only, or otherwise should remain pristine.

The **CMAKE_STAGING_PREFIX** location is also used as a search prefix by the **find_*** commands. This can be controlled by setting the *CMAKE_FIND_NO_INSTALL_PREFIX* variable.

If any **RPATH**/**RUNPATH** entries passed to the linker contain the **CMAKE_STAGING_PREFIX**, the matching path fragments are replaced with the *CMAKE_INSTALL_PREFIX*.

CMAKE_SUBLIME_TEXT_2_ENV_SETTINGS
Added in version 3.8.

This variable contains a list of env vars as a list of tokens with the syntax **var=value**.

Example:

```
set(CMAKE_SUBLIME_TEXT_2_ENV_SETTINGS
   "FOO=FOO1\;FOO2\;FOON"
   "BAR=BAR1\;BAR2\;BARN"
   "BAZ=BAZ1\;BAZ2\;BAZN"
   "FOOBAR=FOOBAR1\;FOOBAR2\;FOOBARN"
   "VALID="
   )
```

In case of malformed variables CMake will fail:

```
set(CMAKE_SUBLIME_TEXT_2_ENV_SETTINGS
    "THIS_IS_NOT_VALID"
    )
```

CMAKE_SUBLIME_TEXT_2_EXCLUDE_BUILD_TREE
Added in version 3.8.

If this variable evaluates to **ON** at the end of the top−level **CMakeLists.txt** file, the *Sublime Text 2* extra generator excludes the build tree from the **.sublime−project** if it is inside the source tree.

CMAKE_SUPPRESS_REGENERATION
Added in version 3.12.

If **CMAKE_SUPPRESS_REGENERATION** is **OFF**, which is default, then CMake adds a special target on which all other targets depend that checks the build system and optionally re−runs CMake to regenerate the build system when the target specification source changes.

If this variable evaluates to **ON** at the end of the top−level **CMakeLists.txt** file, CMake will not add the regeneration target to the build system or perform any build system checks.

CMAKE_SYSROOT
Path to pass to the compiler in the **−−sysroot** flag.

The **CMAKE_SYSROOT** content is passed to the compiler in the **−−sysroot** flag, if supported. The path is also stripped from the **RPATH**/**RUNPATH** if necessary on installation. The **CMAKE_SYSROOT** is also used to prefix paths searched by the **find_*** commands.

This variable may only be set in a toolchain file specified by the *CMAKE_TOOLCHAIN_FILE* variable.

See also the *CMAKE_SYSROOT_COMPILE* and *CMAKE_SYSROOT_LINK* variables.

CMAKE_SYSROOT_COMPILE
Added in version 3.9.

Path to pass to the compiler in the **−−sysroot** flag when compiling source files. This is the same as *CMAKE_SYSROOT* but is used only for compiling sources and not linking.

This variable may only be set in a toolchain file specified by the *CMAKE_TOOLCHAIN_FILE* variable.

CMAKE_SYSROOT_LINK
Added in version 3.9.

Path to pass to the compiler in the **−−sysroot** flag when linking. This is the same as *CMAKE_SYSROOT* but is used only for linking and not compiling sources.

This variable may only be set in a toolchain file specified by the *CMAKE_TOOLCHAIN_FILE* variable.

CMAKE_SYSTEM_APPBUNDLE_PATH
Added in version 3.4.

Search path for macOS application bundles used by the *find_program()*, and *find_package()* commands. By default it contains the standard directories for the current system. It is *not* intended to be modified by the project, use *CMAKE_APPBUNDLE_PATH* for this.

CMAKE_SYSTEM_FRAMEWORK_PATH
Added in version 3.4.

Search path for macOS frameworks used by the *find_library()*, *find_package()*, *find_path()*, and *find_file()* commands. By default it contains the standard directories for the current system. It is *not* intended to be modified by the project, use *CMAKE_FRAMEWORK_PATH* for this.

CMAKE_SYSTEM_IGNORE_PATH
Semicolon−separated list of directories to be ignored by the various **find...()** commands.

For *find_program()*, *find_library()*, *find_file()*, and *find_path()*, any file found in one of the listed directories will be ignored. The listed directories do not apply recursively, so any subdirectories to be ignored must also be explicitly listed. **CMAKE_SYSTEM_IGNORE_P ATH** does not affect the search *prefixes* used by these four commands. To ignore individual paths under a search prefix (e.g. **bin**, **include**, **lib**, etc.), each path must be listed in **CMAKE_SYSTEM_IGNORE_PATH** as a full absolute path. *CMAKE_SYSTEM_IGNORE_PREFIX_PATH* provides a more appropriate way to ignore a whole search prefix.

find_package() is also affected by **CMAKE_SYSTEM_IGNORE_PATH**, but only for *Config mode* searches. Any **<Name>Config.cmake** or **<name>−config.cmake** file found in one of the specified directories will be ignored. In addition, any search *prefix* found in **CMAKE_SYSTEM_IGNORE_PATH** will be skipped for backward compatibility reasons, but new code should prefer to use *CMAKE_SYSTEM_IGNORE_PREFIX_PATH* to ignore prefixes instead.

Ignoring search locations can be useful in cross−compiling 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.

CMAKE_SYSTEM_IGNORE_PATH is populated by CMake as part of its platform and toolchain setup. Its purpose is to ignore locations containing incompatible binaries meant for the host rather than the target platform. The project or end user should not modify this variable, they should use *CMAKE_IGNORE_PATH* instead.

See also the following variables:

- *CMAKE_SYSTEM_IGNORE_PREFIX_PATH*
- *CMAKE_SYSTEM_PREFIX_PATH*
- *CMAKE_SYSTEM_LIBRARY_PATH*
- *CMAKE_SYSTEM_INCLUDE_PATH*
- *CMAKE_SYSTEM_PROGRAM_PATH*

CMAKE_SYSTEM_IGNORE_PREFIX_PATH
Added in version 3.23.

Semicolon−separated list of search *prefixes* to be ignored by the *find_program()*, *find_library()*, *find_file()*, and *find_path()* commands. The prefixes are also ignored by the *Config mode* of the *find_package()* command (*Module mode* is unaffected). To ignore specific directories instead, see *CMAKE_SYSTEM_IGNORE_PATH*.

Ignoring search locations can be useful in cross−compiling 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.

CMAKE_SYSTEM_IGNORE_PREFIX_PATH is populated by CMake as part of its platform and toolchain setup. Its purpose is to ignore locations containing incompatible binaries meant for the host rather than the target platform. The project or end user should not modify this variable, they should use *CMAKE_IGNORE_PREFIX_PATH* instead.

See also the following variables:

- *CMAKE_SYSTEM_IGNORE_PATH*
- *CMAKE_SYSTEM_PREFIX_PATH*
- *CMAKE_SYSTEM_LIBRARY_PATH*
- *CMAKE_SYSTEM_INCLUDE_PATH*
- *CMAKE_SYSTEM_PROGRAM_PATH*

CMAKE_SYSTEM_INCLUDE_PATH
Semicolon−separated list of directories specifying a search path for the *find_file()* and *find_path()* commands. By default this 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
Semicolon−separated list of directories specifying a search path for the *find_library()* command. By default this 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
Semicolon−separated list of directories specifying installation *prefixes* to be searched by the *find_package()*, *find_program()*, *find_library()*, *find_file()*, and *find_path()* commands. Each command will

add appropriate subdirectories (like **bin**, **lib**, or **include**) as specified in its own documentation.

By default this contains the system directories for the current system, the *CMAKE_INSTALL_PREFIX*, and the *CMAKE_STAGING_PREFIX*. The installation and staging prefixes may be excluded by setting the *CMAKE_FIND_NO_INSTALL_PREFIX* variable before the first *project()* invocation.

The system directories that are contained in **CMAKE_SYSTEM_PREFIX_PATH** are locations that typically include installed software. An example being **/usr/local** for UNIX based platforms. In addition to standard platform locations, CMake will also add values to **CMAKE_SYSTEM_PREFIX_PATH** based on environment variables. The environment variables and search locations that CMake uses may evolve over time, as platforms and their conventions also evolve. The following provides an indicative list of environment variables and locations that CMake searches, but they are subject to change:

CrayLinuxEnvironment:

- **ENV{SYSROOT_DIR}/**
- **ENV{SYSROOT_DIR}/usr**
- **ENV{SYSROOT_DIR}/usr/local**

Darwin:

- **ENV{SDKROOT}/usr** When **CMAKE_OSX_SYSROOT** is not explicitly specified.

OpenBSD:

- **ENV{LOCALBASE}**

Unix:

- **ENV{CONDA_PREFIX}** when using a conda compiler

MSYSTEM environment with MinGW toolchain:
Added in version 3.28.

- **ENV{MSYSTEM_PREFIX}/local**
- **ENV{MSYSTEM_PREFIX}**

Windows:

- **ENV{ProgramW6432}**
- **ENV{ProgramFiles}**
- **ENV{ProgramFiles(x86)}**
- **ENV{SystemDrive}/Program Files**
- **ENV{SystemDrive}/Program Files (x86)**

CMAKE_SYSTEM_PREFIX_PATH 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
Semicolon−separated list of directories specifying a search path for the *find_program()* command. By default this 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_TLS_CAINFO
Specify the default value for the *file(DOWNLOAD)* and *file(UPLOAD)* commands' **TLS_CAINFO** options. It is unset by default.

This variable is also used by the *ExternalProject* and *FetchContent* modules for internal calls to *file(DOWNLOAD)*.

CMAKE_TLS_VERIFY
Specify the default value for the *file(DOWNLOAD)* and *file(UPLOAD)* commands' **TLS_VERIFY** options. If this variable is not set, the commands check the *CMAKE_TLS_VERIFY* environment variable. If neither is set, the default is *on*.

Changed in version 3.31: The default is on. Previously, the default was off. Users may set the *CMAKE_TLS_VERIFY* environment variable to **0** to restore the old default.

This variable is also used by the *ExternalProject* and *FetchContent* modules for internal calls to *file(DOWNLOAD)*.

TLS verification can help provide confidence that one is connecting to the desired server. When downloading known content, one should also use file hashes to verify it.

```
set(CMAKE_TLS_VERIFY TRUE)
```

CMAKE_TLS_VERSION
Added in version 3.30.

Specify the default value for the *file(DOWNLOAD)* and *file(UPLOAD)* commands' **TLS_VERSION** option. If this variable is not set, the commands check the *CMAKE_TLS_VERSION* environment variable. If neither is set, the default is TLS 1.2.

Changed in version 3.31: The default is TLS 1.2. Previously, no minimum version was enforced by default.

The value may be one of:

- **1.0**
- **1.1**
- **1.2**
- **1.3**

This variable is also used by the *ExternalProject* and *FetchContent* modules for internal calls to *file(DOWNLOAD)* and **git clone**.

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. See*CMAKE_USER_MAKE_R ULES_OVERRIDE_<LANG>* for the language−specific version of this variable.

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 the *CMAKE_C_FLAGS_INIT* variable to change the default value used to initialize the *CMAKE_C_FLAGS* variable 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 warnings for deprecated functionality.

If not **FALSE**, use of deprecated functionality will issue warnings. If this variable is not set, CMake behaves as if it were set to **TRUE**.

When running *cmake(1)*, this option can be enabled with the *−Wdeprecated* option, or disabled with the *−Wno−deprecated* option.

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.

CMAKE_XCODE_GENERATE_SCHEME
Added in version 3.9.

If enabled, the *Xcode* generator will generate schema files. These are useful to invoke analyze, archive, build−for−testing and test actions from the command line.

This variable initializes the *XCODE_GENERATE_SCHEME* target property on all targets.

CMAKE_XCODE_GENERATE_TOP_LEVEL_PROJECT_ONLY
Added in version 3.11.

If enabled, the *Xcode* generator will generate only a single Xcode project file for the topmost *project()* command instead of generating one for every **project()** command.

This could be useful to speed up the CMake generation step for large projects and to work−around a bug in the **ZERO_CHECK** logic.

CMAKE_XCODE_LINK_BUILD_PHASE_MODE
Added in version 3.19.

This variable is used to initialize the *XCODE_LINK_BUILD_PHASE_MODE* property on targets. It affects the methods that the *Xcode* generator uses to link different kinds of libraries. Its default value is **NONE**.

CMAKE_XCODE_SCHEME_ADDRESS_SANITIZER
Added in version 3.13.

Whether to enable **Address Sanitizer** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_ADDRESS_SANITIZER* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_ADDRESS_SANITIZER_USE_AFTER_RETURN
Added in version 3.13.

Whether to enable **Detect use of stack after return** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_ADDRESS_SANITIZER_USE_AFTER_RETURN* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_DEBUG_DOCUMENT_VERSIONING
Added in version 3.16.

Whether to enable **Allow debugging when using document Versions Browser** in the Options section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_DEBUG_DOCUMENT_VERSIONING* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_DISABLE_MAIN_THREAD_CHECKER
Added in version 3.13.

Whether to disable the **Main Thread Checker** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_DISABLE_MAIN_THREAD_CHECKER* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_DYNAMIC_LIBRARY_LOADS
Added in version 3.13.

Whether to enable **Dynamic Library Loads** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_DYNAMIC_LIBRARY_LOADS* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_DYNAMIC_LINKER_API_USAGE
Added in version 3.13.

Whether to enable **Dynamic Linker API usage** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_DYNAMIC_LINKER_API_USAGE* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_ENABLE_GPU_API_VALIDATION
Added in version 3.25.

Property value for **Metal: API Validation** in the Options section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_ENABLE_GPU_API_VALIDATION* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_ENABLE_GPU_FRAME_CAPTURE_MODE
Added in version 3.23.

Property value for **GPU Frame Capture** in the Options section of the generated Xcode scheme. Example values are **Metal** and **Disabled**.

This variable initializes the *XCODE_SCHEME_ENABLE_GPU_FRAME_CAPTURE_MODE* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_ENABLE_GPU_SHADER_VALIDATION
Added in version 3.25.

Property value for **Metal: Shader Validation** in the Options section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_ENABLE_GPU_SHADER_VALIDATION* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_ENVIRONMENT
Added in version 3.17.

Specify environment variables that should be added to the Arguments section of the generated Xcode scheme.

If set to a list of environment variables and values of the form **MYVAR=value** those environment variables will be added to the scheme.

This variable initializes the *XCODE_SCHEME_ENVIRONMENT* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_GUARD_MALLOC
Added in version 3.13.

Whether to enable **Guard Malloc** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_GUARD_MALLOC* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_LAUNCH_CONFIGURATION
Added in version 3.25.

Set the build configuration to run the target.

This variable initializes the *XCODE_SCHEME_LAUNCH_CONFIGURATION* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_TEST_CONFIGURATION
Added in version 4.0.

Set the build configuration to test the target.

This variable initializes the *XCODE_SCHEME_TEST_CONFIGURATION* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_LAUNCH_MODE
Added in version 3.25.

Property value for **Launch** in the Info section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_LAUNCH_MODE* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_LLDB_INIT_FILE
Added in version 4.0.

Property value for **LLDB Init File** in the Info section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_LLDB_INIT_FILE* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_MAIN_THREAD_CHECKER_STOP
Added in version 3.13.

Whether to enable the **Main Thread Checker** option **Pause on issues** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_MAIN_THREAD_CHECKER_STOP* property on all

targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_MALLOC_GUARD_EDGES
Added in version 3.13.

Whether to enable **Malloc Guard Edges** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_MALLOC_GUARD_EDGES* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_MALLOC_SCRIBBLE
Added in version 3.13.

Whether to enable **Malloc Scribble** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_MALLOC_SCRIBBLE* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_MALLOC_STACK
Added in version 3.13.

Whether to enable **Malloc Stack** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_MALLOC_STACK* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_THREAD_SANITIZER
Added in version 3.13.

Whether to enable **Thread Sanitizer** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_THREAD_SANITIZER* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_THREAD_SANITIZER_STOP
Added in version 3.13.

Whether to enable **Thread Sanitizer − Pause on issues** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_THREAD_SANITIZER_STOP* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_UNDEFINED_BEHAVIOUR_SANITIZER
Added in version 3.13.

Whether to enable **Undefined Behavior Sanitizer** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_UNDEFINED_BEHAVIOUR_SANITIZER* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_UNDEFINED_BEHAVIOUR_SANITIZER_STOP
Added in version 3.13.

Whether to enable **Undefined Behavior Sanitizer** option **Pause on issues** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_UNDEFINED_BEHAVIOUR_SANITIZER_STOP* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_WORKING_DIRECTORY
Added in version 3.17.

Specify the **Working Directory** of the *Run* and *Profile* actions in the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_WORKING_DIRECTORY* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_SCHEME_ZOMBIE_OBJECTS
Added in version 3.13.

Whether to enable **Zombie Objects** in the Diagnostics section of the generated Xcode scheme.

This variable initializes the *XCODE_SCHEME_ZOMBIE_OBJECTS* property on all targets.

Please refer to the *XCODE_GENERATE_SCHEME* target property documentation to see all Xcode schema related properties.

CMAKE_XCODE_XCCONFIG
Added in version 3.24.

If set, the *Xcode* generator will register the specified file as a global XCConfig file. For target−level XCConfig files see the *XCODE_XCCONFIG* target property.

This feature is intended to ease migration from native Xcode projects to CMake projects.

Contents of **CMAKE_XCODE_XCCONFIG** may use *generator expressions*.

<PackageName>_ROOT
Added in version 3.12.

Calls to *find_package(<PackageName>)* will search in prefixes specified by the **<PackageName>_ROOT** CMake variable, where **<PackageName>** is the (case−preserved) name given to the *find_package()* call and **_ROOT** is literal. For example, **find_package(Foo)** will search prefixes specified in the **Foo_ROOT** CMake variable (if set). See policy *CMP0074*.

This variable may hold a single prefix or a *semicolon−separated list* of multiple prefixes.

See also the *<PackageName>_ROOT* environment variable.

<PACKAGENAME>_ROOT
Added in version 3.27.

Calls to *find_package(<PackageName>)* will also search in prefixes specified by the upper−case **<PACKAGENAME>_ROOT** CMake variable. See policy *CMP0144*.

VARIABLES THAT DESCRIBE THE SYSTEM

AIX
Added in version 4.0.

Set to true when the target system is AIX.

ANDROID
Added in version 3.7.

Set to **1** when the target system (*CMAKE_SYSTEM_NAME*) is **Android**.

APPLE
Set to **True** when the target system is an Apple platform (macOS, iOS, tvOS, visionOS or watchOS).

BORLAND
True if the Borland compiler is being used.

This is set to **true** if the Borland compiler is being used.

BSD
Added in version 3.25.

Set to a string value when the target system is BSD. This value can be one of the following: DragonFly-BSD, FreeBSD, OpenBSD, or NetBSD.

CMAKE_ANDROID_NDK_VERSION
Added in version 3.20.

When *Cross Compiling for Android with the NDK* and using an Android NDK version 11 or higher, this variable is provided by CMake to report the NDK version number.

CMAKE_CL_64
Discouraged. Use *CMAKE_SIZEOF_VOID_P* instead.

Set to a true value when using a Microsoft Visual Studio **cl** compiler that *targets* a 64−bit architecture.

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 macOS operating systems.

Set to **true** when the host system is Apple macOS.

CMAKE_HOST_AIX
Added in version 4.0.

Set to true when the host system is AIX.

CMAKE_HOST_BSD
Added in version 3.25.

Set to a string value when the host system is BSD. This value can be one of the following: DragonFlyBSD, FreeBSD, OpenBSD, or NetBSD.

CMAKE_HOST_EXECUTABLE_SUFFIX
Added in version 3.31.

The suffix for executables on the host platform. This may differ from the suffix for the target platform, *CMAKE_EXECUTABLE_SUFFIX*.

The suffix to use for the end of an executable filename if any, **.exe** on Windows.

See also *CMAKE_EXECUTABLE_SUFFIX*.

CMAKE_HOST_LINUX
Added in version 3.25.

Set to true when the host system is Linux.

CMAKE_HOST_SOLARIS
Added in version 3.6.

True for Oracle Solaris operating systems.

Set to **true** when the host system is Oracle Solaris.

CMAKE_HOST_SYSTEM
Composite Name of OS CMake is being run on.

This variable is the composite of *CMAKE_HOST_SYSTEM_NAME* and *CMAKE_HOST_SYSTEM_VERSION*, e.g. **${CMAKE_HOST_SYSTEM_NAME}−${CMAKE_HOST_SYSTEM_VERSION}**. If*CMAKE_HOST_SYSTEM_VERSION* is

not set, then this variable is the same as *CMAKE_HOST_SYSTEM_NAME*.

CMAKE_HOST_SYSTEM_NAME
Name of the OS CMake is running on.

On systems that have the uname command, this variable is set to the output of **uname −s**. **Linux**, **Windows**, and **Darwin** for macOS are the values found on the big three operating systems.

For a list of possible values, see *CMAKE_SYSTEM_NAME*.

CMAKE_HOST_SYSTEM_PROCESSOR
The name of the CPU CMake is running on.

Windows Platforms
On Windows, this variable is set to the value of the environment variable **PROCESSOR_ARCHITECTURE**.

Unix Platforms
On systems that support **uname**, this variable is set to the output of:

- **uname −m** on GNU, Linux, Cygwin, Android, or
- **arch** on OpenBSD, or
- on other systems,
  - **uname −p** if its exit code is nonzero, or
  - **uname −m** otherwise.

macOS Platforms
The value of **uname −m** is used by default.

On Apple Silicon hosts, the architecture printed by **uname −m** may vary based on CMake's own architecture and that of the invoking process tree.

Added in version 3.19.2: On Apple Silicon hosts:

- The *CMAKE_APPLE_SILICON_PROCESSOR* variable or the *CMAKE_APPLE_SILICON_PROCESSOR* environment variable may be set to specify the host architecture explicitly.
- If *CMAKE_OSX_ARCHITECTURES* is not set, CMake adds explicit flags to tell the compiler to build for the host architecture so the toolchain does not have to guess based on the process tree's architecture.

CMAKE_HOST_SYSTEM_VERSION
The OS version CMake is running on.

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.

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 if the host system is running Windows, including Windows 64−bit and MSYS.

Set to **false** 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
Composite name of operating system CMake is compiling for.

This variable is the composite of *CMAKE_SYSTEM_NAME* and *CMAKE_SYSTEM_VERSION*, e.g. **${CMAKE_SYSTEM_NAME}−${CMAKE_SYSTEM_VERSION}**. If*CMAKE_SYSTEM_VERSION* is not set, then this variable is the same as *CMAKE_SYSTEM_NAME*.

CMAKE_SYSTEM_NAME
The name of the operating system for which CMake is to build. See the*CMAKE_SYSTEM_VERSION* v ariable for the OS version.

Note that **CMAKE_SYSTEM_NAME** is not set to anything by default when running in script mode, since it's not building anything.

System Name for Host Builds
CMAKE_SYSTEM_NAME is by default set to the same value as the *CMAKE_HOST_SYSTEM_NAME* variable so that the build targets the host system.

System Name for Cross Compiling
CMAKE_SYSTEM_NAME may be set explicitly when first configuring a new build tree in order to enable *cross compiling*. In this case the*CMAKE_SYSTEM_VERSION* v ariable must also be set explicitly.

System Names Known to CMake
The following is a list of possible values, each associated with corresponding operating systems or environments.

Platform−specific notes:

- MSYS2's **msys/cmake** package (**/usr/bin/cmake**) works only under **MSYSTEM=MSYS** environments, with system name **MSYS**. Under other environments like **MSYSTEM=MINGW64**, use another package such as **mingw64/mingw−w64−x86_64−cmake** (**/mingw64/bin/cmake**), which targets **MSYSTEM=MINGW64** with system name **Windows**.
- Cygwin's **cmake** package (**/usr/bin/cmake**) uses system name **CYGWIN**. A non−cygwin CMake on Windows (e.g. **$PROGRAMFILES/CMake/bin/cmake**) uses system name **Windows** even when it runs under a Cygwin environment.

CMAKE_SYSTEM_PROCESSOR
When not cross−compiling, this variable has the same value as the *CMAKE_HOST_SYSTEM_PROCESSOR* variable. In many cases, this will correspond to the target architecture for the build, but this is not guaranteed. (E.g. on Windows, the host may be **AMD64** even when using a MSVC **cl** compiler with a 32−bit target.)

When cross−compiling, a *CMAKE_TOOLCHAIN_FILE* should set the **CMAKE_SYSTEM_PROCESSOR** variable to match target architecture that it specifies (via *CMAKE_<LANG>_COMPILER* and perhaps *CMAKE_<LANG>_COMPILER_TARGET*).

CMAKE_SYSTEM_VERSION
The version of the operating system for which CMake is to build. See the*CMAKE_SYSTEM_N AME* variable for the OS name.

System Version for Host Builds
When the *CMAKE_SYSTEM_NAME* variable takes its default value then **CMAKE_SYSTEM_VERSION** is by default set to the same value as the *CMAKE_HOST_SYSTEM_VERSION* variable so that the build targets the host system version.

In the case of a host build then **CMAKE_SYSTEM_VERSION** may be set explicitly when first configuring a new build tree in order to enable targeting the build for a different version of the host operating system than is actually running on the host. This is allowed and not considered cross compiling so long as the binaries built for the specified OS version can still run on the host.

System Version for Cross Compiling
When the *CMAKE_SYSTEM_NAME* variable is set explicitly to enable *cross compiling* then the value of **CMAKE_SYSTEM_VERSION** must also be set explicitly to specify the target system version.

CYGWIN
True for Cygwin.

Set to **true** when using Cygwin.

GHSMULTI
Added in version 3.3.

1 when using *Green Hills MULTI* generator.

Also, Set to **1** when the target system is a Green Hills platform (i.e. When *CMAKE_SYSTEM_NAME* is **GHS−MULTI**).

IOS
Added in version 3.14.

Set to **1** when the target system (*CMAKE_SYSTEM_NAME*) is **iOS**.

LINUX
Added in version 3.25.

Set to true when the target system is Linux.

MINGW
Added in version 3.2.

Set to a true value when at least one language is enabled with a compiler targeting the GNU ABI on Windows (MinGW).

Otherwise, this variable is not set by CMake.

MSVC
Set to **true** when the compiler is some version of Microsoft Visual C++ or another compiler simulating the Visual C++ **cl** command−line syntax.

See also the *MSVC_VERSION* variable.

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.

NOTE:
This variable is only available after compiler detection has been performed, so it is not available to toolchain files or before the first *project()* or *enable_language()* call which uses an MSVC−like compiler.

MSVC_TOOLSET_VERSION
Added in version 3.12.

The toolset version of Microsoft Visual C/C++ being used if any. If MSVC−like is being used, this variable is set based on the version of the compiler as given by the *MSVC_VERSION* variable.

Known toolset version numbers are:

Value	Version
80	VS 2005 (8.0)
90	VS 2008 (9.0)
100	VS 2010 (10.0)
110	VS 2012 (11.0)
120	VS 2013 (12.0)
140	VS 2015 (14.0)
141	VS 2017 (15.0)
142	VS 2019 (16.0)
143	VS 2022 (17.0)

Compiler versions newer than those known to CMake will be reported as the latest known toolset version.

See also the *MSVC_VERSION* variable.

MSVC_VERSION
The version of Microsoft Visual C/C++ being used if any. If a compiler simulating Visual C++ is being used, this variable is set to the toolset version simulated as given by the **_MSC_VER** preprocessor definition.

Known version numbers are:

Value	Version
1200	VS 6.0
1300	VS 7.0
1310	VS 7.1
1400	VS 8.0 (v80 toolset)
1500	VS 9.0 (v90 toolset)
1600	VS 10.0 (v100 toolset)
1700	VS 11.0 (v110 toolset)
1800	VS 12.0 (v120 toolset)
1900	VS 14.0 (v140 toolset)
1910−1919	VS 15.0 (v141 toolset)
1920−1929	VS 16.0 (v142 toolset)
1930−1949	VS 17.0 (v143 toolset)

See also the *CMAKE_<LANG>_COMPILER_VERSION* and *MSVC_TOOLSET_VERSION* variable.

MSYS
Added in version 3.14.

True when using the *MSYS Makefiles* generator.

UNIX
Set to **True** when the target system is UNIX or UNIX−like (e.g. *APPLE* and *CYGWIN*). The *CMAKE_SYSTEM_NAME* variable should be queried if a more specific understanding of the target system is required.

WASI
Added in version 3.31.

Set to **1** when the target system is WebAssembly System Interface (*CMAKE_SYSTEM_NAME* is **WASI**).

WIN32
Set to **True** when the target system is Windows, including Win64.

WINCE
Added in version 3.1.

True when the *CMAKE_SYSTEM_NAME* variable is set to **WindowsCE**.

WINDOWS_PHONE
Added in version 3.1.

True when the *CMAKE_SYSTEM_NAME* variable is set to **WindowsPhone**.

WINDOWS_STORE
Added in version 3.1.

True when the *CMAKE_SYSTEM_NAME* variable is set to **WindowsStore**.

XCODE
Added in version 3.7.

True when using *Xcode* generator.

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 THAT CONTROL THE BUILD

CMAKE_ADSP_ROOT
Added in version 3.24.

When *Cross Compiling for ADSP SHARC/Blackfin*, this variable holds the absolute path to the latest CCES or VDSP++ install. The directory is expected to contain the **cc21k.exe** and **ccblkfn.exe** compilers. This will be set automatically if a default install of CCES or VDSP++ can be found.

See also the *ADSP_ROOT* environment variable.

CMAKE_AIX_SHARED_LIBRARY_ARCHIVE
Added in version 3.31.

On AIX, enable or disable creation of shared library archives.

This variable initializes the *AIX_SHARED_LIBRARY_ARCHIVE* target property on non−imported **SHARED** library targets as they are created by *add_library()*. See that target property for details.

CMAKE_AIX_EXPORT_ALL_SYMBOLS
Added in version 3.17.

Default value for *AIX_EXPORT_ALL_SYMBOLS* target property. This variable is used to initialize the property on each target as it is created.

CMAKE_ANDROID_ANT_ADDITIONAL_OPTIONS
Added in version 3.4.

Default value for the *ANDROID_ANT_ADDITIONAL_OPTIONS* target property. See that target property for additional information.

CMAKE_ANDROID_API
Added in version 3.1.

When *Cross Compiling for Android with NVIDIA Nsight Tegra Visual Studio Edition*, this variable may be set to specify the default value for the *ANDROID_API* target property. See that target property for additional information.

When *Cross Compiling for Android*, the *CMAKE_SYSTEM_VERSION* variable represents the Android API version number targeted. For historical reasons, if a toolchain file sets **CMAKE_ANDROID_API**, but not **CMAKE_SYSTEM_VERSION**, the latter will be initialized using the former.

CMAKE_ANDROID_API_MIN
Added in version 3.2.

Default value for the *ANDROID_API_MIN* target property. See that target property for additional information.

CMAKE_ANDROID_ARCH
Added in version 3.4.

When *Cross Compiling for Android with NVIDIA Nsight Tegra Visual Studio Edition*, this variable may be set to specify the default value for the *ANDROID_ARCH* target property. See that target property for additional information.

Otherwise, when *Cross Compiling for Android*, this variable provides the name of the Android architecture corresponding to the value of the *CMAKE_ANDROID_ARCH_ABI* variable. The architecture name may be one of:

- **arm**
- **arm64**
- **mips**
- **mips64**
- **x86**
- **x86_64**

CMAKE_ANDROID_ARCH_ABI
Added in version 3.7.

When *Cross Compiling for Android*, this variable specifies the target architecture and ABI to be used. Valid values are:

- **arm64−v8a**
- **armeabi−v7a**
- **armeabi−v6**
- **armeabi**
- **mips**
- **mips64**
- **x86**
- **x86_64**

See also the *CMAKE_ANDROID_ARM_MODE* and *CMAKE_ANDROID_ARM_NEON* variables.

CMAKE_ANDROID_ARM_MODE
Added in version 3.7.

When *Cross Compiling for Android* and *CMAKE_ANDROID_ARCH_ABI* is set to one of the **armeabi** architectures, set **CMAKE_ANDROID_ARM_MODE** to **ON** to target 32−bit ARM processors (**−marm**). Otherwise, the default is to target the 16−bit Thumb processors (**−mthumb**).

CMAKE_ANDROID_ARM_NEON
Added in version 3.7.

When *Cross Compiling for Android* and *CMAKE_ANDROID_ARCH_ABI* is set to **armeabi-v7a** set **CMAKE_ANDROID_ARM_NEON** to **ON** to target ARM NEON devices.

CMAKE_ANDROID_ASSETS_DIRECTORIES
Added in version 3.4.

Default value for the *ANDROID_ASSETS_DIRECTORIES* target property. See that target property for additional information.

CMAKE_ANDROID_EXCEPTIONS
Added in version 3.20.

When *Cross Compiling for Android with the NDK*, this variable may be set to specify whether exceptions are enabled.

CMAKE_ANDROID_GUI
Added in version 3.1.

Default value for the *ANDROID_GUI* target property of executables. See that target property for additional information.

CMAKE_ANDROID_JAR_DEPENDENCIES
Added in version 3.4.

Default value for the *ANDROID_JAR_DEPENDENCIES* target property. See that target property for additional information.

CMAKE_ANDROID_JAR_DIRECTORIES
Added in version 3.4.

Default value for the *ANDROID_JAR_DIRECTORIES* target property. See that target property for additional information.

CMAKE_ANDROID_JAVA_SOURCE_DIR
Added in version 3.4.

Default value for the *ANDROID_JAVA_SOURCE_DIR* target property. See that target property for additional information.

CMAKE_ANDROID_NATIVE_LIB_DEPENDENCIES
Added in version 3.4.

Default value for the *ANDROID_NATIVE_LIB_DEPENDENCIES* target property. See that target property for additional information.

CMAKE_ANDROID_NATIVE_LIB_DIRECTORIES
Added in version 3.4.

Default value for the *ANDROID_NATIVE_LIB_DIRECTORIES* target property. See that target property for additional information.

CMAKE_ANDROID_NDK
Added in version 3.7.

When *Cross Compiling for Android with the NDK*, this variable holds the absolute path to the root directory of the NDK. The directory must contain a **platforms** subdirectory holding the **android-<api>** directories.

CMAKE_ANDROID_NDK_DEPRECATED_HEADERS
Added in version 3.9.

When *Cross Compiling for Android with the NDK*, this variable may be set to specify whether to use the deprecated per-api-level headers instead of the unified headers.

If not specified, the default will be *false* if using a NDK version that provides the unified headers and *true* otherwise.

CMAKE_ANDROID_NDK_TOOLCHAIN_HOST_TAG
Added in version 3.7.1.

When *Cross Compiling for Android with the NDK*, this variable provides the NDK's "host tag" used to construct the path to prebuilt toolchains that run on the host.

CMAKE_ANDROID_NDK_TOOLCHAIN_VERSION
Added in version 3.7.

When *Cross Compiling for Android with the NDK*, this variable may be set to specify the version of the toolchain to be used as the compiler.

On NDK r19 or above, this variable must be unset or set to **clang**.

On NDK r18 or below, this variable must be set to one of these forms:

- **<major>.<minor>**: GCC of specified version
- **clang<major>.<minor>**: Clang of specified version
- **clang**: Clang of most recent available version

A toolchain of the requested version will be selected automatically to match the ABI named in the *CMAKE_ANDROID_ARCH_ABI* variable.

If not specified, the default will be a value that selects the latest available GCC toolchain.

CMAKE_ANDROID_PROCESS_MAX
Added in version 3.4.

Default value for the *ANDROID_PROCESS_MAX* target property. See that target property for additional information.

CMAKE_ANDROID_PROGUARD
Added in version 3.4.

Default value for the *ANDROID_PROGUARD* target property. See that target property for additional information.

CMAKE_ANDROID_PROGUARD_CONFIG_PATH
Added in version 3.4.

Default value for the *ANDROID_PROGUARD_CONFIG_PATH* target property. See that target property for additional information.

CMAKE_ANDROID_RTTI
Added in version 3.20.

When *Cross Compiling for Android with the NDK*, this variable may be set to specify whether RTTI is enabled.

CMAKE_ANDROID_SECURE_PROPS_PATH
Added in version 3.4.

Default value for the *ANDROID_SECURE_PROPS_PATH* target property. See that target property for additional information.

CMAKE_ANDROID_SKIP_ANT_STEP
Added in version 3.4.

Default value for the *ANDROID_SKIP_ANT_STEP* target property. See that target property for additional information.

CMAKE_ANDROID_STANDALONE_TOOLCHAIN
Added in version 3.7.

When *Cross Compiling for Android with a Standalone Toolchain*, this variable holds the absolute path to the root directory of the toolchain. The specified directory must contain a **sysroot** subdirectory.

CMAKE_ANDROID_STL_TYPE
Added in version 3.4.

When *Cross Compiling for Android with NVIDIA Nsight Tegra Visual Studio Edition*, this variable may be set to specify the default value for the *ANDROID_STL_TYPE* target property. See that target property for additional information.

When *Cross Compiling for Android with the NDK*, this variable may be set to specify the STL variant to be used. The value may be one of:

none No C++ Support

system Minimal C++ without STL

gabi++_static
GAbi++ Static

gabi++_shared
GAbi++ Shared

gnustl_static
GNU libstdc++ Static

gnustl_shared
GNU libstdc++ Shared

c++_static
LLVM libc++ Static

c++_shared
LLVM libc++ Shared

stlport_static
STLport Static

stlport_shared
STLport Shared

The default value is **gnustl_static** on NDK versions that provide it and otherwise **c++_static**. Note that this default differs from the native NDK build system because CMake may be used to build projects for Android that are not natively implemented for it and use the C++ standard library.

CMAKE_APPLE_SILICON_PROCESSOR
Added in version 3.19.2.

On Apple Silicon hosts running macOS, set this variable to tell CMake what architecture to use for *CMAKE_HOST_SYSTEM_PROCESSOR*. The value must be either **arm64** or **x86_64**.

The value of this variable should never be modified by project code. It is meant to be set as a cache entry provided by the user, e.g. via **-DCMAKE_APPLE_SILICON_PROCESSOR=...**.

See also the *CMAKE_APPLE_SILICON_PROCESSOR* environment variable.

CMAKE_ARCHIVE_OUTPUT_DIRECTORY
Where to put all the *ARCHIVE* target files 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_ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>
Added in version 3.3.

Where to put all the *ARCHIVE* target files when built for a specific configuration.

This variable is used to initialize the *ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>* property on all the targets. See that target property for additional information.

CMAKE_AUTOGEN_BETTER_GRAPH_MULTI_CONFIG
Added in version 3.29.

This variable is used to initialize the *AUTOGEN_BETTER_GRAPH_MULTI_CONFIG* property on all targets as they are created. See that target property for additional information.

By default **CMAKE_AUTOGEN_BETTER_GRAPH_MULTI_CONFIG** is unset.

CMAKE_AUTOGEN_COMMAND_LINE_LENGTH_MAX
Added in version 3.29.

Command line length limit for autogen targets, i.e. **moc** or **uic**, that triggers the use of response files on Windows instead of passing all arguments to the command line.

By default **CMAKE_AUTOGEN_COMMAND_LINE_LENGTH_MAX** is unset.

CMAKE_AUTOGEN_ORIGIN_DEPENDS
Added in version 3.14.

Switch for forwarding origin target dependencies to the corresponding *The <ORIGIN>_autogen target* targets.

NOTE:
If Qt 5.15 or later is used and the generator is either *Ninja* or *Makefile Generators*, additional target dependencies are added to the *The <ORIGIN>_autogen_timestamp_deps target* target instead of the *The <ORIGIN>_autogen target* target.

This variable is used to initialize the *AUTOGEN_ORIGIN_DEPENDS* property on all the targets. See that target property for additional information.

By default **CMAKE_AUTOGEN_ORIGIN_DEPENDS** is **ON**.

CMAKE_AUTOGEN_PARALLEL
Added in version 3.11.

Number of parallel **moc** or **uic** processes to start when using *AUTOMOC* and *AUTOUIC*.

This variable is used to initialize the *AUTOGEN_PARALLEL* property on all the targets. See that target property for additional information.

By default **CMAKE_AUTOGEN_PARALLEL** is unset.

CMAKE_AUTOGEN_USE_SYSTEM_INCLUDE
Added in version 3.27.

This variable is used to initialize the *AUTOGEN_USE_SYSTEM_INCLUDE* property on all targets as they are created. See that target property for additional information.

By default **CMAKE_AUTOGEN_USE_SYSTEM_INCLUDE** is unset.

CMAKE_AUTOGEN_VERBOSE
Added in version 3.13.

Sets the verbosity of *AUTOMOC*, *AUTOUIC* and *AUTORCC*. A positive integer value or a true boolean value lets the **AUTO*** generators output additional processing information.

Setting **CMAKE_AUTOGEN_VERBOSE** has the same effect as setting the **VERBOSE** environment variable during generation (e.g. by calling **make VERBOSE=1**). The extra verbosity is limited to the **AUTO*** generators though.

By default **CMAKE_AUTOGEN_VERBOSE** is unset.

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_COMPILER_PREDEFINES
Added in version 3.10.

This variable is used to initialize the *AUTOMOC_COMPILER_PREDEFINES* property on all the targets. See that target property for additional information.

By default it is ON.

CMAKE_AUTOMOC_DEPEND_FILTERS
Added in version 3.9.

Filter definitions used by *CMAKE_AUTOMOC* to extract file names from source code as additional dependencies for the **moc** file.

This variable is used to initialize the *AUTOMOC_DEPEND_FILTERS* property on all the targets. See that target property for additional information.

By default it is empty.

CMAKE_AUTOMOC_MACRO_NAMES
Added in version 3.10.

Semicolon−separated list list of macro names used by *CMAKE_AUTOMOC* to determine if a C++ file needs to be processed by **moc**.

This variable is used to initialize the *AUTOMOC_MACRO_NAMES* property on all the targets. See that target property for additional information.

The default value is **Q_OBJECT;Q_GADGET;Q_NAMESPACE;Q_NAMESPACE_EXPORT**.

Example
Let CMake know that source files that contain **CUSTOM_MACRO** must be **moc** processed as well:

```
set(CMAKE_AUTOMOC ON)
list(APPEND CMAKE_AUTOMOC_MACRO_NAMES "CUSTOM_MACRO")
```

CMAKE_AUTOMOC_MOC_OPTIONS
Additional options for **moc** when using *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_AUTOMOC_PATH_PREFIX
Added in version 3.16.

Whether to generate the **−p** path prefix option for **moc** on *AUTOMOC* enabled Qt targets.

This variable is used to initialize the *AUTOMOC_PATH_PREFIX* property on all the targets. See that target property for additional information.

The default value is **OFF**.

CMAKE_AUTOMOC_EXECUTABLE
Added in version 3.27.

This variable is used to initialize the *AUTOMOC_EXECUTABLE* property on all the targets. See that target property for additional information.

By default it is empty.

CMAKE_AUTORCC
Whether to handle **rcc** automatically for Qt targets.

This variable is used to initialize the *AUTORCC* property on all the targets. See that target property for additional information.

CMAKE_AUTORCC_OPTIONS
Additional options for **rcc** when using *CMAKE_AUTORCC*.

This variable is used to initialize the *AUTORCC_OPTIONS* property on all the targets. See that target property for additional information.

EXAMPLE

```
# ...
set(CMAKE_AUTORCC_OPTIONS "--compress;9")
# ...
```

CMAKE_AUTORCC_EXECUTABLE
Added in version 3.27.

This variable is used to initialize the *AUTORCC_EXECUTABLE* property on all the targets. See that target property for additional information.

By default it is empty.

CMAKE_AUTOUIC
Whether to handle **uic** automatically for Qt targets.

This variable is used to initialize the *AUTOUIC* property on all the targets. See that target property for additional information.

CMAKE_AUTOUIC_OPTIONS
Additional options for **uic** when using *CMAKE_AUTOUIC*.

This variable is used to initialize the *AUTOUIC_OPTIONS* property on all the targets. See that target property for additional information.

EXAMPLE

```
# ...
set_property(CMAKE_AUTOUIC_OPTIONS "--no-protection")
# ...
```

CMAKE_AUTOUIC_SEARCH_PATHS
Added in version 3.9.

Search path list used by *CMAKE_AUTOUIC* to find included **.ui** files.

This variable is used to initialize the *AUTOUIC_SEARCH_PATHS* property on all the targets. See that target property for additional information.

By default it is empty.

CMAKE_AUTOUIC_EXECUTABLE
Added in version 3.27.

This variable is used to initialize the *AUTOUIC_EXECUTABLE* property on all the targets. See that target property for additional information.

By default it is empty.

CMAKE_BUILD_RPATH
Added in version 3.8.

Semicolon−separated list specifying runtime path (**RPATH**) entries to add to binaries linked in the build tree (for platforms that support it). The entries will *not* be used for binaries in the install tree. See also the *CMAKE_INSTALL_RPATH* variable.

This is used to initialize the *BUILD_RPATH* target property for all targets.

CMAKE_BUILD_RPATH_USE_ORIGIN
Added in version 3.14.

Whether to use relative paths for the build **RPATH**.

This is used to initialize the *BUILD_RPATH_USE_ORIGIN* target property for all targets, see that property for more details.

CMAKE_BUILD_WITH_INSTALL_NAME_DIR
Added in version 3.9.

Whether to use *INSTALL_NAME_DIR* on targets in the build tree.

This variable is used to initialize the *BUILD_WITH_INSTALL_NAME_DIR* property on all targets.

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.

This is used to initialize the *BUILD_WITH_INSTALL_RPATH* target property for all targets.

CMAKE_COMPILE_PDB_OUTPUT_DIRECTORY
Added in version 3.1.

Output directory for MS debug symbol **.pdb** files generated by the compiler while building source files.

This variable is used to initialize the *COMPILE_PDB_OUTPUT_DIRECTORY* property on all the targets.

CMAKE_COMPILE_PDB_OUTPUT_DIRECTORY_<CONFIG>
Added in version 3.1.

Per−configuration output directory for MS debug symbol **.pdb** files generated by the compiler while building source files.

This is a per−configuration version of *CMAKE_COMPILE_PDB_OUTPUT_DIRECTORY*. This variable is used to initialize the *COMPILE_PDB_OUTPUT_DIRECTORY_<CONFIG>* property on all the targets.

CMAKE_COMPILE_WARNING_AS_ERROR
Added in version 3.24.

Specify whether to treat warnings on compile as errors.

This variable is used to initialize the *COMPILE_WARNING_AS_ERROR* property on all the targets.

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_CROSS_CONFIGS
Added in version 3.17.

Specifies a *semicolon−separated list* of configurations available from all **build−<Config>.ninja** files in the *Ninja Multi−Config* generator. This variable activates cross−config mode. Targets from each config specified in this variable can be built from any **build−<Config>.ninja** file. Custom commands will use the configuration native to **build−<Config>.ninja**. If it is set to **all**, all configurations from *CMAKE_CONFIGURATION_TYPES* are cross−configs. If it is not specified, or empty, each **build−<Config>.ninja** file will only contain build rules for its own configuration.

The value of this variable must be a subset of *CMAKE_CONFIGURATION_TYPES*.

CMAKE_CTEST_ARGUMENTS
Added in version 3.17.

Set this to a *semicolon−separated list* of command−line arguments to pass to *ctest(1)* when running tests through the **test** (or **RUN_TESTS**) target of the generated build system.

CMAKE_CUDA_RESOLVE_DEVICE_SYMBOLS
Added in version 3.16.

Default value for *CUDA_RESOLVE_DEVICE_SYMBOLS* target property when defined. By default this variable is not defined.

This variable is used to initialize the property on each target as it is created.

CMAKE_CUDA_RUNTIME_LIBRARY
Added in version 3.17.

Select the CUDA runtime library for use when compiling and linking CUDA. This variable is used to initialize the *CUDA_RUNTIME_LIBRARY* property on all targets as they are created.

The allowed case insensitive values are:

None Link with **−cudart=none** or equivalent flag(s) to use no CUDA runtime library.

Shared Link with **−cudart=shared** or equivalent flag(s) to use a dynamically−linked CUDA runtime library.

Static Link with **−cudart=static** or equivalent flag(s) to use a statically−linked CUDA runtime library.

Contents of **CMAKE_CUDA_RUNTIME_LIBRARY** may use *generator expressions*.

If this variable is not set then the *CUDA_RUNTIME_LIBRARY* target property will not be set automatically. If that property is not set then CMake uses an appropriate default value based on the compiler to select the CUDA runtime library.

NOTE:
This property has effect only when the **CUDA** language is enabled. To control the CUDA runtime linking when only using the CUDA SDK with the **C** or **C++** language we recommend using the *FindCUDAToolkit* module.

CMAKE_CUDA_SEPARABLE_COMPILATION
Added in version 3.11.

Default value for *CUDA_SEPARABLE_COMPILATION* target property. This variable is used to initialize the property on each target as it is created.

CMAKE_CXX_MODULE_STD
Added in version 3.30.

Whether to add utility targets as dependencies to targets with at least **cxx_std_23** or not.

NOTE:
This setting is meaningful only when experimental support for **import std;** has been enabled by the **CMAKE_EXPERIMENTAL_CXX_IMPORT_STD** gate.

This variable is used to initialize the *CXX_MODULE_STD* property on all targets. See that target property for additional information.

CMAKE_CXX_SCAN_FOR_MODULES
Added in version 3.28.

Whether to scan C++ source files for module dependencies.

This variable is used to initialize the *CXX_SCAN_FOR_MODULES* property on all the targets. See that target property for additional information.

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_DEBUGGER_WORKING_DIRECTORY
Added in version 4.0.

This variable is used to initialize the *DEBUGGER_WORKING_DIRECTORY* property on each target as it is created. See that target property for additional information.

CMAKE_DEFAULT_BUILD_TYPE
Added in version 3.17.

Specifies the configuration to use by default in a **build.ninja** file in the *Ninja Multi−Config* generator. If this variable is specified, **build.ninja** uses build rules from **build−<Config>.ninja** by default. All custom commands are executed with this configuration. If the variable is not specified, the first item from *CMAKE_CONFIGURATION_TYPES* is used instead.

The value of this variable must be one of the items from *CMAKE_CONFIGURATION_TYPES*.

CMAKE_DEFAULT_CONFIGS
Added in version 3.17.

Specifies a *semicolon−separated list* of configurations to build for a target in **build.ninja** if no **:<Config>** suffix is specified in the *Ninja Multi−Config* generator. If it is set to **all**, all configurations from *CMAKE_CROSS_CONFIGS* are used. If it is not specified, it defaults to *CMAKE_DEFAULT_BUILD_TYPE*.

For example, if you set *CMAKE_DEFAULT_BUILD_TYPE* to **Release**, but set **CMAKE_DEFAULT_CONFIGS** to **Debug** or **all**, all **<target>** aliases in **build.ninja** will resolve to **<target>:Debug** or **<target>:all**, but custom commands will still use the **Release** configuration.

The value of this variable must be a subset of *CMAKE_CROSS_CONFIGS* or be the same as *CMAKE_DEFAULT_BUILD_TYPE*. It must not be specified if *CMAKE_DEFAULT_BUILD_TYPE* or *CMAKE_CROSS_CONFIGS* is not used.

CMAKE_DEPENDS_USE_COMPILER
Added in version 3.20.

For the *Makefile Generators*, source dependencies are now, for a selection of compilers, generated by the compiler itself. By defining this variable with value **FALSE**, you can restore the legacy behavior (i.e. using CMake for dependencies discovery).

CMAKE_DISABLE_PRECOMPILE_HEADERS
Added in version 3.16.

Default value for *DISABLE_PRECOMPILE_HEADERS* of targets.

By default **CMAKE_DISABLE_PRECOMPILE_HEADERS** is **OFF**.

CMAKE_DLL_NAME_WITH_SOVERSION
Added in version 3.27.

This variable is used to initialize the *DLL_NAME_WITH_SOVERSION* property on shared library targets for the Windows platform, which is selected when the *WIN32* variable is set.

See this target property for additional information.

Please note that setting this variable has no effect if versioned filenames are globally disabled with the *CMAKE_PLATFORM_NO_VERSIONED_SONAME* variable.

CMAKE_ENABLE_EXPORTS
Added in version 3.4.

Specify whether executables export symbols for loadable modules.

This variable is used to initialize the *ENABLE_EXPORTS* target property for executable targets when they are created by calls to the *add_executable()* command. See the property documentation for details.

This variable has been superseded by the *CMAKE_EXECUTABLE_ENABLE_EXPORTS* variable. It is provided for backward compatibility with older CMake code, but should not be used in new projects.

CMAKE_EXECUTABLE_ENABLE_EXPORTS
Added in version 3.27.

Specify whether executables export symbols for loadable modules.

This variable is used to initialize the *ENABLE_EXPORTS* target property for executable targets when they are created by calls to the *add_executable()* command. See the property documentation for details.

This variable supersede the *CMAKE_ENABLE_EXPORTS* variable.

CMAKE_EXE_LINKER_FLAGS
Linker flags to be used to create executables.

These flags will be used by the linker when creating an executable.

Handling Compiler Driver Differences
Added in version 4.0.

To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* and *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variables.

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

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes **"LINKER:SHELL:-z defs"**.

NOTE:
Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

This support implies to parse and re-quote the content of the variable. See policy *CMP0181*.

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.

Handling Compiler Driver Differences
Added in version 4.0.

To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* and *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variables.

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

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes "**LINKER:SHELL:−z defs**".

NOTE:
Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

This support implies to parse and re−quote the content of the variable. See policy *CMP0181*.

CMAKE_EXE_LINKER_FLAGS_<CONFIG>_INIT
Added in version 3.7.

Value used to initialize the *CMAKE_EXE_LINKER_FLAGS_<CONFIG>* cache entry the first time a build tree is configured. This variable is meant to be set by a *toolchain file*. CMake may prepend or append content to the value based on the environment and target platform.

See also *CMAKE_EXE_LINKER_FLAGS_INIT*.

CMAKE_EXE_LINKER_FLAGS_INIT
Added in version 3.7.

Value used to initialize the *CMAKE_EXE_LINKER_FLAGS* cache entry the first time a build tree is configured. This variable is meant to be set by a *toolchain file*. CMake may prepend or append content to the value based on the environment and target platform.

See also the configuration−specific variable *CMAKE_EXE_LINKER_FLAGS_<CONFIG>_INIT*.

CMAKE_EXPORT_FIND_PACKAGE_NAME
NOTE:
Experimental. Gated by **CMAKE_EXPERIMENTAL_EXPORT_PACKAGE_DEPENDENCIES**.

Initializes the value of *EXPORT_FIND_PACKAGE_NAME*.

CMAKE_FOLDER
Added in version 3.12.

Set the folder name. Use to organize targets in an IDE.

This variable is used to initialize the *FOLDER* property on all the targets. See that target property for additional information.

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_Fortran_PREPROCESS
Added in version 3.18.

Default value for *Fortran_PREPROCESS* of targets.

This variable is used to initialize the *Fortran_PREPROCESS* property on all the targets. See that target property for additional information.

CMAKE_FRAMEWORK
Added in version 3.15.

Default value for *FRAMEWORK* of targets.

This variable is used to initialize the *FRAMEWORK* property on all the targets. See that target property for additional information.

CMAKE_FRAMEWORK_MULTI_CONFIG_POSTFIX_<CONFIG>
Added in version 3.18.

Default framework filename postfix under configuration **<CONFIG>** when using a multi−config generator.

When a framework target is created its *FRAMEWORK_MULTI_CONFIG_POSTFIX_<CONFIG>* target property is initialized with the value of this variable if it is set.

CMAKE_GHS_NO_SOURCE_GROUP_FILE
Added in version 3.14.

ON / **OFF** boolean to control if the project file for a target should be one single file or multiple files. Refer to *GHS_NO_SOURCE_GROUP_FILE* for further details.

CMAKE_GLOBAL_AUTOGEN_TARGET
Added in version 3.14.

Switch to enable generation of a global **autogen** target.

When **CMAKE_GLOBAL_AUTOGEN_TARGET** is enabled, a custom target **autogen** is generated. This target depends on all *AUTOMOC* and *AUTOUIC* generated *The <ORIGIN>_autogen target* targets in

the project. By building the global **autogen** target, all *AUTOMOC* and *AUTOUIC* files in the project will be generated.

The name of the global **autogen** target can be changed by setting *CMAKE_GLOBAL_AUTOGEN_TARGET_NAME*.

By default **CMAKE_GLOBAL_AUTOGEN_TARGET** is unset.

See the *cmake−qt(7)* manual for more information on using CMake with Qt.

NOTE:
The <ORIGIN>_autogen target targets by default inherit their origin target's dependencies. This might result in unintended dependency target builds when only *The <ORIGIN>_autogen target* targets are built. A solution is to disable*A UTOGEN_ORIGIN_DEPENDS* on the respective origin targets.

CMAKE_GLOBAL_AUTOGEN_TARGET_NAME
Added in version 3.14.

Change the name of the global **autogen** target.

When *CMAKE_GLOBAL_AUTOGEN_TARGET* is enabled, a global custom target named **autogen** is created. **CMAKE_GLOBAL_AUTOGEN_TARGET_NAME** allows to set a different name for that target.

By default **CMAKE_GLOBAL_AUTOGEN_TARGET_NAME** is unset.

See the *cmake−qt(7)* manual for more information on using CMake with Qt.

CMAKE_GLOBAL_AUTORCC_TARGET
Added in version 3.14.

Switch to enable generation of a global **autorcc** target.

When **CMAKE_GLOBAL_AUTORCC_TARGET** is enabled, a custom target **autorcc** is generated. This target depends on all *AUTORCC* generated **<ORIGIN>_arcc_<QRC>** targets in the project. By building the global **autorcc** target, all *AUTORCC* files in the project will be generated.

The name of the global **autorcc** target can be changed by setting *CMAKE_GLOBAL_AUTORCC_TARGET_NAME*.

By default **CMAKE_GLOBAL_AUTORCC_TARGET** is unset.

See the *cmake−qt(7)* manual for more information on using CMake with Qt.

CMAKE_GLOBAL_AUTORCC_TARGET_NAME
Added in version 3.14.

Change the name of the global **autorcc** target.

When *CMAKE_GLOBAL_AUTORCC_TARGET* is enabled, a global custom target named **autorcc** is created. **CMAKE_GLOBAL_AUTORCC_TARGET_NAME** allows to set a different name for that target.

By default **CMAKE_GLOBAL_AUTORCC_TARGET_NAME** is unset.

See the *cmake−qt(7)* manual for more information on using CMake with Qt.

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 *CMAKE_CURRENT_SOURCE_DIR* and *CMAKE_CURRENT_BINARY_DIR* to the include path for each 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* target property.

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* target property. By default **CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE** is **OFF**.

CMAKE_INSTALL_NAME_DIR
Directory name for installed targets on Apple platforms.

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_REMOVE_ENVIRONMENT_RPATH
Added in version 3.16.

Sets the default for whether toolchain−defined rpaths should be removed during installation.

CMAKE_INSTALL_REMOVE_ENVIRONMENT_RPATH is a boolean that provides the default value for the *INSTALL_REMOVE_ENVIRONMENT_RPATH* property of all subsequently created targets.

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 to the run-time search path (rpath) of installed binaries any directories outside the project that are in the linker search path or contain linked library files. The directories are appended after the value of the *INSTALL_RPATH* target property.

This variable is used to initialize the target property *INSTALL_RPATH_USE_LINK_PATH* for all targets.

CMAKE_INTERPROCEDURAL_OPTIMIZATION
Added in version 3.9.

Default value for *INTERPROCEDURAL_OPTIMIZATION* of targets.

This variable is used to initialize the *INTERPROCEDURAL_OPTIMIZATION* property on all the targets. See that target property for additional information.

CMAKE_INTERPROCEDURAL_OPTIMIZATION_<CONFIG>
Added in version 3.9.

Default value for *INTERPROCEDURAL_OPTIMIZATION_<CONFIG>* of targets.

This variable is used to initialize the *INTERPROCEDURAL_OPTIMIZATION_<CONFIG>* property on all the targets. See that target property for additional information.

CMAKE_<LANG>_CLANG_TIDY
Added in version 3.6.

Default value for *<LANG>_CLANG_TIDY* target property when **<LANG>** is **C**, **CXX**, **OBJC** or **OBJCXX**.

This variable is used to initialize the property on each target as it is created. For example:

```
set(CMAKE_CXX_CLANG_TIDY clang-tidy -checks=-*,readability-*)
add_executable(foo foo.cxx)
```

CMAKE_<LANG>_CLANG_TIDY_EXPORT_FIXES_DIR
Added in version 3.26.

Default value for *<LANG>_CLANG_TIDY_EXPORT_FIXES_DIR* target property when **<LANG>** is **C**, **CXX**, **OBJC** or **OBJCXX**.

This variable is used to initialize the property on each target as it is created. For example:

```
set(CMAKE_CXX_CLANG_TIDY_EXPORT_FIXES_DIR clang-tidy-fixes)
add_executable(foo foo.cxx)
```

CMAKE_<LANG>_COMPILER_LAUNCHER
Added in version 3.4.

Default value for *<LANG>_COMPILER_LAUNCHER* target property. This variable is used to initialize the property on each target as it is created. This is done only when **<LANG>** is **C**, **CXX**, **Fortran**, **HIP**, **ISPC**, **OBJC**, **OBJCXX**, or **CUDA**.

This variable is initialized to the *CMAKE_<LANG>_COMPILER_LAUNCHER* environment variable if it is set.

CMAKE_<LANG>_CPPCHECK
Added in version 3.10.

Default value for *<LANG>_CPPCHECK* target property. This variable is used to initialize the property on each target as it is created. This is done only when **<LANG>** is **C** or **CXX**.

CMAKE_<LANG>_CPPLINT
Added in version 3.8.

Default value for *<LANG>_CPPLINT* target property. This variable is used to initialize the property on each target as it is created. This is done only when **<LANG>** is **C** or **CXX**.

CMAKE_<LANG>_INCLUDE_WHAT_YOU_USE
Added in version 3.3.

Default value for *<LANG>_INCLUDE_WHAT_YOU_USE* target property. This variable is used to initialize the property on each target as it is created. This is done only when **<LANG>** is **C** or **CXX**.

CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>
Added in version 3.24.

This variable defines how to link a group of libraries for the specified**<FEATURE>** when a *LINK_GROUP* generator expression is used and the link language for the target is **<LANG>**. For this variable to have any effect, the associated *CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>_SUPPORTED* variable must be set to true.

The *CMAKE_LINK_GROUP_USING_<FEATURE>* variable should be defined instead for features that are independent of the link language.

Feature names are case−sensitive and may only contain letters, numbers and underscores. Feature names defined in all uppercase are reserved for CMake's own built−in features (see *Predefined Features* further below).

Feature Definitions
A group feature definition is a list that contains exactly two elements:

```
<PREFIX> <SUFFIX>
```

On the linker command line, **<PREFIX>** will precede the list of libraries in the group and **<SUFFIX>** will follow after.

For the elements of this variable, the **LINKER:** prefix can be used.

To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* and *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variables.

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

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

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes **"LINKER:SHELL:−z defs"**.

NOTE:
Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

Examples

Solving cross−references between two static libraries

A project may define two or more static libraries which have circular dependencies between them. In order for the linker to resolve all symbols at link time, it may need to search repeatedly among the libraries until no new undefined references are created. Different linkers use different syntax for achieving this. The following example shows how this may be implemented for some linkers. Note that this is for illustration purposes only. Projects should use the built−in **RESCAN** group feature instead (see *Predefined Features*), which provides a more complete and more robust implementation of this functionality.

```
set(CMAKE_C_LINK_GROUP_USING_cross_refs_SUPPORTED TRUE)
if(CMAKE_C_COMPILER_ID STREQUAL "GNU" AND CMAKE_SYSTEM_NAME STREQUAL "Linux")
  set(CMAKE_C_LINK_GROUP_USING_cross_refs
    "LINKER:--start-group"
    "LINKER:--end-group"
  )
elseif(CMAKE_C_COMPILER_ID STREQUAL "SunPro" AND CMAKE_SYSTEM_NAME STREQUAL "SunO
  set(CMAKE_C_LINK_GROUP_USING_cross_refs
    "LINKER:-z,rescan-start"
    "LINKER:-z,rescan-end"
  )
else()
  # feature not yet supported for the other environments
  set(CMAKE_C_LINK_GROUP_USING_cross_refs_SUPPORTED FALSE)
endif()

add_library(lib1 STATIC ...)
add_library(lib2 SHARED ...)

if(CMAKE_C_LINK_GROUP_USING_cross_refs_SUPPORTED)
  target_link_libraries(lib2 PRIVATE "$<LINK_GROUP:cross_refs,lib1,external>")
else()
  target_link_libraries(lib2 PRIVATE lib1 external)
endif()
```

CMake will generate the following linker command line fragments when linking **lib2**:

- **GNU**: **−Wl,−−start−group /path/to/lib1.a −lexternal −Wl,−−end−group**
- **SunPro**: **−Wl,−z,rescan−start /path/to/lib1.a −lexternal −Wl,−z,rescan−end**

Predefined Features

The following built−in group features are pre−defined by CMake:

RESCAN

Some linkers are single−pass only. For such linkers, circular references between libraries typically result in unresolved symbols. This feature instructs the linker to search the specified static libraries repeatedly until no new undefined references are created.

Normally, a static library is searched only once in the order that it is specified on the command line. If a symbol in that library is needed to resolve an undefined symbol referred to by an object in a library that appears later on the command line, the linker would not be able to resolve that reference. By grouping the static libraries with the**RESCAN** feature, the y will all be searched repeatedly until all possible references are resolved. This will use linker options like

−−start−group and **−−end−group**, or on SunOS, **−z rescan−start** and **−z rescan−end**.

Using this feature has a significant performance cost. It is best to use it only when there are unavoidable circular references between two or more static libraries.

This feature is available when using toolchains that target Linux, BSD, and SunOS. It can also be used when targeting Windows platforms if the GNU toolchain is used.

CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>_SUPPORTED
Added in version 3.24.

This variable specifies whether the **<FEATURE>** is supported for the link language **<LANG>**. If this variable is true, then the **<FEATURE>** must be defined by *CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>*, and the more generic *CMAKE_LINK_GROUP_USING_<FEATURE>_SUPPORTED* and *CMAKE_LINK_GROUP_USING_<FEATURE>* variables are not used.

If **CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>_SUPPORTED** is false or is not set, then the *CMAKE_LINK_GROUP_USING_<FEATURE>_SUPPORTED* variable will determine whether **<FEATURE>** is deemed to be supported.

CMAKE_<LANG>_LINK_LIBRARY_<FEATURE>_ATTRIBUTES
Added in version 3.30.

This variable defines the semantics of the specified link library **<FEATURE>** when linking with the link language **<LANG>**. It takes precedence over *CMAKE_LINK_LIBRARY_<FEATURE>_ATTRIBUTES* if that variable is also defined for the same **<FEATURE>**, but otherwise has similar effects. See *CMAKE_LINK_LIBRARY_<FEATURE>_ATTRIBUTES* for further details.

CMAKE_<LANG>_LINK_LIBRARY_FILE_FLAG
Added in version 3.16.

Language−specific 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_<LANG>_LINK_LIBRARY_FLAG
Added in version 3.16.

Flag to be used to link a library into a shared library or executable.

This flag will be used to specify a library to link to a shared library or an executable for the specific language. On most compilers this is**−l**.

CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>
Added in version 3.24.

This variable defines how to link a library or framework for the specified **<FEATURE>** when a *LINK_LIBRARY* generator expression is used and the link language for the target is **<LANG>**. For this variable to have any effect, the associated *CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED* variable must be set to true.

The *CMAKE_LINK_LIBRARY_USING_<FEATURE>* variable should be defined instead for features that are independent of the link language.

Feature names are case−sensitive and may only contain letters, numbers and underscores. Feature names defined in all uppercase are reserved for CMake's own built−in features (see *Predefined Features* further below).

Some aspects of feature behavior can be defined by the *CMAKE_<LANG>_LINK_LIBRARY_<FEATURE>_ATTRIBUTES* and *CMAKE_LINK_LIBRARY_<FEATURE>_ATTRIBUTES* variables.

Feature Definitions

A library feature definition is a list that contains one or three elements:

```
[<PREFIX>] <LIBRARY_EXPRESSION> [<SUFFIX>]
```

When **<PREFIX>** and **<SUFFIX>** are specified, they precede and follow respectively the whole list of libraries specified in the *LINK_LIBRARY* expression, not each library item individually. There is no guarantee that the list of specified libraries will be kept grouped together though, so the **<PREFIX>** and **<SUFFIX>** may appear more than once if the library list is reorganized by CMake to satisfy other constraints. This means constructs like **−−start−group** and **−−end−group**, as supported by the GNU **ld** linker, cannot be used in this way. The*LINK_GR OUP* generator expression should be used instead for such constructs.

<LIBRARY_EXPRESSION> is used to specify the pattern for constructing the corresponding fragment on the linker command line for each library. The following placeholders can be used in the expression:

- **<LIBRARY>** is expanded to the full path to the library for CMake targets, or to a platform−specific value based on the item otherwise (the same as **<LINK_ITEM>** on Windows, or the library base name for other platforms).
- **<LINK_ITEM>** is expanded to how the library would normally be linked on the linker command line.
- **<LIB_ITEM>** is expanded to the full path to the library for CMake targets, or the item itself exactly as specified in the **<LIBRARY_EXPRESSION>** otherwise.

In addition to the above, it is possible to have one pattern for paths (CMake targets and external libraries specified with file paths) and another for other items specified by name only. The**P ATH{}** and **NAME{}** wrappers can be used to provide the expansion for those two cases, respectively. When wrappers are used, both must be present. For example:

```
set(CMAKE_LINK_LIBRARY_USING_weak_library
    "PATH{−weak_library <LIBRARY>}NAME{LINKER:−weak−l<LIB_ITEM>}"
)
```

For all three elements of this variable (**<PREFIX>**, **<LIBRARY_EXPRESSION>**, and **<SUFFIX>**), the **LINKER:** prefix can be used.

To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* and *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variables.

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

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

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes "**LINKER:SHELL:−z defs**".

NOTE:
Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

Examples

Loading a whole static library

A common need is to prevent the linker from discarding any symbols from a static library. Different linkers use different syntax for achieving this. The following example shows how this may be implemented for some linkers. Note that this is for illustration purposes only. Projects should use the built−in **WHOLE_ARCHIVE** feature instead (see *Predefined Features*), which provides a more complete and more robust implementation of this functionality.

```
set(CMAKE_C_LINK_LIBRARY_USING_load_archive_SUPPORTED TRUE)
if(CMAKE_C_COMPILER_ID STREQUAL "AppleClang")
  set(CMAKE_C_LINK_LIBRARY_USING_load_archive "−force_load <LIB_ITEM>")
elseif(CMAKE_C_COMPILER_ID STREQUAL "GNU" AND CMAKE_SYSTEM_NAME STREQUAL "Linux")
  set(CMAKE_C_LINK_LIBRARY_USING_load_archive
    "LINKER:−−push−state,−−whole−archive"
    "<LINK_ITEM>"
    "LINKER:−−pop−state"
  )
elseif(CMAKE_C_COMPILER_ID STREQUAL "MSVC")
  set(CMAKE_C_LINK_LIBRARY_USING_load_archive "/WHOLEARCHIVE:<LIBRARY>")
else()
  # feature not yet supported for the other environments
  set(CMAKE_C_LINK_LIBRARY_USING_load_archive_SUPPORTED FALSE)
endif()

add_library(lib1 STATIC ...)
add_library(lib2 SHARED ...)

if(CMAKE_C_LINK_LIBRARY_USING_load_archive_SUPPORTED)
  # The −force_load Apple linker option requires a file name
  set(external_lib
    "$<IF:$<LINK_LANG_AND_ID:C,AppleClang>,libexternal.a,external>"
  )
  target_link_libraries(lib2 PRIVATE
    "$<LINK_LIBRARY:load_archive,lib1,${external_lib}>"
  )
else()
  target_link_libraries(lib2 PRIVATE lib1 external)
endif()
```

CMake will generate the following link expressions:

- **AppleClang**: **−force_load /path/to/lib1.a −force_load libexternal.a**
- **GNU**: **−Wl,−−push−state,−−whole−archive /path/to/lib1.a −lexternal −Wl,−−pop−state**
- **MSVC**: **/WHOLEARCHIVE:/path/to/lib1.lib /WHOLEARCHIVE:external.lib**

Linking a library as weak

On macOS, it is possible to link a library in weak mode (the library and all references are marked as weak imports). Different flags must be used for a library specified by file path compared to one specified by

name. This constraint can be solved using **PATH{}** and **NAME{}** wrappers. Again, the following example shows how this may be implemented for some linkers, but it is for illustration purposes only. Projects should use the built−in **WEAK_FRAMEWORK** or **WEAK_LIBRARY** features instead (see *Predefined Features*), which provide more complete and more robust implementations of this functionality.

```
if (CMAKE_C_COMPILER_ID STREQUAL "AppleClang")
  set(CMAKE_LINK_LIBRARY_USING_weak_library
      "PATH{-weak_library <LIBRARY>}NAME{LINKER:-weak-l<LIB_ITEM>}"
  )
  set(CMAKE_LINK_LIBRARY_USING_weak_library_SUPPORTED TRUE)
endif()

add_library(lib SHARED ...)
add_executable(main ...)
if(CMAKE_LINK_LIBRARY_USING_weak_library_SUPPORTED)
  target_link_libraries(main PRIVATE "$<LINK_LIBRARY:weak_library,lib,external>")
else()
  target_link_libraries(main PRIVATE lib external)
endif()
```

CMake will generate the following linker command line fragment when linking **main** using the **AppleClang** toolchain:

−weak_library /path/to/lib −Xlinker −weak−lexternal.

Predefined Features

The following built−in library features are pre−defined by CMake:

DEFAULT

This feature corresponds to standard linking, essentially equivalent to using no feature at all. It is typically only used with the *LINK_LIBRARY_OVERRIDE* and *LINK_LIBRARY_OVERRIDE_<LIBRARY>* target properties.

WHOLE_ARCHIVE

Force inclusion of all members of a static library when linked as a dependency of consuming *Executables*, *Shared Libraries*, and *Module Libraries*. This feature is only supported for the following platforms, with limitations as noted:

- Linux.
- All BSD variants.
- SunOS.
- All Apple variants. The library must be specified as a CMake target name, a library file name (such as **libfoo.a**), or a library file path (such as **/path/to/libfoo.a**). Due to a limitation of the Apple linker, it cannot be specified as a plain library name like **foo**, where **foo** is not a CMake target.
- Windows. When using a MSVC or MSVC−like toolchain, the MSVC version must be greater than 1900.
- Cygwin.
- MSYS.

NOTE:

Since *Static Libraries* are archives and not linked binaries, CMake records their link dependencies for transitive use when linking consuming binaries. Therefore **WHOLE_ARCHIVE** does not cause a static library's objects to be included in other static libraries. Use *Object*

Libraries for that.

FRAMEWORK
This option tells the linker to search for the specified framework using the **−framework** linker option. It can only be used on Apple platforms, and only with a linker that understands the option used (i.e. the linker provided with Xcode, or one compatible with it).

The framework can be specified as a CMake framework target, a bare framework name, or a file path. If a target is given, that target must have the *FRAMEWORK* target property set to true. For a file path, if it contains a directory part, that directory will be added as a framework search path.

```
add_library(lib SHARED ...)
target_link_libraries(lib PRIVATE "$<LINK_LIBRARY:FRAMEWORK,/path/to/my_fra

# The constructed linker command line will contain:
#   -F/path/to -framework my_framework
```

File paths must conform to one of the following patterns (***** is a wildcard, and optional parts are shown as **[...]**):

- **[/path/to/]FwName[.framework]**
- **[/path/to/]FwName.framework/FwName[suffix]**
- **[/path/to/]FwName.framework/Versions/*/FwName[suffix]**

Note that CMake recognizes and automatically handles framework targets, even without using the *$<LINK_LIBRARY:FRAMEWORK,...>* expression. The generator expression can still be used with a CMake target if the project wants to be explicit about it, but it is not required to do so. The linker command line may have some differences between using the generator expression or not, but the final result should be the same. On the other hand, if a file path is given, CMake will recognize some paths automatically, but not all cases. The project may want to use *$<LINK_LIBRARY:FRAMEWORK,...>* for file paths so that the expected behavior is clear.

Added in version 3.25: The *FRAMEWORK_MULTI_CONFIG_POSTFIX_<CONFIG>* target property as well as the **suffix** of the framework library name are now supported by the **FRAMEWORK** features.

NEEDED_FRAMEWORK
This is similar to the **FRAMEWORK** feature, except it forces the linker to link with the framework even if no symbols are used from it. It uses the **−needed_framework** option and has the same linker constraints as **FRAMEWORK**.

REEXPORT_FRAMEWORK
This is similar to the **FRAMEWORK** feature, except it tells the linker that the framework should be available to clients linking to the library being created. It uses the **−reexport_framework** option and has the same linker constraints as **FRAMEWORK**.

WEAK_FRAMEWORK
This is similar to the **FRAMEWORK** feature, except it forces the linker to mark the framework and all references to it as weak imports. It uses the **−weak_framework** option and has the same linker constraints as **FRAMEWORK**.

NEEDED_LIBRARY
This is similar to the **NEEDED_FRAMEWORK** feature, except it is for use with non−framework targets or libraries (Apple platforms only). It uses the **−needed_library** or **−needed−l** option as appropriate, and has the same linker constraints as **NEEDED_FRAMEWORK**.

REEXPORT_LIBRARY
This is similar to the **REEXPORT_FRAMEWORK** feature, except it is for use with non−framework targets or libraries (Apple platforms only). It uses the **−reexport_library** or **−reexport−l** option as appropriate, and has the same linker constraints as **REEXPORT_FRAMEWORK**.

WEAK_LIBRARY
This is similar to the **WEAK_FRAMEWORK** feature, except it is for use with non−framework targets or libraries (Apple platforms only). It uses the **−weak_library** or **−weak−l** option as appropriate, and has the same linker constraints as **WEAK_FRAMEWORK**.

CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED
Added in version 3.24.

Set to **TRUE** if the **<FEATURE>**, as defined by variable *CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>*, is supported for the linker language **<LANG>**.

NOTE:
This variable is evaluated before the more generic variable *CMAKE_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED*.

CMAKE_<LANG>_LINK_WHAT_YOU_USE_FLAG
Added in version 3.22.

Linker flag used by *LINK_WHAT_YOU_USE* to tell the linker to link all shared libraries specified on the command line even if none of their symbols is needed. This is an implementation detail used so that the command in *CMAKE_LINK_WHAT_YOU_USE_CHECK* can check the binary for unnecessarily−linked shared libraries.

NOTE:
Do not rely on this abstraction to intentionally link to shared libraries whose symbols are not needed.

CMAKE_<LANG>_LINKER_LAUNCHER
Added in version 3.21.

Default value for *<LANG>_LINKER_LAUNCHER* target property. This variable is used to initialize the property on each target as it is created. This is done only when **<LANG>** is **C**, **CXX**, **OBJC**, or **OBJCXX**.

This variable is initialized to the *CMAKE_<LANG>_LINKER_LAUNCHER* environment variable if it is set.

CMAKE_<LANG>_USING_LINKER_<TYPE>
Added in version 3.29.

This variable defines how to specify the **<TYPE>** linker for the link step, as controlled by the *CMAKE_LINKER_TYPE* variable or the *LINKER_TYPE* target property. Depending on the value of the *CMAKE_<LANG>_LINK_MODE* variable, **CMAKE_<LANG>_USING_LINKER_<TYPE>** can hold compiler flags for the link step, or the path to the linker tool.

Changed in version 4.0.

The type of information stored in this variable is now determined by the *CMAKE_<LANG>_LINK_MODE*

variable instead of the *CMAKE_<LANG>_USING_LINKER_MODE* variable.

NOTE:
The specified linker tool is generally expected to be accessible through the **PATH** environment variable.

For example, the **LLD** linker for **GNU** compilers is defined like so:

```
# CMAKE_C_LINK_MODE holds value "DRIVER"
set(CMAKE_C_USING_LINKER_LLD "-fuse-ld=lld")
```

On the **Windows** platform with **Clang** compilers simulating **MSVC** with **GNU** front-end:

```
# CMAKE_C_LINK_MODE holds value "DRIVER"
set(CMAKE_C_USING_LINKER_LLD "-fuse-ld=lld-link")
```

And for the **MSVC** compiler or **Clang** compilers simulating **MSVC** with **MSVC** front-end, the linker is invoked directly, not via the compiler front-end:

```
# CMAKE_C_LINK_MODE holds value "LINKER"
set(CMAKE_C_USING_LINKER_LLD "/path/to/lld-link.exe")
```

A custom linker type can also be defined, usually in a toolchain file:

```
set(CMAKE_LINKER_TYPE lld_launcher)
set(CMAKE_C_USING_LINKER_lld_launcher "-fuse-ld=/path/to/lld-launcher.sh")
```

CMAKE_<LANG>_VISIBILITY_PRESET
Default value for the *<LANG>_VISIBILITY_PRESET* target property when a target is created.

CMAKE_LIBRARY_OUTPUT_DIRECTORY
Where to put all the *LIBRARY* target files 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_OUTPUT_DIRECTORY_<CONFIG>
Added in version 3.3.

Where to put all the *LIBRARY* target files when built for a specific configuration.

This variable is used to initialize the *LIBRARY_OUTPUT_DIRECTORY_<CONFIG>* 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_DEPENDS_USE_LINKER
Added in version 3.27.

For the *Makefile* and *Ninja* generators, link dependencies are now, for a selection of linkers, generated by the linker itself. By defining this variable with value **FALSE**, you can deactivate this feature.

This feature is also deactivated if the *LINK_DEPENDS_NO_SHARED* target property is true.

NOTE:
CMake version 4.0.0−rc1 defaults this variable to **FALSE** if the linker is one from the GNU binutils linkers (**ld** and **ld.bfd** for version less than 2.41 or **ld.gold** for any version) because it generate spurious dependencies on temporary files when LTO is enabled. See *GNU bug 30568*.

CMAKE_LINK_GROUP_USING_<FEATURE>
Added in version 3.24.

This variable defines how to link a group of libraries for the specified **<FEATURE>** when a *LINK_GROUP* generator expression is used. Both of the following conditions must be met for this variable to have any effect:

- The associated *CMAKE_LINK_GROUP_USING_<FEATURE>_SUPPORTED* variable must be set to true.
- There is no language−specific definition for the same **<FEATURE>**. This means *CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>_SUPPORTED* cannot be true for the link language used by the target for which the *LINK_GROUP* generator expression is evaluated.

The *CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>* variable should be defined instead for features that are dependent on the link language.

Feature names are case−sensitive and may only contain letters, numbers and underscores. Feature names defined in all uppercase are reserved for CMake's own built−in features (see *Predefined Features* further below).

Feature Definitions
A group feature definition is a list that contains exactly two elements:

```
<PREFIX> <SUFFIX>
```

On the linker command line, **<PREFIX>** will precede the list of libraries in the group and **<SUFFIX>** will follow after.

For the elements of this variable, the **LINKER:** prefix can be used.

To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* and *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variables.

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

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

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes "**LINKER:SHELL:−z defs**".

NOTE:

Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

Examples

Solving cross−references between two static libraries

A project may define two or more static libraries which have circular dependencies between them. In order for the linker to resolve all symbols at link time, it may need to search repeatedly among the libraries until no new undefined references are created. Different linkers use different syntax for achieving this. The following example shows how this may be implemented for some linkers. Note that this is for illustration purposes only. Projects should use the built−in **RESCAN** group feature instead (see *Predefined Features*), which provides a more complete and more robust implementation of this functionality.

```
set(CMAKE_C_LINK_GROUP_USING_cross_refs_SUPPORTED TRUE)
if(CMAKE_C_COMPILER_ID STREQUAL "GNU" AND CMAKE_SYSTEM_NAME STREQUAL "Linux")
  set(CMAKE_C_LINK_GROUP_USING_cross_refs
    "LINKER:--start-group"
    "LINKER:--end-group"
  )
elseif(CMAKE_C_COMPILER_ID STREQUAL "SunPro" AND CMAKE_SYSTEM_NAME STREQUAL "SunO
  set(CMAKE_C_LINK_GROUP_USING_cross_refs
    "LINKER:-z,rescan-start"
    "LINKER:-z,rescan-end"
  )
else()
  # feature not yet supported for the other environments
  set(CMAKE_C_LINK_GROUP_USING_cross_refs_SUPPORTED FALSE)
endif()

add_library(lib1 STATIC ...)
add_library(lib2 SHARED ...)

if(CMAKE_C_LINK_GROUP_USING_cross_refs_SUPPORTED)
  target_link_libraries(lib2 PRIVATE "$<LINK_GROUP:cross_refs,lib1,external>")
else()
  target_link_libraries(lib2 PRIVATE lib1 external)
endif()
```

CMake will generate the following linker command line fragments when linking **lib2**:

- **GNU**: **−Wl,−−start−group /path/to/lib1.a −lexternal −Wl,−−end−group**
- **SunPro**: **−Wl,−z,rescan−start /path/to/lib1.a −lexternal −Wl,−z,rescan−end**

Predefined Features

The following built−in group features are pre−defined by CMake:

RESCAN

Some linkers are single−pass only. For such linkers, circular references between libraries typically result in unresolved symbols. This feature instructs the linker to search the specified static libraries repeatedly until no new undefined references are created.

Normally, a static library is searched only once in the order that it is specified on the command line. If a symbol in that library is needed to resolve an undefined symbol referred to by an object in a library that appears later on the command line, the linker would not be able to resolve that reference. By grouping the static libraries with the**RESCAN** feature, the y will all be searched repeatedly until all possible references are resolved. This will use linker options like **−−start−group** and **−−end−group**, or on SunOS, **−z rescan−start** and **−z rescan−end**.

Using this feature has a significant performance cost. It is best to use it only when there are unavoidable circular references between two or more static libraries.

This feature is available when using toolchains that target Linux, BSD, and SunOS. It can also be used when targeting Windows platforms if the GNU toolchain is used.

CMAKE_LINK_GROUP_USING_<FEATURE>_SUPPORTED
Added in version 3.24.

This variable specifies whether the **<FEATURE>** is supported regardless of the link language. If this variable is true, then the **<FEATURE>** must be defined by *CMAKE_LINK_GROUP_USING_<FEATURE>*.

Note that this variable has no effect if *CMAKE_<LANG>_LINK_GROUP_USING_<FEATURE>_SUPPORTED* is true for the link language of the target.

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_LIBRARIES_STRATEGY
Added in version 3.31.

Specify a strategy for ordering targets' direct link dependencies on linker command lines.

If set, this variable acts as the default value for the *LINK_LIBRARIES_STRATEGY* target property when a target is created. Set that property directly to specify a strategy for a single target.

CMAKE_LINK_LIBRARY_<FEATURE>_ATTRIBUTES
Added in version 3.30.

This variable defines the behavior of the specified link library **<FEATURE>**. It specifies how the **<FEATURE>** interacts with other features, when the **<FEATURE>** should be applied, and aspects of how the **<FEATURE>** should be handled when CMake assembles the final linker command line (e.g. de−duplication).

The syntax of the linker flags for the **<FEATURE>** are controlled by the *CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>* and *CMAKE_LINK_LIBRARY_USING_<FEATURE>* variables. The *CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED* and *CMAKE_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED* variables control whether the **<FEATURE>** is available at all.

When linking for a particular language **<LANG>**, **CMAKE_LINK_LIBRARY_<FEATURE>_ATTRIBUTES** is ignored if the *CMAKE_<LANG>_LINK_LIBRARY_<FEATURE>_ATTRIBUTES* variable is also

defined for the same **<FEATURE>**.

The value of **CMAKE_LINK_LIBRARY_<FEATURE>_ATTRIBUTES** and *CMAKE_<LANG>_LINK_LIBRARY_<FEATURE>_ATTRIBUTES* at the end of the directory scope in which a target is defined is what matters.

Feature Attributes Definition

A feature attributes definition is a *semicolon−separated list* of **attribute=value(s)** items. If an attribute has multiple values, those values must be comma−separated.

The following attributes are supported:

LIBRARY_TYPE=<library−type−list>

Specify the library types supported by the feature. Supported values are: **STATIC**, **SHARED**, **MODULE**, and **EXECUTABLE**.

If this attribute is not specified, the default is **LIBRARY_TYPE=STATIC,SHARED,MODULE,EXECUTABLE**.

If the feature is used with an unsupported library type, CMake will emit a developer warning and the feature will be ignored.

OVERRIDE=<feature−list>

Specify which features this one replaces in the event of a conflict. This override mechanism is superseded by *LINK_LIBRARY_OVERRIDE* or *LINK_LIBRARY_OVERRIDE_<LIBRARY>* target property definitions, if defined.

If this attribute is not specified, the default is an empty list.

DEDUPLICATION=YES|NO|DEFAULT

Specify the de−duplication strategy for a library using this feature.

YES The library is always de−duplicated. The default strategy CMake would normally apply is ignored.

NO The library is never de−duplicated. The default strategy CMake would normally apply is ignored.

DEFAULT

Let CMake determine a de−duplication strategy automatically.

If this attribute is not specified, **DEFAULT** will be used.

Example

A common need is the loading of a full archive as part of the creation of a shared library. The built−in **WHOLE_ARCHIVE** feature can be used for that purpose. The implementation of that built−in feature sets the following link library feature attributes:

```
set(CMAKE_LINK_LIBRARY_WHOLE_ARCHIVE_ATTRIBUTES
  LIBRARY_TYPE=STATIC
  OVERRIDE=DEFAULT
  DEDUPLICATION=YES
)
```

LIBRARY_TYPE=STATIC

This feature is only meaningful for static libraries.

OVERRIDE=DEFAULT

The **DEFAULT** feature will be overridden by the **WHOLE_ARCHIVE** feature because they are compatible and enhance the user's experience: standard library specification and

$<LINK_LIBRARY:WHOLE_ARCHIVE> can be used freely.

DEDUPLICATION=YES
When this feature is used, the linker loads all symbols from the static library, so there is no need to repeat the library on the linker command line.

The **WHOLE_ARCHIVE** feature can be used like so:

```
add_library(A STATIC ...)
add_library(B STATIC ...)

target_link_libraries(B PUBLIC A)
target_link_libraries(A PUBLIC B)

add_library(global SHARED ...)
target_link_libraries(global PRIVATE $<LINK_LIBRARY:WHOLE_ARCHIVE,A>)
```

The resulting link command will only have one instance of the **A** library specified, and the linker flags will ensure that all symbols are loaded from the **A** library.

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_LINK_LIBRARY_USING_<FEATURE>
Added in version 3.24.

This variable defines how to link a library or framework for the specified **<FEATURE>** when a *LINK_LIBRARY* generator expression is used. Both of the following conditions must be met for this variable to have any effect:

- The associated *CMAKE_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED* variable must be set to true.
- There is no language−specific definition for the same **<FEATURE>**. This means *CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED* cannot be true for the link language used by the target for which the *LINK_LIBRARY* generator expression is evaluated.

Feature names are case−sensitive and may only contain letters, numbers and underscores. Feature names defined in all uppercase are reserved for CMake's own built−in features (see *Predefined Features* further below).

Some aspects of feature behavior can be defined by the *CMAKE_<LANG>_LINK_LIBRARY_<FEATURE>_ATTRIBUTES* and *CMAKE_LINK_LIBRARY_<FEATURE>_ATTRIBUTES* variables.

Feature Definitions
A library feature definition is a list that contains one or three elements:

```
[<PREFIX>] <LIBRARY_EXPRESSION> [<SUFFIX>]
```

When **<PREFIX>** and **<SUFFIX>** are specified, they precede and follow respectively the whole list of libraries specified in the *LINK_LIBRARY* expression, not each library item individually. There is no guarantee that the list of specified libraries will be kept grouped together though, so the **<PREFIX>** and **<SUFFIX>** may appear more than once if the library list is reorganized by CMake to satisfy other constraints. This means constructs like **−−start−group** and **−−end−group**, as supported by the GNU **ld** linker, cannot be used in this way. The*LINK_GR OUP* generator expression should be used instead for such constructs.

<LIBRARY_EXPRESSION> is used to specify the pattern for constructing the corresponding fragment on the linker command line for each library. The following placeholders can be used in the expression:

- **<LIBRARY>** is expanded to the full path to the library for CMake targets, or to a platform−specific value based on the item otherwise (the same as **<LINK_ITEM>** on Windows, or the library base name for other platforms).
- **<LINK_ITEM>** is expanded to how the library would normally be linked on the linker command line.
- **<LIB_ITEM>** is expanded to the full path to the library for CMake targets, or the item itself exactly as specified in the **<LIBRARY_EXPRESSION>** otherwise.

In addition to the above, it is possible to have one pattern for paths (CMake targets and external libraries specified with file paths) and another for other items specified by name only. The**P ATH{}** and **NAME{}** wrappers can be used to provide the expansion for those two cases, respectively. When wrappers are used, both must be present. For example:

```
set(CMAKE_LINK_LIBRARY_USING_weak_library
    "PATH{−weak_library <LIBRARY>}NAME{LINKER:−weak−l<LIB_ITEM>}"
)
```

For all three elements of this variable (**<PREFIX>**, **<LIBRARY_EXPRESSION>**, and **<SUFFIX>**), the **LINKER:** prefix can be used.

To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* and *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variables.

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

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

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes **"LINKER:SHELL:−z defs"**.

NOTE:

Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

Examples

Loading a whole static library

A common need is to prevent the linker from discarding any symbols from a static library. Different linkers use different syntax for achieving this. The following example shows how this may be implemented for some linkers. Note that this is for illustration purposes only. Projects should use the built−in **WHOLE_ARCHIVE** feature instead (see *Predefined Features*), which provides a more complete and more robust implementation of this functionality.

```
set(CMAKE_C_LINK_LIBRARY_USING_load_archive_SUPPORTED TRUE)
if(CMAKE_C_COMPILER_ID STREQUAL "AppleClang")
  set(CMAKE_C_LINK_LIBRARY_USING_load_archive "-force_load <LIB_ITEM>")
elseif(CMAKE_C_COMPILER_ID STREQUAL "GNU" AND CMAKE_SYSTEM_NAME STREQUAL "Linux")
  set(CMAKE_C_LINK_LIBRARY_USING_load_archive
    "LINKER:--push-state,--whole-archive"
    "<LINK_ITEM>"
    "LINKER:--pop-state"
  )
elseif(CMAKE_C_COMPILER_ID STREQUAL "MSVC")
  set(CMAKE_C_LINK_LIBRARY_USING_load_archive "/WHOLEARCHIVE:<LIBRARY>")
else()
  # feature not yet supported for the other environments
  set(CMAKE_C_LINK_LIBRARY_USING_load_archive_SUPPORTED FALSE)
endif()

add_library(lib1 STATIC ...)
add_library(lib2 SHARED ...)

if(CMAKE_C_LINK_LIBRARY_USING_load_archive_SUPPORTED)
  # The -force_load Apple linker option requires a file name
  set(external_lib
    "$<IF:$<LINK_LANG_AND_ID:C,AppleClang>,libexternal.a,external>"
  )
  target_link_libraries(lib2 PRIVATE
    "$<LINK_LIBRARY:load_archive,lib1,${external_lib}>"
  )
else()
  target_link_libraries(lib2 PRIVATE lib1 external)
endif()
```

CMake will generate the following link expressions:

- **AppleClang**: **-force_load /path/to/lib1.a -force_load libexternal.a**
- **GNU**: **-Wl,--push-state,--whole-archive /path/to/lib1.a -lexternal -Wl,--pop-state**
- **MSVC**: **/WHOLEARCHIVE:/path/to/lib1.lib /WHOLEARCHIVE:external.lib**

Linking a library as weak

On macOS, it is possible to link a library in weak mode (the library and all references are marked as weak imports). Different flags must be used for a library specified by file path compared to one specified by name. This constraint can be solved using **PATH{}** and **NAME{}** wrappers. Again, the following example shows how this may be implemented for some linkers, but it is for illustration purposes only. Projects should use the built-in **WEAK_FRAMEWORK** or **WEAK_LIBRARY** features instead (see *Predefined Features*), which provide more complete and more robust implementations of this functionality.

```
if (CMAKE_C_COMPILER_ID STREQUAL "AppleClang")
  set(CMAKE_LINK_LIBRARY_USING_weak_library
      "PATH{-weak_library <LIBRARY>}NAME{LINKER:-weak-l<LIB_ITEM>}"
  )
  set(CMAKE_LINK_LIBRARY_USING_weak_library_SUPPORTED TRUE)
endif()

add_library(lib SHARED ...)
add_executable(main ...)
if(CMAKE_LINK_LIBRARY_USING_weak_library_SUPPORTED)
```

```
  target_link_libraries(main PRIVATE "$<LINK_LIBRARY:weak_library,lib,external>")
else()
  target_link_libraries(main PRIVATE lib external)
endif()
```

CMake will generate the following linker command line fragment when linking **main** using the **AppleClang** toolchain:

−weak_library /path/to/lib −Xlinker −weak−lexternal.

Predefined Features

The following built−in library features are pre−defined by CMake:

DEFAULT

This feature corresponds to standard linking, essentially equivalent to using no feature at all. It is typically only used with the *LINK_LIBRARY_OVERRIDE* and *LINK_LIBRARY_OVERRIDE_<LIBRARY>* target properties.

WHOLE_ARCHIVE

Force inclusion of all members of a static library when linked as a dependency of consuming *Executables*, *Shared Libraries*, and *Module Libraries*. This feature is only supported for the following platforms, with limitations as noted:

- Linux.
- All BSD variants.
- SunOS.
- All Apple variants. The library must be specified as a CMake target name, a library file name (such as **libfoo.a**), or a library file path (such as **/path/to/libfoo.a**). Due to a limitation of the Apple linker, it cannot be specified as a plain library name like **foo**, where **foo** is not a CMake target.
- Windows. When using a MSVC or MSVC−like toolchain, the MSVC version must be greater than 1900.
- Cygwin.
- MSYS.

NOTE:

Since *Static Libraries* are archives and not linked binaries, CMake records their link dependencies for transitive use when linking consuming binaries. Therefore **WHOLE_ARCHIVE** does not cause a static library's objects to be included in other static libraries. Use *Object Libraries* for that.

FRAMEWORK

This option tells the linker to search for the specified framework using the **−framework** linker option. It can only be used on Apple platforms, and only with a linker that understands the option used (i.e. the linker provided with Xcode, or one compatible with it).

The framework can be specified as a CMake framework target, a bare framework name, or a file path. If a target is given, that target must have the *FRAMEWORK* target property set to true. For a file path, if it contains a directory part, that directory will be added as a framework search path.

```
add_library(lib SHARED ...)
target_link_libraries(lib PRIVATE "$<LINK_LIBRARY:FRAMEWORK,/path/to/my_fra

# The constructed linker command line will contain:
#   −F/path/to −framework my_framework
```

File paths must conform to one of the following patterns (***** is a wildcard, and optional parts are shown as **[...]**):

- **[/path/to/]FwName[.framework]**
- **[/path/to/]FwName.framework/FwName[suffix]**
- **[/path/to/]FwName.framework/Versions/*/FwName[suffix]**

Note that CMake recognizes and automatically handles framework targets, even without using the *$<LINK_LIBRARY:FRAMEWORK,...>* expression. The generator expression can still be used with a CMake target if the project wants to be explicit about it, but it is not required to do so. The linker command line may have some differences between using the generator expression or not, but the final result should be the same. On the other hand, if a file path is given, CMake will recognize some paths automatically, but not all cases. The project may want to use *$<LINK_LIBRARY:FRAMEWORK,...>* for file paths so that the expected behavior is clear.

Added in version 3.25: The *FRAMEWORK_MULTI_CONFIG_POSTFIX_<CONFIG>* target property as well as the **suffix** of the framework library name are now supported by the **FRAMEWORK** features.

NEEDED_FRAMEWORK
This is similar to the **FRAMEWORK** feature, except it forces the linker to link with the framework even if no symbols are used from it. It uses the **−needed_framework** option and has the same linker constraints as **FRAMEWORK**.

REEXPORT_FRAMEWORK
This is similar to the **FRAMEWORK** feature, except it tells the linker that the framework should be available to clients linking to the library being created. It uses the **−reexport_framework** option and has the same linker constraints as **FRAMEWORK**.

WEAK_FRAMEWORK
This is similar to the **FRAMEWORK** feature, except it forces the linker to mark the framework and all references to it as weak imports. It uses the **−weak_framework** option and has the same linker constraints as **FRAMEWORK**.

NEEDED_LIBRARY
This is similar to the **NEEDED_FRAMEWORK** feature, except it is for use with non−framework targets or libraries (Apple platforms only). It uses the **−needed_library** or **−needed−l** option as appropriate, and has the same linker constraints as **NEEDED_FRAMEWORK**.

REEXPORT_LIBRARY
This is similar to the **REEXPORT_FRAMEWORK** feature, except it is for use with non−framework targets or libraries (Apple platforms only). It uses the **−reexport_library** or **−reexport−l** option as appropriate, and has the same linker constraints as **REEXPORT_FRAMEWORK**.

WEAK_LIBRARY
This is similar to the **WEAK_FRAMEWORK** feature, except it is for use with non−framework targets or libraries (Apple platforms only). It uses the **−weak_library** or **−weak−l** option as appropriate, and has the same linker constraints as **WEAK_FRAMEWORK**.

CMAKE_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED
Added in version 3.24.

Set to **TRUE** if the **<FEATURE>**, as defined by variable *CMAKE_LINK_LIBRARY_USING_<FEATURE>*, is supported regardless the linker language.

NOTE:

This variable is evaluated if, and only if, the variable *CMAKE_<LANG>_LINK_LIBRARY_USING_<FEATURE>_SUPPORTED* is not defined.

CMAKE_LINK_WARNING_AS_ERROR
Added in version 4.0.

Specify whether to treat warnings on link as errors.

This variable is used to initialize the *LINK_WARNING_AS_ERROR* property on all the targets.

CMAKE_LINK_WHAT_YOU_USE
Added in version 3.7.

Default value for *LINK_WHAT_YOU_USE* target property. This variable is used to initialize the property on each target as it is created.

CMAKE_LINK_WHAT_YOU_USE_CHECK
Added in version 3.22.

Command executed by *LINK_WHAT_YOU_USE* after the linker to check for unnecessarily−linked shared libraries. This check is currently only defined on**ELF** platforms with v alue **ldd −u −r**.

See also *CMAKE_<LANG>_LINK_WHAT_YOU_USE_FLAG* variables.

CMAKE_LINKER_TYPE
Added in version 3.29.

Specify which linker will be used for the link step.

This variable is used to initialize the *LINKER_TYPE* property on each target created by a call to *add_library()* or *add_executable()*. It is meaningful only for targets having a link step. If set, its value is also used by the *try_compile()* command.

NOTE:
It is assumed that the linker specified is fully compatible with the default one the compiler would normally invoke. CMake will not do any option translation.

Linker types are case−sensitive and may only contain letters, numbers and underscores. Linker types defined in all uppercase are reserved for CMake's own built−in types. The pre−defined linker types are:

DEFAULT
This type corresponds to standard linking, essentially equivalent to the *LINKER_TYPE* target property not being set at all.

SYSTEM
Use the standard linker provided by the platform or toolchain. For example, this implies the Microsoft linker for all MSVC−compatible compilers. This type is supported for the following platform−compiler combinations:

- Linux: **GNU**, **Clang**, **LLVMFlang**, **NVIDIA**, and **Swift** compilers.
- Apple platforms: **AppleClang**, **Clang**, **GNU**, and **Swift** compilers.
- Windows: **MSVC**, **GNU**, **Clang**, **NVIDIA**, and **Swift** compilers.

LLD Use the **LLVM** linker. This type is supported for the following platform−compiler combinations:

- Linux: **GNU**, **Clang**, **LLVMFlang**, **NVIDIA**, and **Swift** compilers.
- Apple platforms: **Clang**, **AppleClang**, and **Swift** compilers.
- Windows: **GNU**, **Clang** with MSVC−like front−end, **Clang** with GNU−like front−end, **MSVC**, **NVIDIA** with MSVC−like front−end, and **Swift**.

BFD Use the **GNU** linker. This type is supported for the following platform−compiler combinations:

- Linux: **GNU**, **Clang**, **LLVMFlang**, and **NVIDIA** compilers.
- Windows: **GNU**, **Clang** with GNU−like front−end.

GOLD Supported on Linux platform with **GNU**, **Clang**, **LLVMFlang**, **NVIDIA**, and **Swift** compilers.

MOLD
Use the *mold linker*. This type is supported on the following platform−compiler combinations:

- Linux: **GNU**, **Clang**, **LLVMFlang**, and **NVIDIA** compilers.
- Apple platforms: **Clang** and **AppleClang** compilers (acts as an alias to the *sold linker*).

SOLD Use the *sold linker*. This type is only supported on Apple platforms with **Clang** and **AppleClang** compilers.

APPLE_CLASSIC
Use the Apple linker in the classic behavior (i.e. before **Xcode 15.0**). This type is only supported on Apple platforms with **GNU**, **Clang**, **AppleClang**, and **Swift** compilers.

MSVC Use the Microsoft linker. This type is only supported on the Windows platform with **MSVC**, **Clang** with MSVC−like front−end, and **Swift** compilers.

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.

This variable is set to **ON** by default if *CMAKE_SYSTEM_NAME* equals to *iOS*, *tvOS*, *visionOS or watchOS*.

CMAKE_MACOSX_RPATH
Whether to use rpaths on macOS and iOS.

This variable is used to initialize the *MACOSX_RPATH* property on all targets.

CMAKE_MAP_IMPORTED_CONFIG_<CONFIG>
Default value for *MAP_IMPORTED_CONFIG_<CONFIG>* of targets.

This variable is used to initialize the *MAP_IMPORTED_CONFIG_<CONFIG>* 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.

Handling Compiler Driver Differences
Added in version 4.0.

To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by

the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* and *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variables.

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

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes **"LINKER:SHELL:−z defs"**.

NOTE:
Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

This support implies to parse and re−quote the content of the variable. See policy *CMP0181*.

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.

Handling Compiler Driver Differences
Added in version 4.0.

To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* and *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variables.

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

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes **"LINKER:SHELL:−z defs"**.

NOTE:
Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

This support implies to parse and re−quote the content of the variable. See policy *CMP0181*.

CMAKE_MODULE_LINKER_FLAGS_<CONFIG>_INIT
Added in version 3.7.

Value used to initialize the *CMAKE_MODULE_LINKER_FLAGS_<CONFIG>* cache entry the first time a build tree is configured. This variable is meant to be set by a *toolchain file*. CMake may prepend or append content to the value based on the environment and target platform.

See also *CMAKE_MODULE_LINKER_FLAGS_INIT*.

CMAKE_MODULE_LINKER_FLAGS_INIT
Added in version 3.7.

Value used to initialize the *CMAKE_MODULE_LINKER_FLAGS* cache entry the first time a build tree is configured. This variable is meant to be set by a *toolchain file*. CMake may prepend or append content to the value based on the environment and target platform.

See also the configuration−specific variable *CMAKE_MODULE_LINKER_FLAGS_<CONFIG>_INIT*.

CMAKE_MSVC_DEBUG_INFORMATION_FORMAT
Added in version 3.25.

Select the MSVC debug information format targeting the MSVC ABI. This variable is used to initialize the *MSVC_DEBUG_INFORMATION_FORMAT* property on all targets as they are created. It is also propagated by calls to the *try_compile()* command into the test project.

The allowed values are:

Embedded
Compile with **−Z7** or equivalent flag(s) to produce object files with full symbolic debugging information.

ProgramDatabase
Compile with **−Zi** or equivalent flag(s) to produce a program database that contains all the symbolic debugging information.

EditAndContinue
Compile with **−ZI** or equivalent flag(s) to produce a program database that supports the Edit and Continue feature.

The value is ignored on compilers not targeting the MSVC ABI, but an unsupported value will be rejected as an error when using a compiler targeting the MSVC ABI.

The value may also be the empty string (""), in which case no debug information format flag will be added explicitly by CMake.

Use *generator expressions* to support per−configuration specification. For example, the code:

```
set(CMAKE_MSVC_DEBUG_INFORMATION_FORMAT "$<$<CONFIG:Debug,RelWithDebInfo>:Program
```

selects for all following targets the program database debug information format for the **Debug** and **RelWithDebInfo** configurations.

If this variable is not set, the *MSVC_DEBUG_INFORMATION_FORMAT* target property will not be set automatically. If that property is not set, CMake selects a debug information format using the default value **$<$<CONFIG:Debug,RelWithDebInfo>:ProgramDatabase>**, if supported by the compiler, and otherwise **$<$<CONFIG:Debug,RelWithDebInfo>:Embedded>**.

NOTE:
This variable has effect only when policy *CMP0141* is set to **NEW** prior to the first *project()* or *enable_language()* command that enables a language using a compiler targeting the MSVC ABI.

CMAKE_MSVC_RUNTIME_CHECKS
Added in version 4.0.

Select the list of enabled runtime checks when targeting the MSVC ABI. This variable is used to initialize the *MSVC_RUNTIME_CHECKS* property on all targets as they are created. It is also propagated by calls to the *try_compile()* command into the test project.

The allowed values are:

PossibleDataLoss
Compile with **−RTCc** or equivalent flag(s) to enable possible data loss checks.

StackFrameErrorCheck
Compile with **−RTCs** or equivalent flag(s) to enable stack frame error checks.

UninitializedVariable
Compile with **−RTCu** or equivalent flag(s) to enable uninitialized variables checks.

The value is ignored on compilers not targeting the MSVC ABI, but an unsupported value will be rejected as an error when using a compiler targeting the MSVC ABI.

The value may also be the empty string (""), in which case no runtime error check flags will be added explicitly by CMake.

Use *generator expressions* to support per−configuration specification. For example, the code:

```
set(CMAKE_MSVC_RUNTIME_CHECKS "$<$<CONFIG:Debug,RelWithDebInfo>:PossibleDataLoss;
```

enables for the target **foo** the possible data loss and uninitialized variables checks for the **Debug** and **RelWithDebInfo** configurations.

If this variable is not set, the *MSVC_RUNTIME_CHECKS* target property will not be set automatically. If that property is not set, CMake selects runtime checks using the default value **$<$<CONFIG:Debug>:StackFrameErrorCheck;UninitializedVariable>**, if supported by the compiler, or empty value otherwise.

NOTE:
This variable has effect only when policy *CMP0184* is set to **NEW** prior to the first *project()* or *enable_language()* command that enables a language using a compiler targeting the MSVC ABI.

CMAKE_MSVC_RUNTIME_LIBRARY
Added in version 3.15.

Select the MSVC runtime library for use by compilers targeting the MSVC ABI. This variable is used to initialize the *MSVC_RUNTIME_LIBRARY* property on all targets as they are created. It is also propagated by calls to the *try_compile()* command into the test project.

The allowed values are:

MultiThreaded
Compile with **−MT** or equivalent flag(s) to use a multi−threaded statically−linked runtime library.

MultiThreadedDLL
Compile with **−MD** or equivalent flag(s) to use a multi−threaded dynamically−linked runtime library.

MultiThreadedDebug
Compile with **−MTd** or equivalent flag(s) to use a multi−threaded statically−linked runtime library.

MultiThreadedDebugDLL
Compile with **−MDd** or equivalent flag(s) to use a multi−threaded dynamically−linked runtime library.

The value is ignored on compilers not targeting the MSVC ABI, but an unsupported value will be rejected

as an error when using a compiler targeting the MSVC ABI.

The value may also be the empty string ("") in which case no runtime library selection flag will be added explicitly by CMake. Note that with*V isual Studio Generators* the native build system may choose to add its own default runtime library selection flag.

Use *generator expressions* to support per−configuration specification. For example, the code:

```
set(CMAKE_MSVC_RUNTIME_LIBRARY "MultiThreaded$<$<CONFIG:Debug>:Debug>")
```

selects for all following targets a multi−threaded statically−linked runtime library with or without debug information depending on the configuration.

If this variable is not set then the *MSVC_RUNTIME_LIBRARY* target property will not be set automatically. If that property is not set then CMake uses the default value **MultiThreaded$<$<CONFIG:Debug>:Debug>DLL** to select a MSVC runtime library.

NOTE:
This variable has effect only when policy *CMP0091* is set to **NEW** prior to the first *project()* or *enable_language()* command that enables a language using a compiler targeting the MSVC ABI.

CMAKE_MSVCIDE_RUN_PATH
Added in version 3.10.

Extra PATH locations that should be used when executing *add_custom_command()* or *add_custom_target()* when using *Visual Studio Generators*. This allows for running commands and using dll's that the IDE environment is not aware of.

If not set explicitly the value is initialized by the **CMAKE_MSVCIDE_RUN_PATH** environment variable, if set, and otherwise left empty.

CMAKE_NINJA_OUTPUT_PATH_PREFIX
Added in version 3.6.

Tell the *Ninja Generators* to add a prefix to every output path in **build.ninja**. A trailing slash is appended to the prefix, if missing.

This is useful when the generated ninja file is meant to be embedded as a **subninja** file into a *super* ninja project. For example, the command:

```
cd super−build−dir &&
cmake −G Ninja −S /path/to/src −B sub −DCMAKE_NINJA_OUTPUT_PATH_PREFIX=sub/
#                                  ^^^−−−−−−−−−− these match −−−−−−−−−−^^^
```

generates a build directory with its top−level (*CMAKE_BINARY_DIR*) in **super−build−dir/sub**. The path to the build directory ends in the output path prefix. This makes it suitable for use in a separately−written **super−build−dir/build.ninja** file with a directive like this:

```
subninja sub/build.ninja
```

The **auto−regeneration** rule in **super−build−dir/build.ninja** must have an order−only dependency on **sub/build.ninja**.

Added in version 3.27: The *Ninja Multi−Config* generator supports this variable.

NOTE:

When **CMAKE_NINJA_OUTPUT_PATH_PREFIX** is set, the project generated by CMake cannot be used as a standalone project. No default targets are specified.

The value of **CMAKE_NINJA_OUTPUT_PATH_PREFIX** must match one or more path components at the *end* of *CMAKE_BINARY_DIR*, or the behavior is undefined. However, this requirement is not checked automatically.

CMAKE_NO_BUILTIN_CHRPATH

Do not use the builtin binary editor to fix runtime library search paths on installation.

When an ELF or XCOFF binary needs to have a different runtime library search path after installation than it does in the build tree, CMake uses a builtin editor to change the runtime search path 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.

For more information on RPATH handling see the *INSTALL_RPATH* and *BUILD_RPATH* target properties.

Added in version 3.20: This variable also applies to XCOFF binaries' LIBPATH. Prior to the addition of the XCOFF editor in CMake 3.20, this variable applied only to ELF binaries' RPATH/RUNPATH.

CMAKE_NO_SYSTEM_FROM_IMPORTED

Default value for *NO_SYSTEM_FROM_IMPORTED* of targets.

This variable is used to initialize the *NO_SYSTEM_FROM_IMPORTED* property on all the targets. See that target property for additional information.

CMAKE_OPTIMIZE_DEPENDENCIES

Added in version 3.19.

Initializes the *OPTIMIZE_DEPENDENCIES* target property.

CMAKE_OSX_ARCHITECTURES

Target specific architectures for macOS and iOS.

This variable is used to initialize the *OSX_ARCHITECTURES* property on each target as it is created. See that target property for additional information.

The value of this variable should be set prior to the first *project()* or *enable_language()* command invocation because it may influence configuration of the toolchain and flags. It is intended to be set locally by the user creating a build tree. This variable should be set as a **CACHE** entry (or else CMake may remove it while initializing a cache entry of the same name) unless policy *CMP0126* is set to **NEW**.

Despite the **OSX** part in the variable name(s) they apply also to other SDKs than macOS like iOS, tvOS, visionOS, or watchOS.

This variable is ignored on platforms other than Apple.

CMAKE_OSX_DEPLOYMENT_TARGET

Specify the minimum version of the target platform (e.g. macOS or iOS) on which the target binaries are to be deployed. CMake uses this variable value for the **-mmacosx-version-min** flag or their respective target platform equivalents. For older Xcode versions that shipped multiple macOS SDKs this variable also helps to choose the SDK in case *CMAKE_OSX_SYSROOT* is unset.

If not set explicitly the value is initialized by the **MACOSX_DEPLOYMENT_TARGET** environment

variable, if set, and otherwise computed based on the host platform.

The value of this variable should be set prior to the first *project()* or *enable_language()* command invocation because it may influence configuration of the toolchain and flags. It is intended to be set locally by the user creating a build tree. This variable should be set as a **CACHE** entry (or else CMake may remove it while initializing a cache entry of the same name) unless policy *CMP0126* is set to **NEW**.

Despite the **OSX** part in the variable name(s) they apply also to other SDKs than macOS like iOS, tvOS, visionOS, or watchOS.

This variable is ignored on platforms other than Apple.

CMAKE_OSX_SYSROOT
Specify the location or name of the macOS platform SDK to be used. CMake uses this value to compute the value of the **−isysroot** flag or equivalent and to help the **find_*** commands locate files in the SDK.

If not set explicitly the value is initialized by the **SDKROOT** environment variable, if set, and otherwise computed based on the *CMAKE_OSX_DEPLOYMENT_TARGET* or the host platform.

The value of this variable should be set prior to the first *project()* or *enable_language()* command invocation because it may influence configuration of the toolchain and flags. It is intended to be set locally by the user creating a build tree. This variable should be set as a **CACHE** entry (or else CMake may remove it while initializing a cache entry of the same name) unless policy *CMP0126* is set to **NEW**.

Despite the **OSX** part in the variable name(s) they apply also to other SDKs than macOS like iOS, tvOS, visionOS, or watchOS.

This variable is ignored on platforms other than Apple.

CMAKE_PCH_INSTANTIATE_TEMPLATES
Added in version 3.19.

This variable is used to initialize the *PCH_INSTANTIATE_TEMPLATES* property of targets when they are created.

CMAKE_PCH_WARN_INVALID
Added in version 3.18.

This variable is used to initialize the *PCH_WARN_INVALID* property of targets when they are created.

CMAKE_PDB_OUTPUT_DIRECTORY
Output directory for MS debug symbol **.pdb** files generated by the linker for executable and shared library targets.

This variable is used to initialize the *PDB_OUTPUT_DIRECTORY* property on all the targets. See that target property for additional information.

CMAKE_PDB_OUTPUT_DIRECTORY_<CONFIG>
Per−configuration output directory for MS debug symbol **.pdb** files generated by the linker for executable and shared library targets.

This is a per−configuration version of *CMAKE_PDB_OUTPUT_DIRECTORY*. This variable is used to initialize the *PDB_OUTPUT_DIRECTORY_<CONFIG>* property on all the targets. See that target property for additional information.

CMAKE_PLATFORM_NO_VERSIONED_SONAME
Added in version 3.1.

This variable is used to globally control whether the *VERSION* and *SOVERSION* target properties should be used for shared libraries. When set to true, adding version information to each shared library target is disabled.

By default this variable is set only on platforms where CMake knows it is needed. On other platforms, the specified properties will be used for shared libraries.

CMAKE_POSITION_INDEPENDENT_CODE
Default value for *POSITION_INDEPENDENT_CODE* of targets.

This variable is used to initialize the *POSITION_INDEPENDENT_CODE* property on targets that are not **SHARED** or **MODULE** library targets. If set, its value is also used by the *try_compile()* command.

CMAKE_RUNTIME_OUTPUT_DIRECTORY
Where to put all the *RUNTIME* target files 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_RUNTIME_OUTPUT_DIRECTORY_<CONFIG>
Added in version 3.3.

Where to put all the *RUNTIME* target files when built for a specific configuration.

This variable is used to initialize the *RUNTIME_OUTPUT_DIRECTORY_<CONFIG>* property on all the targets. See that target property for additional information.

CMAKE_SHARED_LIBRARY_ENABLE_EXPORTS
Added in version 3.27.

Specify whether shared library generates an import file.

This variable is used to initialize the *ENABLE_EXPORTS* target property for shared library targets when they are created by calls to the *add_library()* command. See the property documentation for details.

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.

Handling Compiler Driver Differences
Added in version 4.0.

To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* and *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variables.

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

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes **"LINKER:SHELL:−z defs"**.

NOTE:
Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

This support implies to parse and re−quote the content of the variable. See policy *CMP0181*.

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.

Handling Compiler Driver Differences
Added in version 4.0.

To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* and *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variables.

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

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes **"LINKER:SHELL:−z defs"**.

NOTE:
Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

This support implies to parse and re−quote the content of the variable. See policy *CMP0181*.

CMAKE_SHARED_LINKER_FLAGS_<CONFIG>_INIT
Added in version 3.7.

Value used to initialize the *CMAKE_SHARED_LINKER_FLAGS_<CONFIG>* cache entry the first time a build tree is configured. This variable is meant to be set by a *toolchain file*. CMake may prepend or append content to the value based on the environment and target platform.

See also *CMAKE_SHARED_LINKER_FLAGS_INIT*.

CMAKE_SHARED_LINKER_FLAGS_INIT
Added in version 3.7.

Value used to initialize the *CMAKE_SHARED_LINKER_FLAGS* cache entry the first time a build tree is configured. This variable is meant to be set by a *toolchain file*. CMake may prepend or append content to the value based on the environment and target platform.

See also the configuration−specific variable *CMAKE_SHARED_LINKER_FLAGS_<CONFIG>_INIT*.

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.

This is used to initialize the *SKIP_BUILD_RPATH* target property for all targets. For more information on RPATH handling see the *INSTALL_RPATH* and *BUILD_RPATH* target properties.

See also the *CMAKE_SKIP_INSTALL_RPATH* variable. To omit RPATH in both the build and install steps, use *CMAKE_SKIP_RPATH* instead.

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.

See also the *CMAKE_SKIP_BUILD_RPATH* variable. To omit RPATH in both the build and install steps, use *CMAKE_SKIP_RPATH* instead.

For more information on RPATH handling see the *INSTALL_RPATH* and *BUILD_RPATH* target properties.

CMAKE_STATIC_LINKER_FLAGS
Flags to be used to create static libraries. These flags will be passed to the archiver when creating a static library.

See also *CMAKE_STATIC_LINKER_FLAGS_<CONFIG>*.

NOTE:
Static libraries do not actually link. They are essentially archives of object files. The use of the name "linker" in the name of this variable is kept for compatibility.

CMAKE_STATIC_LINKER_FLAGS_<CONFIG>
Flags to be used to create static libraries. These flags will be passed to the archiver when creating a static library in the **<CONFIG>** configuration.

See also *CMAKE_STATIC_LINKER_FLAGS*.

NOTE:
Static libraries do not actually link. They are essentially archives of object files. The use of the name "linker" in the name of this variable is kept for compatibility.

CMAKE_STATIC_LINKER_FLAGS_<CONFIG>_INIT
Added in version 3.7.

Value used to initialize the *CMAKE_STATIC_LINKER_FLAGS_<CONFIG>* cache entry the first time a build tree is configured. This variable is meant to be set by a *toolchain file*. CMake may prepend or append content to the value based on the environment and target platform.

See also *CMAKE_STATIC_LINKER_FLAGS_INIT*.

CMAKE_STATIC_LINKER_FLAGS_INIT
Added in version 3.7.

Value used to initialize the *CMAKE_STATIC_LINKER_FLAGS* cache entry the first time a build tree is configured. This variable is meant to be set by a *toolchain file*. CMake may prepend or append content to the value based on the environment and target platform.

See also the configuration−specific variable *CMAKE_STATIC_LINKER_FLAGS_<CONFIG>_INIT*.

CMAKE_TASKING_TOOLSET
Added in version 3.25.

Select the Tasking toolset which provides the compiler

Architecture compilers are provided by different toolchains with incompatible versioning schemes. Set this variable in a *toolchain file* so CMake can detect the compiler features correctly. If no toolset is specified, **Standalone** is assumed.

Due to the different versioning schemes, the compiler version (-*CMAKE_<LANG>_COMPILER_VERSION*) depends on the toolset and architecture in use. If projects can be built with multiple toolsets or architectures, the specified **CMAKE_TASKING_TOOLSET** and the automatically determined *CMAKE_<LANG>_COMPILER_ARCHITECTURE_ID* must be taken into account when comparing against the *CMAKE_<LANG>_COMPILER_VERSION*.

TriCore
Compilers are provided by the TriCore toolset.

SmartCode
Compilers are provided by the SmartCode toolset.

Standalone
Compilers are provided by the standalone toolsets.

NOTE:
For the TriCore architecture, the compiler from the TriCore toolset is selected as standalone compiler.

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_TRY_COMPILE_NO_PLATFORM_VARIABLES
Added in version 3.24.

Set to a true value to tell the *try_compile()* command not to propagate any platform variables into the test project.

The *try_compile()* command normally passes some CMake variables that configure the platform and toolchain behavior into test projects. See policy *CMP0137*. This variable may be set to disable that behavior.

CMAKE_TRY_COMPILE_PLATFORM_VARIABLES
Added in version 3.6.

List of variables that the *try_compile()* command source file signature must propagate into the test project in order to target the same platform as the host project.

This variable should not be set by project code. It is meant to be set by CMake's platform information modules for the current toolchain, or by a toolchain file when used with *CMAKE_TOOLCHAIN_FILE*.

Variables meaningful to CMake, such as *CMAKE_<LANG>_FLAGS*, are propagated automatically. The **CMAKE_TRY_COMPILE_PLATFORM_VARIABLES** variable may be set to pass custom variables meaningful to a toolchain file. For example, a toolchain file may contain:

```
set(CMAKE_SYSTEM_NAME ...)
set(CMAKE_TRY_COMPILE_PLATFORM_VARIABLES MY_CUSTOM_VARIABLE)
# ... use MY_CUSTOM_VARIABLE ...
```

If a user passes **-DMY_CUSTOM_VARIABLE=SomeValue** to CMake then this setting will be made visible to the toolchain file both for the main project and for test projects generated by the *try_compile()* command source file signature.

Changed in version 3.24: Listed variables are propagated to the *try_compile() whole-project* signature too. See *CMP0137*.

Added in version 3.24: The *CMAKE_TRY_COMPILE_NO_PLATFORM_VARIABLES* variable may be set to disable passing platform variables into the test project.

CMAKE_TRY_COMPILE_TARGET_TYPE
Added in version 3.6.

Type of target generated for *try_compile()* calls using the source file signature. Valid values are:

EXECUTABLE
Use *add_executable()* to name the source file in the generated project. This is the default if no value is given.

STATIC_LIBRARY
Use *add_library()* with the **STATIC** option to name the source file in the generated project. This avoids running the linker and is intended for use with cross-compiling toolchains that cannot link without custom flags or linker scripts.

CMAKE_UNITY_BUILD
Added in version 3.16.

This variable is used to initialize the *UNITY_BUILD* property of targets when they are created. Setting it to true enables batch compilation of multiple sources within each target. This feature is known as a *Unity* or *Jumbo* build.

Projects should not set this variable, it is intended as a developer control to be set on the *cmake(1)* command line or other equivalent methods. The developer must have the ability to enable or disable unity builds according to the capabilities of their own machine and compiler.

By default, this variable is not set, which will result in unity builds being disabled.

NOTE:
This option currently does not work well in combination with the

CMAKE_EXPORT_COMPILE_COMMANDS variable.

CMAKE_UNITY_BUILD_BATCH_SIZE
Added in version 3.16.

This variable is used to initialize the *UNITY_BUILD_BATCH_SIZE* property of targets when they are created. It specifies the default upper limit on the number of source files that may be combined in any one unity source file when unity builds are enabled for a target.

CMAKE_UNITY_BUILD_UNIQUE_ID
Added in version 3.20.

This variable is used to initialize the *UNITY_BUILD_UNIQUE_ID* property of targets when they are created. It specifies the name of the unique identifier generated per file in a unity build.

CMAKE_VERIFY_INTERFACE_HEADER_SETS
Added in version 3.24.

This variable is used to initialize the *VERIFY_INTERFACE_HEADER_SETS* property of targets when they are created. Setting it to true enables header set verification.

Projects should not normally set this variable, it is intended as a developer control to be set on the *cmake(1)* command line or other equivalent methods. The developer must have the ability to enable or disable header set verification according to the capabilities of their own machine and compiler.

Verification of a dependency's header sets is not typically of interest to developers. Therefore, *FetchContent_MakeAvailable()* explicitly sets **CMAKE_VERIFY_INTERFACE_HEADER_SETS** to false for the duration of its call, but restores its original value before returning. If a project brings a dependency directly into the main build (e.g. calling *add_subdirectory()* on a vendored project from a git submodule), it should also do likewise. For example:

```
# Save original setting so we can restore it later
set(want_header_set_verification ${CMAKE_VERIFY_INTERFACE_HEADER_SETS})

# Include the vendored dependency with header set verification disabled
set(CMAKE_VERIFY_INTERFACE_HEADER_SETS OFF)
add_subdirectory(...)   # Vendored sources, e.g. from git submodules

# Add the project's own sources. Restore the developer's original choice
# for whether to enable header set verification.
set(CMAKE_VERIFY_INTERFACE_HEADER_SETS ${want_header_set_verification})
add_subdirectory(src)
```

By default, this variable is not set, which will result in header set verification being disabled.

CMAKE_VISIBILITY_INLINES_HIDDEN
Default value for the *VISIBILITY_INLINES_HIDDEN* target property when a target is created.

CMAKE_VS_DEBUGGER_COMMAND
Added in version 3.27.

This variable is used to initialize the *VS_DEBUGGER_COMMAND* property on each target as it is created. See that target property for additional information.

CMAKE_VS_DEBUGGER_COMMAND_ARGUMENTS
Added in version 3.27.

This variable is used to initialize the *VS_DEBUGGER_COMMAND_ARGUMENTS* property on each target as it is created. See that target property for additional information.

CMAKE_VS_DEBUGGER_ENVIRONMENT
Added in version 3.27.

This variable is used to initialize the *VS_DEBUGGER_ENVIRONMENT* property on each target as it is created. See that target property for additional information.

CMAKE_VS_DEBUGGER_WORKING_DIRECTORY
Added in version 3.27.

This variable is used to initialize the *VS_DEBUGGER_WORKING_DIRECTORY* property on each target as it is created. See that target property for additional information.

CMAKE_VS_GLOBALS
Added in version 3.13.

List of **Key=Value** records to be set per target as target properties *VS_GLOBAL_<variable>* with **variable=Key** and value **Value**.

For example:

```
set(CMAKE_VS_GLOBALS
  "DefaultLanguage=en-US"
  "MinimumVisualStudioVersion=14.0"
  )
```

will set properties **VS_GLOBAL_DefaultLanguage** to **en-US** and **VS_GLOBAL_MinimumVisualStudioVersion** to **14.0** for all targets (except for **INTERFACE** libraries).

This variable is meant to be set by a *toolchain file*.

CMAKE_VS_INCLUDE_INSTALL_TO_DEFAULT_BUILD
Added in version 3.3.

Include **INSTALL** target to default build.

In Visual Studio solution, by default the **INSTALL** target will not be part of the default build. Setting this variable will enable the **INSTALL** target to be part of the default build.

CMAKE_VS_INCLUDE_PACKAGE_TO_DEFAULT_BUILD
Added in version 3.8.

Include **PACKAGE** target to default build.

In Visual Studio solution, by default the **PACKAGE** target will not be part of the default build. Setting this variable will enable the **PACKAGE** target to be part of the default build.

CMAKE_VS_JUST_MY_CODE_DEBUGGING
Added in version 3.15.

Enable Just My Code with Visual Studio debugger.

This variable is used to initialize the *VS_JUST_MY_CODE_DEBUGGING* property on all targets when they are created. See that target property for additional information.

CMAKE_VS_NO_COMPILE_BATCHING
Added in version 3.24.

Turn off compile batching when using *Visual Studio Generators*.

This variable is used to initialize the *VS_NO_COMPILE_BATCHING* property on all targets when they are created. See that target property for additional information.

Example
This shows setting the property for the target **foo** using the variable.

```
set(CMAKE_VS_NO_COMPILE_BATCHING ON)
add_library(foo SHARED foo.cpp)
```

CMAKE_VS_SDK_EXCLUDE_DIRECTORIES
Added in version 3.12.

This variable allows to override Visual Studio default Exclude Directories.

CMAKE_VS_SDK_EXECUTABLE_DIRECTORIES
Added in version 3.12.

This variable allows to override Visual Studio default Executable Directories.

CMAKE_VS_SDK_INCLUDE_DIRECTORIES
Added in version 3.12.

This variable allows to override Visual Studio default Include Directories.

CMAKE_VS_SDK_LIBRARY_DIRECTORIES
Added in version 3.12.

This variable allows to override Visual Studio default Library Directories.

CMAKE_VS_SDK_LIBRARY_WINRT_DIRECTORIES
Added in version 3.12.

This variable allows to override Visual Studio default Library WinRT Directories.

CMAKE_VS_SDK_REFERENCE_DIRECTORIES
Added in version 3.12.

This variable allows to override Visual Studio default Reference Directories.

CMAKE_VS_SDK_SOURCE_DIRECTORIES
Added in version 3.12.

This variable allows to override Visual Studio default Source Directories.

CMAKE_VS_WINRT_BY_DEFAULT
Added in version 3.13.

Inform *Visual Studio Generators* for VS 2010 and above that the target platform enables WinRT compilation by default and it needs to be explicitly disabled if **/ZW** or *VS_WINRT_COMPONENT* is omitted (as opposed to enabling it when either of those options is present)

This makes cmake configuration consistent in terms of WinRT among platforms − if you did not enable the WinRT compilation explicitly, it will be disabled (by either not enabling it or explicitly disabling it)

Note: WinRT compilation is always explicitly disabled for C language source files, even if it is expliclty enabled for a project

This variable is meant to be set by a *toolchain file* for such platforms.

CMAKE_WATCOM_RUNTIME_LIBRARY
Added in version 3.24.

Select the Watcom runtime library for use by compilers targeting the Watcom ABI. This variable is used to initialize the *WATCOM_RUNTIME_LIBRARY* property on all targets as they are created. It is also propagated by calls to the *try_compile()* command into the test project.

The allowed values are:

SingleThreaded
Compile without additional flags to use a single−threaded statically−linked runtime library.

SingleThreadedDLL
Compile with **−br** or equivalent flag(s) to use a single−threaded dynamically−linked runtime library. This is not available for Linux targets.

MultiThreaded
Compile with **−bm** or equivalent flag(s) to use a multi−threaded statically−linked runtime library.

MultiThreadedDLL
Compile with **−bm −br** or equivalent flag(s) to use a multi−threaded dynamically−linked runtime library. This is not available for Linux targets.

The value is ignored on non−Watcom compilers but an unsupported value will be rejected as an error when using a compiler targeting the Watcom ABI.

The value may also be the empty string ("") in which case no runtime library selection flag will be added explicitly by CMake.

Use *generator expressions* to support per−configuration specification.

For example, the code:

```
set(CMAKE_WATCOM_RUNTIME_LIBRARY "MultiThreaded")
```

selects for all following targets a multi−threaded statically−linked runtime library.

If this variable is not set then the *WATCOM_RUNTIME_LIBRARY* target property will not be set automatically. If that property is not set then CMake uses the default value **MultiThreadedDLL** on Windows and **SingleThreaded** on other platforms to select a Watcom runtime library.

NOTE:
This variable has effect only when policy *CMP0136* is set to **NEW** prior to the first *project()* or *enable_language()* command that enables a language using a compiler targeting the Watcom ABI.

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.

CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS
Added in version 3.4.

Default value for *WINDOWS_EXPORT_ALL_SYMBOLS* target property. This variable is used to initialize the property on each target as it is created.

CMAKE_XCODE_ATTRIBUTE_<an−attribute>
Added in version 3.1.

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.

This offers low−level control over the generated Xcode project file. It is meant as a last resort for specifying settings that CMake does not otherwise have a way to control. Although this can override a setting CMake normally produces on its own, doing so bypasses CMake's model of the project and can break things.

See the *XCODE_ATTRIBUTE_<an−attribute>* target property to set attributes on a specific target.

Contents of **CMAKE_XCODE_ATTRIBUTE_<an−attribute>** may use "generator expressions" with the syntax **$<...>**. See the*cmak e−generator−expressions(7)* manual for available expressions. See the *cmake−buildsystem(7)* manual for more on defining buildsystem properties.

EXECUTABLE_OUTPUT_PATH
Old executable location variable.

The target property *RUNTIME_OUTPUT_DIRECTORY* supersedes 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* supersede this variable for a target if they are set. Library targets are otherwise placed in this directory.

VARIABLES FOR LANGUAGES

CMAKE_C_COMPILE_FEATURES

Added in version 3.1.

List of features known to the C compiler

These features are known to be available for use with the C compiler. This list is a subset of the features listed in the *CMAKE_C_KNOWN_FEATURES* global property.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_C_EXTENSIONS

Added in version 3.1.

Default value for *C_EXTENSIONS* target property if set when a target is created.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_C_STANDARD

Added in version 3.1.

Default value for *C_STANDARD* target property if set when a target is created.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_C_STANDARD_REQUIRED

Added in version 3.1.

Default value for *C_STANDARD_REQUIRED* target property if set when a target is created.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_CUDA_ARCHITECTURES

Added in version 3.18.

Default value for *CUDA_ARCHITECTURES* property of targets.

Initialized by the *CUDAARCHS* environment variable if set. Otherwise as follows depending on *CMAKE_CUDA_COMPILER_ID*:

- For **Clang**: the oldest architecture that works.
- For **NVIDIA**: the default architecture chosen by the compiler. See policy *CMP0104*.

Users are encouraged to override this, as the default varies across compilers and compiler versions.

This variable is used to initialize the *CUDA_ARCHITECTURES* property on all targets. See the target property for additional information.

Examples

```
cmake_minimum_required(VERSION)

if(NOT DEFINED CMAKE_CUDA_ARCHITECTURES)
  set(CMAKE_CUDA_ARCHITECTURES 75)
endif()

project(example LANGUAGES CUDA)
```

CMAKE_CUDA_ARCHITECTURES will default to **75** unless overridden by the user.

CMAKE_CUDA_COMPILE_FEATURES
Added in version 3.17.

List of features known to the CUDA compiler

These features are known to be available for use with the CUDA compiler. This list is a subset of the features listed in the *CMAKE_CUDA_KNOWN_FEATURES* global property.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_CUDA_EXTENSIONS
Added in version 3.8.

Default value for *CUDA_EXTENSIONS* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_CUDA_HOST_COMPILER
Added in version 3.10.

This is the original CUDA−specific name for the more general *CMAKE_<LANG>_HOST_COMPILER* variable. See the latter for details.

CMAKE_CUDA_STANDARD
Added in version 3.8.

Default value for *CUDA_STANDARD* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_CUDA_STANDARD_REQUIRED
Added in version 3.8.

Default value for *CUDA_STANDARD_REQUIRED* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_CUDA_TOOLKIT_INCLUDE_DIRECTORIES
Added in version 3.8.

When the **CUDA** language has been enabled, this provides a *semicolon−separated list* of include directories provided by the CUDA Toolkit. The value may be useful for C++ source files to include CUDA headers.

CMAKE_CXX_COMPILE_FEATURES
Added in version 3.1.

List of features known to the C++ compiler

These features are known to be available for use with the C++ compiler. This list is a subset of the features listed in the *CMAKE_CXX_KNOWN_FEATURES* global property.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_CXX_COMPILER_IMPORT_STD
Added in version 3.30.

A list of C++ standard levels for which **import std** support exists for the current C++ toolchain. Support for C++<NN> may be detected using a **<NN> IN_LIST CMAKE_CXX_COMPILER_IMPORT_STD** predicate with the *if()* command.

NOTE:
This variable is meaningful only when experimental support for **import std;** has been enabled by the **CMAKE_EXPERIMENTAL_CXX_IMPORT_STD** gate.

CMAKE_CXX_EXTENSIONS
Added in version 3.1.

Default value for *CXX_EXTENSIONS* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_CXX_STANDARD
Added in version 3.1.

Default value for *CXX_STANDARD* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_CXX_STANDARD_REQUIRED
Added in version 3.1.

Default value for *CXX_STANDARD_REQUIRED* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported

compilers.

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_HIP_ARCHITECTURES
Added in version 3.21.

List of GPU architectures to for which to generate device code. Architecture names are interpreted based on *CMAKE_HIP_PLATFORM*.

This is initialized based on the value of *CMAKE_HIP_PLATFORM*:

amd Uses architectures reported by **rocm_agent_enumerator**, if available, and otherwise to a default chosen by the compiler.

This variable is used to initialize the *HIP_ARCHITECTURES* property on all targets. See the target property for additional information.

CMAKE_HIP_COMPILE_FEATURES
Added in version 3.21.

List of features known to the HIP compiler

These features are known to be available for use with the HIP compiler. This list is a subset of the features listed in the *CMAKE_HIP_KNOWN_FEATURES* global property.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_HIP_EXTENSIONS
Added in version 3.21.

Default value for *HIP_EXTENSIONS* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_HIP_PLATFORM
Added in version 3.28.

GPU platform for which HIP language sources are to be compiled.

The value must be one of:

amd AMD GPUs

nvidia NVIDIA GPUs

If not specified, a default is computed via **hipconfig −−platform**.

CMAKE_HIP_ARCHITECTURES entries are interpreted with as architectures of the GPU platform.

CMAKE_HIP_COMPILER must target the same GPU platform.

CMAKE_HIP_STANDARD
Added in version 3.21.

Default value for *HIP_STANDARD* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_HIP_STANDARD_REQUIRED
Added in version 3.21.

Default value for *HIP_STANDARD_REQUIRED* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_ISPC_HEADER_DIRECTORY
Added in version 3.19.

ISPC generated header output directory.

This variable is used to initialize the *ISPC_HEADER_DIRECTORY* property on all the targets. See the target property for additional information.

CMAKE_ISPC_HEADER_SUFFIX
Added in version 3.19.2.

Output suffix to be used for ISPC generated headers.

This variable is used to initialize the *ISPC_HEADER_SUFFIX* property on all the targets. See the target property for additional information.

CMAKE_ISPC_INSTRUCTION_SETS
Added in version 3.19.

Default value for *ISPC_INSTRUCTION_SETS* property of targets.

This variable is used to initialize the *ISPC_INSTRUCTION_SETS* property on all targets. See the target property for additional information.

CMAKE_<LANG>_ANDROID_TOOLCHAIN_MACHINE
Added in version 3.7.1.

When *Cross Compiling for Android* this variable contains the toolchain binutils machine name (e.g. **gcc -dumpmachine**). The binutils typically have a **<machine>-** prefix on their name.

See also *CMAKE_<LANG>_ANDROID_TOOLCHAIN_PREFIX* and *CMAKE_<LANG>_ANDROID_TOOLCHAIN_SUFFIX*.

CMAKE_<LANG>_ANDROID_TOOLCHAIN_PREFIX
Added in version 3.7.

When *Cross Compiling for Android* this variable contains the absolute path prefixing the toolchain GNU compiler and its binutils.

See also *CMAKE_<LANG>_ANDROID_TOOLCHAIN_SUFFIX* and *CMAKE_<LANG>_ANDROID_TOOLCHAIN_MACHINE*.

For example, the path to the linker is:

```
${CMAKE_CXX_ANDROID_TOOLCHAIN_PREFIX}ld${CMAKE_CXX_ANDROID_TOOLCHAIN_SUFFIX}
```

CMAKE_<LANG>_ANDROID_TOOLCHAIN_SUFFIX
Added in version 3.7.

When *Cross Compiling for Android* this variable contains the host platform suffix of the toolchain GNU compiler and its binutils.

See also *CMAKE_<LANG>_ANDROID_TOOLCHAIN_PREFIX* and *CMAKE_<LANG>_ANDROID_TOOLCHAIN_MACHINE*.

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_CREA TE* 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_CREA TE* and *CMAKE_<LANG>_ARCHIVE_APPEND*.

CMAKE_<LANG>_ARCHIVER_WRAPPER_FLAG
Added in version 4.0.

Defines the syntax of compiler driver option to pass options to the archiver tool. It will be used to translate the **ARCHIVER:** prefix in the static library options (see *STATIC_LIBRARY_OPTIONS*).

This variable holds a *semicolon−separated list* of tokens. If a space (i.e. " ") is specified as last token, flag and **ARCHIVER:** arguments will be specified as separate arguments to the compiler driver. The *CMAKE_<LANG>_ARCHIVER_WRAPPER_FLAG_SEP* variable can be specified to manage concatenation of arguments.

See *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* variable for examples of definitions because **CMAKE_<LANG>_ARCHIVER_WRAPPER_FLAG** use the same syntax.

CMAKE_<LANG>_ARCHIVER_WRAPPER_FLAG_SEP
Added in version 4.0.

This variable is used with *CMAKE_<LANG>_ARCHIVER_WRAPPER_FLAG* variable to format **ARCHIVER:** prefix in the static library options (see *STATIC_LIBRARY_OPTIONS*).

When specified, arguments of the **ARCHIVER:** prefix will be concatenated using this value as separator.

CMAKE_<LANG>_BYTE_ORDER
Added in version 3.20.

Byte order of **<LANG>** compiler target architecture, if known. If defined and not empty, the value is one of:

BIG_ENDIAN
The target architecture is Big Endian.

LITTLE_ENDIAN
The target architecture is Little Endian.

This is defined for languages **C**, **CXX**, **OBJC**, **OBJCXX**, and **CUDA**.

If *CMAKE_OSX_ARCHITECTURES* specifies multiple architectures, the value of **CMAKE_<LANG>_BYTE_ORDER** is non−empty only if all architectures share the same byte order.

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>_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.

Usage
This variable can be set by the user during the first time a build tree is configured.

If a non−full path value is supplied then CMake will resolve the full path of the compiler.

The variable could be set in a user supplied toolchain file or via *−D* on the command line.

NOTE:
Options that are required to make the compiler work correctly can be included as items in a list; they can not be changed.

```
#set within user supplied toolchain file
set(CMAKE_C_COMPILER /full/path/to/qcc --arg1 --arg2)
```

or

```
$ cmake ... -DCMAKE_C_COMPILER='qcc;--arg1;--arg2'
```

CMAKE_<LANG>_COMPILER_EXTERNAL_TOOLCHAIN
The external toolchain for cross-compiling, if supported.

Some compiler toolchains do not ship their own auxiliary utilities such as archivers and linkers. The compiler driver may support a command-line argument to specify the location of such tools. **CMAKE_<LANG>_COMPILER_EXTERNAL_TOOLCHAIN** may be set to a path to the external toolchain and will be passed to the compiler driver if supported.

This variable may only be set in a toolchain file specified by the *CMAKE_TOOLCHAIN_FILE* variable.

CMAKE_<LANG>_COMPILER_ID
Compiler identification string.

A short string unique to the compiler vendor. Possible values include:

Value	Name
Absoft	Absoft Fortran
ADSP	Analog VisualDSP++
AppleClang	Apple Clang
ARMCC	ARM Compiler
ARMClang	ARM Compiler based on Clang
Bruce	Bruce C Compiler
CCur	Concurrent Fortran
Clang	*LLVM Clang*
Cray	Cray Compiler
CrayClang	Cray Clang−based Compiler
Embarcadero, **Borland**	*Embarcadero*
Flang	*Classic Flang Fortran Compiler*
LLVMFlang	*LLVM Flang Fortran Compiler*
Fujitsu	Fujitsu HPC compiler (Trad mode)
FujitsuClang	Fujitsu HPC compiler (Clang mode)
G95	*G95 Fortran*
GNU	*GNU Compiler Collection*
GHS	*Green Hills Software*
HP	Hewlett−Packard Compiler
IAR	IAR Systems
Intel	Intel Classic Compiler
IntelLLVM	*Intel LLVM−Based Compiler*
LCC	MCST Elbrus C/C++/Fortran Compiler
LFortran	LFortran Fortran Compiler
MSVC	*Microsoft Visual Studio*
NVHPC	*NVIDIA HPC Compiler*
NVIDIA	*NVIDIA CUDA Compiler*
OrangeC	*OrangeC Compiler*
OpenWatcom	*Open Watcom*
PGI	The Portland Group
PathScale	PathScale
SDCC	*Small Device C Compiler*
SunPro	Oracle Developer Studio
Tasking	*Tasking Compiler Toolsets*
TI	Texas Instruments
TIClang	*Texas Instruments Clang−based Compilers*
TinyCC	*Tiny C Compiler*
XL, **VisualAge**, **zOS**	IBM XL
XLClang	IBM Clang−based XL
IBMClang	IBM LLVM−based Compiler

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_PREDEFINES_COMMAND
Added in version 3.10.

Command that outputs the compiler pre definitions.

See *AUTOMOC* which uses *CMAKE_CXX_COMPILER_PREDEFINES_COMMAND* to generate the *AUTOMOC_COMPILER_PREDEFINES*.

CMAKE_<LANG>_COMPILER_TARGET
The target for cross−compiling, if supported.

Some compiler drivers are inherently cross−compilers, such as clang and QNX qcc. These compiler drivers support a command−line argument to specify the target to cross−compile for.

This variable may only be set in a toolchain file specified by the *CMAKE_TOOLCHAIN_FILE* variable.

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.

For example **CMAKE_C_COMPILER_VERSION** and **CMAKE_CXX_COMPILER_VERSION** might indicate the respective C and C++ compiler version.

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>**. This rule variable is a **;** delimited list of commands to run to perform the linking step.

CMAKE_<LANG>_CREATE_SHARED_LIBRARY_ARCHIVE
Added in version 3.31.

Rule variable to create a shared library with archive.

This is a rule variable that tells CMake how to create a shared library with an archive for the language <LANG>. This rule variable is a ; delimited list of commands to run to perform the linking step.

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>**. This rule variable is a **;** delimited list of commands to run.

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>_EXTENSIONS
The variations are:

- *CMAKE_C_EXTENSIONS*
- *CMAKE_CXX_EXTENSIONS*
- *CMAKE_CUDA_EXTENSIONS*
- *CMAKE_HIP_EXTENSIONS*
- *CMAKE_OBJC_EXTENSIONS*
- *CMAKE_OBJCXX_EXTENSIONS*

Default values for *<LANG>_EXTENSIONS* target properties if set when a target is created. For the compiler's default setting see *CMAKE_<LANG>_EXTENSIONS_DEFAULT*.

For supported CMake versions see the respective pages.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_<LANG>_EXTENSIONS_DEFAULT
Added in version 3.22.

Compiler's default extensions mode. Used as the default for the *<LANG>_EXTENSIONS* target property when *CMAKE_<LANG>_EXTENSIONS* is not set (see *CMP0128*).

This variable is read−only. Modifying it is undefined behavior.

CMAKE_<LANG>_FLAGS
Language−wide flags for language **<LANG>** used when building for all configurations. These flags will be passed to all invocations of the compiler. This includes invocations that drive compiling and those that drive linking.

For each language, if this variable is not defined, it is initialized and stored in the cache using values from environment variables in combination with CMake's builtin defaults for the toolchain:

- **CMAKE_C_FLAGS**: Initialized by the *CFLAGS* environment variable.
- **CMAKE_CXX_FLAGS**: Initialized by the *CXXFLAGS* environment variable.
- **CMAKE_CUDA_FLAGS**: Initialized by the *CUDAFLAGS* environment variable.
- **CMAKE_Fortran_FLAGS**: Initialized by the *FFLAGS* environment variable.
- **CMAKE_CSharp_FLAGS**: Initialized by the *CSFLAGS* environment variable.
- **CMAKE_HIP_FLAGS**: Initialized by the *HIPFLAGS* environment variable.
- **CMAKE_ISPC_FLAGS**: Initialized by the *ISPCFLAGS* environment variable.
- **CMAKE_OBJC_FLAGS**: Initialized by the *OBJCFLAGS* environment variable.
- **CMAKE_OBJCXX_FLAGS**: Initialized by the *OBJCXXFLAGS* environment variable.

This value is a command−line string fragment. Therefore, multiple options should be separated by spaces, and options with spaces should be quoted.

The flags in this variable will be passed before those in the per−configuration *CMAKE_<LANG>_FLAGS_<CONFIG>* variable. On invocations driving compiling, flags from both variables will be passed before flags added by commands such as *add_compile_options()* and *target_compile_options()*. On invocations driving linking, they will be passed before flags added by

commands such as *add_link_options()* and *target_link_options()*.

CMAKE_<LANG>_FLAGS_<CONFIG>
Language−wide flags for language **<LANG>** used when building for the **<CONFIG>** configuration. These flags will be passed to all invocations of the compiler in the corresponding configuration. This includes invocations that drive compiling and those that drive linking.

The flags in this variable will be passed after those in the *CMAKE_<LANG>_FLAGS* variable. On invocations driving compiling, flags from both variables will be passed before flags added by commands such as *add_compile_options()* and *target_compile_options()*. On invocations driving linking, they will be passed before flags added by commands such as *add_link_options()* and *target_link_options()*.

CMAKE_<LANG>_FLAGS_<CONFIG>_INIT
Added in version 3.11.

Value used to initialize the *CMAKE_<LANG>_FLAGS_<CONFIG>* cache entry the first time a build tree is configured for language **<LANG>**. This variable is meant to be set by a *toolchain file*. CMake may prepend or append content to the value based on the environment and target platform.

See also *CMAKE_<LANG>_FLAGS_INIT*.

CMAKE_<LANG>_FLAGS_DEBUG
This variable is the **Debug** variant of the *CMAKE_<LANG>_FLAGS_<CONFIG>* variable.

CMAKE_<LANG>_FLAGS_DEBUG_INIT
Added in version 3.7.

This variable is the **Debug** variant of the *CMAKE_<LANG>_FLAGS_<CONFIG>_INIT* variable.

CMAKE_<LANG>_FLAGS_INIT
Added in version 3.7.

Value used to initialize the *CMAKE_<LANG>_FLAGS* cache entry the first time a build tree is configured for language **<LANG>**. This variable is meant to be set by a *toolchain file*. CMake may prepend or append content to the value based on the environment and target platform. For example, the contents of a **xxxFLAGS** environment variable will be prepended, where **xxx** will be language−specific but not necessarily the same as **<LANG>** (e.g. *CXXFLAGS* for **CXX**, *FFLAGS* for **Fortran**, and so on). This value is a command−line string fragment. Therefore, multiple options should be separated by spaces, and options with spaces should be quoted.

See also the configuration−specific *CMAKE_<LANG>_FLAGS_<CONFIG>_INIT* variable.

CMAKE_<LANG>_FLAGS_MINSIZEREL
This variable is the **MinSizeRel** variant of the *CMAKE_<LANG>_FLAGS_<CONFIG>* variable.

CMAKE_<LANG>_FLAGS_MINSIZEREL_INIT
Added in version 3.7.

This variable is the **MinSizeRel** variant of the *CMAKE_<LANG>_FLAGS_<CONFIG>_INIT* variable.

CMAKE_<LANG>_FLAGS_RELEASE
This variable is the **Release** variant of the *CMAKE_<LANG>_FLAGS_<CONFIG>* variable.

CMAKE_<LANG>_FLAGS_RELEASE_INIT
Added in version 3.7.

This variable is the **Release** variant of the *CMAKE_<LANG>_FLAGS_<CONFIG>_INIT* variable.

CMAKE_<LANG>_FLAGS_RELWITHDEBINFO
This variable is the **RelWithDebInfo** variant of the *CMAKE_<LANG>_FLAGS_<CONFIG>* variable.

CMAKE_<LANG>_FLAGS_RELWITHDEBINFO_INIT
Added in version 3.7.

This variable is the **RelWithDebInfo** variant of the *CMAKE_<LANG>_FLAGS_<CONFIG>_INIT* variable.

CMAKE_<LANG>_HOST_COMPILER
Added in version 3.10: **CMAKE_CUDA_HOST_COMPILER**

Added in version 3.28: **CMAKE_HIP_HOST_COMPILER**

This variable is available when **<LANG>** is **CUDA** or **HIP**.

When *CMAKE_<LANG>_COMPILER_ID* is **NVIDIA**, **CMAKE_<LANG>_HOST_COMPILER** selects the compiler executable to use when compiling host code for **CUDA** or **HIP** language files. This maps to the **nvcc −ccbin** option.

The **CMAKE_<LANG>_HOST_COMPILER** variable may be set explicitly before CUDA or HIP is first enabled by a *project()* or *enable_language()* command. This can be done via **−DCMAKE_<LANG>_HOST_COMPILER=...** on the command line or in a *toolchain file*. Or, one may set the *CUDAHOSTCXX* or *HIPHOSTCXX* environment variable to provide a default value.

Once the CUDA or HIP language is enabled, the **CMAKE_<LANG>_HOST_COMPILER** variable is read−only and changes to it are undefined behavior.

NOTE:
Since **CMAKE_<LANG>_HOST_COMPILER** is meaningful only when the *CMAKE_<LANG>_COMPILER_ID* is **NVIDIA**, it does not make sense to set **CMAKE_<LANG>_HOST_COMPILER** without also setting **CMAKE_<LANG>_COMPILER** to NVCC.

NOTE:
Projects should not try to set **CMAKE_<LANG>_HOST_COMPILER** to match *CMAKE_CXX_COMPILER* themselves. It is the end−user's responsibility, not the project's, to ensure that NVCC targets the same ABI as the C++ compiler.

NOTE:
Ignored when using *Visual Studio Generators*.

See the *CMAKE_<LANG>_HOST_COMPILER_ID* and *CMAKE_<LANG>_HOST_COMPILER_VERSION* variables for information about the host compiler used by **nvcc**, whether by default or specified by **CMAKE_<LANG>_HOST_COMPILER**.

CMAKE_<LANG>_HOST_COMPILER_ID
Added in version 3.31.

This variable is available when **<LANG>** is **CUDA** or **HIP** and *CMAKE_<LANG>_COMPILER_ID* is **NVIDIA**. It contains the identity of the host compiler invoked by **nvcc**, either by default or as specified by

CMAKE_<LANG>_HOST_COMPILER, among possibilities documented by *CMAKE_<LANG>_COMPILER_ID*.

CMAKE_<LANG>_HOST_COMPILER_VERSION
Added in version 3.31.

This variable is available when **<LANG>** is **CUDA** or **HIP** and *CMAKE_<LANG>_COMPILER_ID* is **NVIDIA**. It contains the version of the host compiler invoked by **nvcc**, either by default or as specified by *CMAKE_<LANG>_HOST_COMPILER*, in the same format as *CMAKE_<LANG>_COMPILER_VERSION*.

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, which is important for **C**, **CXX**, and **CUDA** to avoid overriding standard library headers.

This value is not used for **Fortran** because it has no standard library headers and some compilers do not search their implicit include directories for module **.mod** files.

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.

For each language enabled by the *project()* or *enable_language()* command, CMake automatically detects these directories and reports the results in this variable. The *CMAKE_<LANG>_IMPLICIT_LINK_DIRECTORIES_EXCLUDE* environment variable may be set to exclude specific directories from the automatically detected results.

When linking to a static library, CMake adds the implicit link directories from this variable for each language used in the static library (except the language whose compiler is used to drive linking). In the case of an imported static library, the *IMPORTED_LINK_INTERFACE_LANGUAGES* target property lists the languages whose implicit link information is needed. If any of the languages is not enabled, its value for the **CMAKE_<LANG>_IMPLICIT_LINK_DIRECTORIES** variable may instead be provided by the project. Or, a *toolchain file* may set the variable to a value known for the specified toolchain. It will either be overridden when the language is enabled, or used as a fallback.

Some toolchains read implicit directories from an environment variable such as **LIBRARY_PATH**. If using such an environment variable, keep its value consistent when operating in a given build tree because CMake saves the value detected when first creating a build tree.

In CMake versions prior to 4.0, if policy *CMP0060* is not set to **NEW**, then 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 for historical purposes.

See also the *CMAKE_<LANG>_IMPLICIT_LINK_LIBRARIES* variable.

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. For each language enabled by the *project()* or *enable_language()* command, CMake automatically detects these libraries and flags and reports the results in this variable.

When linking to a static library, CMake adds the implicit link libraries and flags from this variable for each language used in the static library (except the language whose compiler is used to drive linking). In the case of an imported static library, the *IMPORTED_LINK_INTERFACE_LANGUAGES* target property lists the languages whose implicit link information is needed. If any of the languages is not enabled, its value for the **CMAKE_<LANG>_IMPLICIT_LINK_LIBRARIES** variable may instead be provided by the project. Or, a *toolchain file* may set the variable to a value known for the specified toolchain. It will either be overridden when the language is enabled, or used as a fallback.

See also the *CMAKE_<LANG>_IMPLICIT_LINK_DIRECTORIES* 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>_LINK_EXECUTABLE
Rule variable to link an executable.

Rule variable to link an executable for the given language.

CMAKE_<LANG>_LINKER_WRAPPER_FLAG
Added in version 3.13.

Defines the syntax of compiler driver option to pass options to the linker tool. It will be used to translate the **LINKER:** prefix in the link options (see *add_link_options()* and *target_link_options()*).

This variable holds a *semicolon−separated list* of tokens. If a space (i.e. " ") is specified as last token, flag and **LINKER:** arguments will be specified as separate arguments to the compiler driver. The *CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP* variable can be specified to manage concatenation of arguments.

For example, for **Clang** we have:

```
set (CMAKE_C_LINKER_WRAPPER_FLAG "-Xlinker" " ")
```

Specifying "**LINKER:−z,defs**" will be transformed in **−Xlinker −z −Xlinker defs**.

For **GNU GCC**:

```
set (CMAKE_C_LINKER_WRAPPER_FLAG "-Wl,")
set (CMAKE_C_LINKER_WRAPPER_FLAG_SEP ",")
```

Specifying "**LINKER:−z,defs**" will be transformed in **−Wl,−z,defs**.

And for **SunPro**:

```
set (CMAKE_C_LINKER_WRAPPER_FLAG "-Qoption" "ld" " ")
set (CMAKE_C_LINKER_WRAPPER_FLAG_SEP ",")
```

Specifying "**LINKER:−z,defs**" will be transformed in **−Qoption ld −z,defs**.

CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP
Added in version 3.13.

This variable is used with *CMAKE_<LANG>_LINKER_WRAPPER_FLAG* variable to format **LINKER:** prefix in the link options (see *add_link_options()* and *target_link_options()*).

When specified, arguments of the **LINKER:** prefix will be concatenated using this value as separator.

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>_SIMULATE_ID
Identification string of the "simulated" compiler.

Some compilers simulate other compilers to serve as drop−in replacements. When CMake detects such a compiler it sets this variable to what would have been the *CMAKE_<LANG>_COMPILER_ID* for the simulated compiler.

NOTE:
In other words, this variable describes the ABI compatibility of the generated code.

CMAKE_<LANG>_SIMULATE_VERSION
Version string of "simulated" compiler.

Some compilers simulate other compilers to serve as drop−in replacements. When CMake detects such a compiler it sets this variable to what would have been the *CMAKE_<LANG>_COMPILER_VERSION* for the simulated compiler.

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_<LANG>_STANDARD
The variations are:

- *CMAKE_C_STANDARD*
- *CMAKE_CXX_STANDARD*

- *CMAKE_CUDA_STANDARD*
- *CMAKE_HIP_STANDARD*
- *CMAKE_OBJC_STANDARD*
- *CMAKE_OBJCXX_STANDARD*

Default values for *<LANG>_STANDARD* target properties if set when a target is created.

For supported CMake versions see the respective pages.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_<LANG>_STANDARD_DEFAULT
Added in version 3.9.

The compiler's default standard for the language **<LANG>**. Empty if the compiler has no conception of standard levels.

CMAKE_<LANG>_STANDARD_INCLUDE_DIRECTORIES
Added in version 3.6.

Include directories to be used for every source file compiled with the **<LANG>** compiler. This is meant for specification of system include directories needed by the language for the current platform. The directories always appear at the end of the include path passed to the compiler.

This variable should not be set by project code. It is meant to be set by CMake's platform information modules for the current toolchain, or by a toolchain file when used with *CMAKE_TOOLCHAIN_FILE*.

See also *CMAKE_<LANG>_STANDARD_LIBRARIES*.

CMAKE_<LANG>_STANDARD_LATEST
Added in version 3.30.

This variable represents the minimum between the latest version of the standard for language **<LANG>** which is supported by the current compiler and the latest version which is supported by CMake. Its value will be set to one of the supported values of the corresponding *<LANG>_STANDARD* target property; see the documentation of that property for a list of supported languages.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

NOTE:
CMAKE_<LANG>_STANDARD_LATEST will never be set to a language standard which CMake recognizes but provides no support for. Unless explicitly stated otherwise, every value which is supported by the corresponding *<LANG>_STANDARD* target property represents a standard of language **<LANG>** which is both recognized and supported by CMake.

Checking for Language Standard Support
It is possible to use the value of the **CMAKE_<LANG>_STANDARD_LATEST** variable to check for language standard support. This can be used to, e.g., conditionally enable optional features for a distributed library.

When doing so, one should be careful to **not** rely on integer value comparisons between standard levels.

This is because some older standards of a given language which are supported by CMake (e.g., C++98, represented as **98**) will have a higher numerical value than newer standards of that same language.

The following code sample demonstrates how one might correctly check for C++17 support:

```
# Careful! We cannot do direct integer comparisons with
# CMAKE_CXX_STANDARD_LATEST because some earlier C++ standards (e.g.,
# C++98) will have a higher numerical value than our requirement (C++17).
#
# Instead, we keep a list of unsupported C++ standards and check if
# CMAKE_CXX_STANDARD_LATEST appears in that list.
set(UNSUPPORTED_CXX_STANDARDS
  98
  11
  14
)

list(FIND UNSUPPORTED_CXX_STANDARDS ${CMAKE_CXX_STANDARD_LATEST} UNSUPPORTED_CXX_

if(UNSUPPORTED_CXX_STANDARDS_INDEX EQUAL -1)
  # We know that the current compiler supports at least C++17. Enabling
  # some optional feature...
else()
  message(STATUS
    "Feature X is disabled because it requires C++17, but the current "
    "compiler only supports C++${CMAKE_CXX_STANDARD_LATEST}."
  )
endif()
```

CMAKE_<LANG>_STANDARD_LIBRARIES
Added in version 3.6.

Libraries linked into every executable and shared library linked for language **<LANG>**. This is meant for specification of system libraries needed by the language for the current platform.

This variable should not be set by project code. It is meant to be set by CMake's platform information modules for the current toolchain, or by a toolchain file when used with *CMAKE_TOOLCHAIN_FILE*.

See also *CMAKE_<LANG>_STANDARD_INCLUDE_DIRECTORIES*.

CMAKE_<LANG>_STANDARD_LINK_DIRECTORIES
Added in version 3.31.

Link directories specified for every executable and library linked for language **<LANG>**. This is meant for specification of system link directories needed by the language for the current platform.

This variable should not be set by project code. It is meant to be set by CMake's platform information modules for the current toolchain, or by a toolchain file when used with *CMAKE_TOOLCHAIN_FILE*.

See also *CMAKE_<LANG>_STANDARD_LIBRARIES*.

CMAKE_<LANG>_STANDARD_REQUIRED
The variations are:

- *CMAKE_C_STANDARD_REQUIRED*
- *CMAKE_CXX_STANDARD_REQUIRED*
- *CMAKE_CUDA_STANDARD_REQUIRED*
- *CMAKE_HIP_STANDARD_REQUIRED*
- *CMAKE_OBJC_STANDARD_REQUIRED*
- *CMAKE_OBJCXX_STANDARD_REQUIRED*

Default values for *<LANG>_STANDARD_REQUIRED* target properties if set when a target is created.

For supported CMake versions see the respective pages.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_OBJC_EXTENSIONS
Added in version 3.16.

Default value for *OBJC_EXTENSIONS* target property if set when a target is created.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_OBJC_STANDARD
Added in version 3.16.

Default value for *OBJC_STANDARD* target property if set when a target is created.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_OBJC_STANDARD_REQUIRED
Added in version 3.16.

Default value for *OBJC_STANDARD_REQUIRED* target property if set when a target is created.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_OBJCXX_EXTENSIONS
Added in version 3.16.

Default value for *OBJCXX_EXTENSIONS* target property if set when a target is created.

See the *cmake-compile-features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_OBJCXX_STANDARD
Added in version 3.16.

Default value for *OBJCXX_STANDARD* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_OBJCXX_STANDARD_REQUIRED
Added in version 3.16.

Default value for *OBJCXX_STANDARD_REQUIRED* target property if set when a target is created.

See the *cmake−compile−features(7)* manual for information on compile features and a list of supported compilers.

CMAKE_Swift_LANGUAGE_VERSION
Added in version 3.7.

Set to the Swift language version number. If not set, the oldest legacy version known to be available in the host Xcode version is assumed:

- Swift **4.0** for Xcode 10.2 and above.
- Swift **3.0** for Xcode 8.3 and above.
- Swift **2.3** for Xcode 8.2 and below.

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 FOR CTEST

CTEST_BINARY_DIRECTORY
Added in version 3.1.

Specify the CTest **BuildDirectory** setting in a *ctest(1)* dashboard client script.

CTEST_BUILD_COMMAND
Added in version 3.1.

Specify the CTest **MakeCommand** setting in a *ctest(1)* dashboard client script.

CTEST_BUILD_NAME
Added in version 3.1.

Specify the CTest **BuildName** setting in a *ctest(1)* dashboard client script.

CTEST_BZR_COMMAND
Added in version 3.1.

Specify the CTest **BZRCommand** setting in a *ctest(1)* dashboard client script.

CTEST_BZR_UPDATE_OPTIONS
Added in version 3.1.

Specify the CTest **BZRUpdateOptions** setting in a *ctest(1)* dashboard client script.

CTEST_CHANGE_ID
Added in version 3.4.

Specify the CTest **ChangeId** setting in a *ctest(1)* dashboard client script.

This setting allows CTest to pass arbitrary information about this build up to CDash. One use of this feature is to allow CDash to post comments on your pull request if anything goes wrong with your build.

CTEST_CHECKOUT_COMMAND
Added in version 3.1.

Tell the *ctest_start()* command how to checkout or initialize the source directory in a *ctest(1)* dashboard client script.

CTEST_CONFIGURATION_TYPE
Added in version 3.1.

Specify the CTest **DefaultCTestConfigurationType** setting in a *ctest(1)* dashboard client script.

If the configuration type is set via *−C <cfg>* from the command line then this variable is populated accordingly.

CTEST_CONFIGURE_COMMAND
Added in version 3.1.

Specify the CTest **ConfigureCommand** setting in a *ctest(1)* dashboard client script.

CTEST_COVERAGE_COMMAND
Added in version 3.1.

Specify the CTest **CoverageCommand** setting in a *ctest(1)* dashboard client script.

Cobertura
Using *Cobertura* as the coverage generation within your multi−module Java project can generate a series of XML files.

The Cobertura Coverage parser expects to read the coverage data from a single XML file which contains the coverage data for all modules. Cobertura has a program with the ability to merge given **cobertura.ser** files and then another program to generate a combined XML file from the previous merged file. For command line testing, this can be done by hand prior to CTest looking for the coverage files. For script builds, set the **CTEST_COVERAGE_COMMAND** variable to point to a file which will perform these same steps, such as a **.sh** or **.bat** file.

```
set(CTEST_COVERAGE_COMMAND .../run−coverage−and−consolidate.sh)
```

where the **run−coverage−and−consolidate.sh** script is perhaps created by the *configure_file()* command and might contain the following code:

```
#!/usr/bin/env bash
CoberturaFiles="$(find "/path/to/source" −name "cobertura.ser")"
SourceDirs="$(find "/path/to/source" −name "java" −type d)"
cobertura−merge −−datafile coberturamerge.ser $CoberturaFiles
```

```
cobertura-report --datafile coberturamerge.ser --destination . \
                 --format xml $SourceDirs
```

The script uses **find** to capture the paths to all of the **cobertura.ser** files found below the project's source directory. It keeps the list of files and supplies it as an argument to the **cobertura-merge** program. The **--datafile** argument signifies where the result of the merge will be kept.

The combined **coberturamerge.ser** file is then used to generate the XML report using the **cobertura-report** program. The call to the cobertura-report program requires some named arguments.

--datafila
path to the merged **.ser** file

--destination
path to put the output files(s)

--format
file format to write output in: xml or html

The rest of the supplied arguments consist of the full paths to the **/src/main/java** directories of each module within the source tree. These directories are needed and should not be forgotten.

CTEST_COVERAGE_EXTRA_FLAGS
Added in version 3.1.

Specify the CTest **CoverageExtraFlags** setting in a *ctest(1)* dashboard client script.

CTEST_CUSTOM_COVERAGE_EXCLUDE
A list of regular expressions which will be used to exclude files by their path from coverage output by the *ctest_coverage()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_ERROR_EXCEPTION
A list of regular expressions which will be used to exclude when detecting error messages in build outputs by the *ctest_build()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_ERROR_MATCH
A list of regular expressions which will be used to detect error messages in build outputs by the *ctest_build()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_ERROR_POST_CONTEXT
The number of lines to include as context which follow an error message by the *ctest_build()* command. The default is 10.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_ERROR_PRE_CONTEXT
The number of lines to include as context which precede an error message by the *ctest_build()* command. The default is 10.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE
When saving a failing test's output, this is the maximum size, in bytes, that will be collected by the *ctest_test()* command. Defaults to 307200 (300 KiB). See *CTEST_CUSTOM_TEST_OUTPUT_TRUNCATION* for possible truncation modes.

If a test's output contains the literal string "CTEST_FULL_OUTPUT", the output will not be truncated and may exceed the maximum size.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

For controlling the output collection of passing tests, see *CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE*.

CTEST_CUSTOM_MAXIMUM_NUMBER_OF_ERRORS
The maximum number of errors in a single build step which will be detected. After this, the *ctest_test()* command will truncate the output. Defaults to 50.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_MAXIMUM_NUMBER_OF_WARNINGS
The maximum number of warnings in a single build step which will be detected. After this, the *ctest_test()* command will truncate the output. Defaults to 50.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE
When saving a passing test's output, this is the maximum size, in bytes, that will be collected by the *ctest_test()* command. Defaults to 1024 (1 KiB). See *CTEST_CUSTOM_TEST_OUTPUT_TRUNCATION* for possible truncation modes.

If a test's output contains the literal string "CTEST_FULL_OUTPUT", the output will not be truncated and may exceed the maximum size.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

For controlling the output collection of failing tests, see *CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE*.

CTEST_CUSTOM_MEMCHECK_IGNORE
A list of regular expressions to use to exclude tests during the *ctest_memcheck()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_POST_MEMCHECK
A list of commands to run at the end of the *ctest_memcheck()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_POST_TEST
A list of commands to run at the end of the *ctest_test()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_PRE_MEMCHECK
A list of commands to run at the start of the *ctest_memcheck()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_PRE_TEST
A list of commands to run at the start of the *ctest_test()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_TEST_OUTPUT_TRUNCATION
Added in version 3.24.

Set the test output truncation mode in case a maximum size is configured via the *CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE* or *CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE* variables. By default the **tail** of the output will be truncated. Other possible values are **middle** and **head**.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_TESTS_IGNORE
A list of test names to be excluded from the set of tests run by the *ctest_test()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_WARNING_EXCEPTION
A list of regular expressions which will be used to exclude when detecting warning messages in build outputs by the *ctest_build()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CUSTOM_WARNING_MATCH
A list of regular expressions which will be used to detect warning messages in build outputs by the *ctest_build()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_CVS_COMMAND
Added in version 3.1.

Specify the CTest **CVSCommand** setting in a *ctest(1)* dashboard client script.

CTEST_CVS_UPDATE_OPTIONS
Added in version 3.1.

Specify the CTest **CVSUpdateOptions** setting in a *ctest(1)* dashboard client script.

CTEST_DROP_LOCATION
Added in version 3.1.

Specify the CTest **DropLocation** setting in a *ctest(1)* dashboard client script.

CTEST_DROP_METHOD
Added in version 3.1.

Specify the CTest **DropMethod** setting in a *ctest(1)* dashboard client script.

CTEST_DROP_SITE
Added in version 3.1.

Specify the CTest **DropSite** setting in a *ctest(1)* dashboard client script.

CTEST_DROP_SITE_CDASH
Added in version 3.1.

Specify the CTest **IsCDash** setting in a *ctest(1)* dashboard client script.

CTEST_DROP_SITE_PASSWORD
Added in version 3.1.

Specify the CTest **DropSitePassword** setting in a *ctest(1)* dashboard client script.

CTEST_DROP_SITE_USER
Added in version 3.1.

Specify the CTest **DropSiteUser** setting in a *ctest(1)* dashboard client script.

CTEST_EXTRA_COVERAGE_GLOB
Added in version 3.4.

A list of regular expressions which will be used to find files which should be covered by the *ctest_coverage()* command.

It is initialized by *ctest(1)*, but may be edited in a **CTestCustom** file. See *ctest_read_custom_files()* documentation.

CTEST_EXTRA_SUBMIT_FILES
Specify files for *ctest_submit(PARTS ExtraFiles)* to submit in a *ctest(1)* dashboard client script.

CTEST_GIT_COMMAND
Added in version 3.1.

Specify the CTest **GITCommand** setting in a *ctest(1)* dashboard client script.

CTEST_GIT_INIT_SUBMODULES
Added in version 3.6.

Specify the CTest **GITInitSubmodules** setting in a *ctest(1)* dashboard client script.

CTEST_GIT_UPDATE_CUSTOM
Added in version 3.1.

Specify the CTest **GITUpdateCustom** setting in a *ctest(1)* dashboard client script.

CTEST_GIT_UPDATE_OPTIONS
Added in version 3.1.

Specify the CTest **GITUpdateOptions** setting in a *ctest(1)* dashboard client script.

CTEST_HG_COMMAND
Added in version 3.1.

Specify the CTest **HGCommand** setting in a *ctest(1)* dashboard client script.

CTEST_HG_UPDATE_OPTIONS
Added in version 3.1.

Specify the CTest **HGUpdateOptions** setting in a *ctest(1)* dashboard client script.

CTEST_LABELS_FOR_SUBPROJECTS
Added in version 3.10.

Specify the CTest **LabelsForSubprojects** setting in a *ctest(1)* dashboard client script.

CTEST_MEMORYCHECK_COMMAND
Added in version 3.1.

Specify the CTest **MemoryCheckCommand** setting in a *ctest(1)* dashboard client script.

CTEST_MEMORYCHECK_COMMAND_OPTIONS
Added in version 3.1.

Specify the CTest **MemoryCheckCommandOptions** setting in a *ctest(1)* dashboard client script.

CTEST_MEMORYCHECK_SANITIZER_OPTIONS
Added in version 3.1.

Specify the CTest **MemoryCheckSanitizerOptions** setting in a *ctest(1)* dashboard client script.

CTest prepends correct sanitizer options ***_OPTIONS** environment variable to executed command. CTests adds its own **log_path** to sanitizer options, don't provide your own **log_path**.

CTEST_MEMORYCHECK_SUPPRESSIONS_FILE
Added in version 3.1.

Specify the CTest **MemoryCheckSuppressionFile** setting in a *ctest(1)* dashboard client script.

CTEST_MEMORYCHECK_TYPE
Added in version 3.1.

Specify the CTest **MemoryCheckType** setting in a *ctest(1)* dashboard client script. Valid values are **Valgrind**, **Purify**, **BoundsChecker**, **DrMemory**, **CudaSanitizer**, **ThreadSanitizer**, **AddressSanitizer**, **LeakSanitizer**, **MemorySanitizer** and **UndefinedBehaviorSanitizer**.

CTEST_NIGHTLY_START_TIME
Added in version 3.1.

Specify the CTest **NightlyStartTime** setting in a *ctest(1)* dashboard client script.

Note that this variable must always be set for a nightly build in a dashboard script. It is needed so that nightly builds can be properly grouped together in CDash.

CTEST_NOTES_FILES
Specify files for *ctest_submit(PARTS Notes)* to submit in a *ctest(1)* dashboard client script.

CTEST_P4_CLIENT
Added in version 3.1.

Specify the CTest **P4Client** setting in a *ctest(1)* dashboard client script.

CTEST_P4_COMMAND
Added in version 3.1.

Specify the CTest **P4Command** setting in a *ctest(1)* dashboard client script.

CTEST_P4_OPTIONS
Added in version 3.1.

Specify the CTest **P4Options** setting in a *ctest(1)* dashboard client script.

CTEST_P4_UPDATE_OPTIONS
Added in version 3.1.

Specify the CTest **P4UpdateOptions** setting in a *ctest(1)* dashboard client script.

CTEST_RESOURCE_SPEC_FILE
Added in version 3.18.

Specify the CTest **ResourceSpecFile** setting in a *ctest(1)* dashboard client script.

This can also be used to specify the resource spec file from a CMake build. If no **RESOURCE_SPEC_FILE** is passed to *ctest_test()*, and **CTEST_RESOURCE_SPEC_FILE** is not specified in the dashboard script, the value of this variable from the build is used.

CTEST_RUN_CURRENT_SCRIPT
Removed. This variable once supported an undocumented feature that has since been removed.

CTEST_SCRIPT_DIRECTORY
The directory containing the top−level CTest script. The concept is similar to *CMAKE_SOURCE_DIR*.

CTEST_SITE
Added in version 3.1.

Specify the CTest **Site** setting in a *ctest(1)* dashboard client script.

CTEST_SOURCE_DIRECTORY
Added in version 3.1.

Specify the CTest **SourceDirectory** setting in a *ctest(1)* dashboard client script.

CTEST_SUBMIT_INACTIVITY_TIMEOUT
Added in version 3.23.

Specify the CTest **SubmitInactivityTimeout** setting in a *ctest(1)* dashboard client script.

CTEST_SUBMIT_URL
Added in version 3.14.

Specify the CTest **SubmitURL** setting in a *ctest(1)* dashboard client script.

CTEST_SVN_COMMAND
Added in version 3.1.

Specify the CTest **SVNCommand** setting in a *ctest(1)* dashboard client script.

CTEST_SVN_OPTIONS
Added in version 3.1.

Specify the CTest **SVNOptions** setting in a *ctest(1)* dashboard client script.

CTEST_SVN_UPDATE_OPTIONS
Added in version 3.1.

Specify the CTest **SVNUpdateOptions** setting in a *ctest(1)* dashboard client script.

CTEST_TEST_LOAD
Added in version 3.4.

Specify the **TestLoad** setting in the *CTest Test Step* of a *ctest(1)* dashboard client script. This sets the default value for the **TEST_LOAD** option of the *ctest_test()* command.

CTEST_TEST_TIMEOUT
Added in version 3.1.

Specify the CTest **TimeOut** setting in a *ctest(1)* dashboard client script.

CTEST_TLS_VERIFY
Added in version 3.30.

Specify the CTest **TLSVerify** setting in a *ctest(1) Dashboard Client* script or in project **CMakeLists.txt**

code before including the *CTest* module. The value is a boolean indicating whether to verify the server certificate when submitting to a dashboard via **https://** URLs.

If **CTEST_TLS_VERIFY** is not set, the *CMAKE_TLS_VERIFY* variable or *CMAKE_TLS_VERIFY* environment variable is used instead. If neither is set, the default is *on*.

Changed in version 3.31: The default is on. Previously, the default was off. Users may set the *CMAKE_TLS_VERIFY* environment variable to **0** to restore the old default.

CTEST_TLS_VERSION
Added in version 3.30.

Specify the CTest **TLSVersion** setting in a *ctest(1) Dashboard Client* script or in project **CMakeLists.txt** code before including the *CTest* module. The value is a minimum TLS version allowed when submitting to a dashboard via **https://** URLs.

The value may be one of:

- **1.0**
- **1.1**
- **1.2**
- **1.3**

If **CTEST_TLS_VERSION** is not set, the *CMAKE_TLS_VERSION* variable or *CMAKE_TLS_VERSION* environment variable is used instead.

CTEST_UPDATE_COMMAND
Added in version 3.1.

Specify the CTest **UpdateCommand** setting in a *ctest(1)* dashboard client script.

CTEST_UPDATE_OPTIONS
Added in version 3.1.

Specify the CTest **UpdateOptions** setting in a *ctest(1)* dashboard client script.

CTEST_UPDATE_VERSION_ONLY
Added in version 3.1.

Specify the CTest *UpdateVersionOnly* setting in a *ctest(1)* dashboard client script.

CTEST_UPDATE_VERSION_OVERRIDE
Added in version 3.15.

Specify the CTest *UpdateVersionOverride* setting in a *ctest(1)* dashboard client script.

CTEST_USE_LAUNCHERS
Added in version 3.1.

Specify the CTest **UseLaunchers** setting in a *ctest(1)* dashboard client script.

VARIABLES FOR CPACK

CPACK_ABSOLUTE_DESTINATION_FILES

List of files which have been installed using an **ABSOLUTE DESTINATION** path.

This variable is a Read−Only variable which is set internally by CPack during installation and before packaging using *CMAKE_ABSOLUTE_DESTINATION_FILES* defined in **cmake_install.cmake** scripts. The value can be used within CPack project configuration file and/or **CPack<GEN>.cmake** file of **<GEN>** generator.

CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY

Boolean toggle to include/exclude top level directory (component case).

Similar usage as *CPACK_INCLUDE_TOPLEVEL_DIRECTORY* but for the component case. See *CPACK_INCLUDE_TOPLEVEL_DIRECTORY* documentation for the detail.

CPACK_CUSTOM_INSTALL_VARIABLES

Added in version 3.21.

CPack variables (set via e.g. *cpack −D*, **CPackConfig.cmake** or *CPACK_PROJECT_CONFIG_FILE* scripts) are not directly visible in installation scripts. Instead, one can pass a list of **varName=value** pairs in the **CPACK_CUSTOM_INSTALL_VARIABLES** variable. At install time, each list item will result in a variable of the specified name (**varName**) being set to the given **value**. The= can be omitted for an empty **value**.

CPACK_CUSTOM_INSTALL_VARIABLES allows the packaging installation to be influenced by the user or driving script at CPack runtime without having to regenerate the install scripts.

Example

```
install(FILES large.txt DESTINATION data)

install(CODE [[
  if(ENABLE_COMPRESSION)
    # "run-compressor" is a fictional tool that produces
    # large.txt.xz from large.txt and then removes the input file
    execute_process(COMMAND run-compressor $ENV{DESTDIR}${CMAKE_INSTALL_PREFIX}/l
  endif()
]])
```

With the above example snippet, *cpack* will by default run the installation script with **ENABLE_COMPRESSION** unset, resulting in a package containing the uncompressed **large.txt**. This can be overridden when invoking *cpack* like so:

```
cpack -D "CPACK_CUSTOM_INSTALL_VARIABLES=ENABLE_COMPRESSION=TRUE"
```

The installation script will then run with **ENABLE_COMPRESSION** set to **TRUE**, resulting in a package containing the compressed **large.txt.xz** instead.

CPACK_ERROR_ON_ABSOLUTE_INSTALL_DESTINATION

Ask CPack 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. Some CPack generators, like **NSIS**, enforce this internally. This variable triggers the definition of *CMAKE_ERROR_ON_ABSOLUTE_INSTALL_DESTINATION* when CPack runs.

CPACK_INCLUDE_TOPLEVEL_DIRECTORY
Boolean toggle to include/exclude top level directory.

When preparing a package CPack installs the item under the so−called top level directory. The purpose of is to include (set to **1** or **ON** or **TRUE**) the top level directory in the package or not (set to **0** or **OFF** or **FALSE**).

Each CPack generator has a built−in default value for this variable. E.g. Archive generators (ZIP, TGZ, ...) includes the top level whereas RPM or DEB don't. The user may override the default value by setting this variable.

There is a similar variable *CPACK_COMPONENT_INCLUDE_TOPLEVEL_DIRECTORY* which may be used to override the behavior for the component packaging case which may have different default value for historical (now backward compatibility) reason.

CPACK_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS
Added in version 3.11.

Default permissions for implicitly created directories during packaging.

This variable serves the same purpose during packaging as the *CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS* variable serves during installation (e.g. **make install**).

If **include(CPack)** is used then by default this variable is set to the content of *CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS*.

CPACK_PACKAGING_INSTALL_PREFIX
The prefix used in the built package.

Each CPack generator has a default value (like **/usr**). This default value may be overwritten from the **CMakeLists.txt** or the *cpack(1)* command line by setting an alternative value. Example:

```
set(CPACK_PACKAGING_INSTALL_PREFIX "/opt")
```

This is not the same purpose as *CMAKE_INSTALL_PREFIX* which is used when installing from the build tree without building a package.

CPACK_SET_DESTDIR
Boolean toggle to make CPack use **DESTDIR** mechanism when packaging.

DESTDIR means DESTination DIRectory. It is commonly used by makefile users in order to install software at non−default location. It is a basic relocation mechanism that should not be used on Windows (see *CMAKE_INSTALL_PREFIX* documentation). 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**. When preparing a package, CPack first installs the items to be packaged in a local (to the build tree) directory by using the same **DESTDIR** mechanism. Nevertheless, if **CPACK_SET_DESTDIR** is set then CPack will set **DESTDIR** before doing the local install. The most noticeable difference is that without **CPACK_SET_DESTDIR**, CPack uses *CPACK_PACKAGING_INSTALL_PREFIX* as a prefix whereas with **CPACK_SET_DESTDIR** set, CPack will use *CMAKE_INSTALL_PREFIX* as a prefix.

Manually setting **CPACK_SET_DESTDIR** may help (or simply be necessary) if some install rules uses absolute **DESTINATION** (see CMake *install()* command). However, starting with CPack/CMake 2.8.3 RPM and DEB installers tries to handle **DESTDIR** automatically so that it is seldom necessary for the user to set it.

CPACK_WARN_ON_ABSOLUTE_INSTALL_DESTINATION
Ask CPack to warn each time a file with absolute **INSTALL DESTINATION** is encountered.

This variable triggers the definition of *CMAKE_WARN_ON_ABSOLUTE_INSTALL_DESTINATION* when CPack runs **cmake_install.cmake** scripts.

VARIABLE EXPANSION OPERATORS

CACHE
Added in version 3.13.

Operator to read cache variables.

Use the syntax **$CACHE{VAR}** to read cache entry **VAR**. See the*cmak e−language(7) variables* documentation for more complete documentation of the interaction of normal variables and cache entries.

When evaluating *Variable References* of the form **${VAR}**, CMake first searches for a normal variable with that name, and if not found CMake will search for a cache entry with that name. The **$CACHE{VAR}** syntax can be used to do direct cache lookup and ignore any existing normal variable.

See the *set()* and *unset()* commands to see how to write or remove cache variables.

ENV
Operator to read environment variables.

Use the syntax **$ENV{VAR}** to read environment variable **VAR**.

To test whether an environment variable is defined, use the signature **if(DEFINED ENV{<name>})** of the *if()* command.

NOTE:
Environment variable names containing special characters like parentheses may need to be escaped. (Policy *CMP0053* must also be enabled.) For example, to get the value of the Windows environment variable **ProgramFiles(x86)**, use:

```
set(ProgramFiles_x86 "$ENV{ProgramFiles\(x86\)}")
```

For general information on environment variables, see the *Environment Variables* section in the *cmake−language(7)* manual.

INTERNAL VARIABLES

CMake has many internal variables. Most of them are undocumented. Some of them, however, were at some point described as normal variables, and therefore may be encountered in legacy code. They are subject to change, and not recommended for use in project code.

CMAKE_HOME_DIRECTORY
Path to top of source tree. Same as *CMAKE_SOURCE_DIR*.

This is an internal cache entry used to locate the source directory when loading a **CMakeCache.txt** from a build tree. It should not be used in project code. The variable *CMAKE_SOURCE_DIR* has the same value and should be preferred.

CMAKE_INTERNAL_PLATFORM_ABI
An internal variable subject to change.

This is used in determining the compiler ABI and is subject to change.

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_ARCHITECTURE_ID
Added in version 3.10.

An internal variable subject to change.

This is used to identify the variant of a compiler based on its target architecture. For some compilers this is needed to determine the correct usage.

CMAKE_<LANG>_COMPILER_VERSION_INTERNAL
Added in version 3.10.

An internal variable subject to change.

This is used to identify the variant of a compiler based on an internal version number. For some compilers this is needed to determine the correct usage.

CMAKE_<LANG>_LINKER_PREFERENCE
An internal variable subject to change.

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_LANGU AGE* 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_PR OPAGATES* variable.

CMAKE_<LANG>_LINKER_PREFERENCE_PROPAGATES
An internal variable subject to change.

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>_PLATFORM_ID
An internal variable subject to change.

This is used in determining the platform and is subject to change.

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_VS_INTEL_Fortran_PROJECT_VERSION
When generating for *Visual Studio 14 2015* or greater with the Intel Fortran plugin installed, this specifies the **.vfproj** project file format version. This is intended for internal use by CMake and should not be used by project code.

DEPRECATED VARIABLES THAT PROVIDE INFORMATION

CMAKE_EXTRA_GENERATOR
Deprecated since version 3.27: Support for *Extra Generators* is deprecated and will be removed from a future version of CMake. IDEs may use the*cmak e−file−api(7)* to view CMake−generated project build trees.

The extra generator used to build the project. See *cmake−generators(7)*.

When using the Eclipse, CodeBlocks, CodeLite, Kate or Sublime 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**).

DEPRECATED VARIABLES THAT CHANGE BEHAVIOR

CMAKE_AUTOMOC_RELAXED_MODE
Deprecated since version 3.15.

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**TR UE**, it accepts more input and tries to find the correct input file for **moc** even if it differs from the documented behavior. 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
Removed. See policy *CMP0001*.

CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY
Added in version 3.1.

Deprecated since version 3.16: Use the *CMAKE_FIND_USE_PACKAGE_REGISTRY* variable instead.

By default this variable is not set. If neither *CMAKE_FIND_USE_PACKAGE_REGISTRY* nor **CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY** is set, then *find_package()* will use the *User Package Registry* unless the **NO_CMAKE_PACKAGE_REGISTRY** option is provided.

CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY is ignored if *CMAKE_FIND_USE_PACKAGE_REGISTRY* is set.

In some cases, for example to locate only system wide installations, it is not desirable to use the *User Package Registry* when searching for packages. If the **CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY** variable is **TRUE**, all the *find_package()* commands will skip the *User Package Registry* as if they were called with the **NO_CMAKE_PACKAGE_REGISTRY** argument.

See also *Disabling the Package Registry*.

CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY
Added in version 3.1.

Deprecated since version 3.16: Use the *CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY* variable instead.

By default this variable is not set. If neither *CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY* nor **CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY** is set, then *find_package()* will use the *System Package Registry* unless the **NO_CMAKE_SYSTEM_PACKAGE_REGISTRY** option is provided.

CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY is ignored if *CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY* is set.

In some cases, it is not desirable to use the *System Package Registry* when searching for packages. If the **CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY** variable is **TRUE**, all the *find_package()* commands will skip the *System Package Registry* as if they were called with the **NO_CMAKE_SYSTEM_PACKAGE_REGISTRY** argument.

See also *Disabling the Package Registry*.

DEPRECATED VARIABLES THAT DESCRIBE THE SYSTEM

MSVC10
Discouraged. Use the *MSVC_VERSION* variable instead.

True when using the Microsoft Visual Studio **v100** toolset (**cl** version 16) or another compiler that simulates it.

MSVC11
Discouraged. Use the *MSVC_VERSION* variable instead.

True when using the Microsoft Visual Studio **v110** toolset (**cl** version 17) or another compiler that simulates it.

MSVC12
Discouraged. Use the *MSVC_VERSION* variable instead.

True when using the Microsoft Visual Studio **v120** toolset (**cl** version 18) or another compiler that simulates it.

MSVC14
Added in version 3.1.

Discouraged. Use the*MSVC_VERSION* v ariable instead.

True when using the Microsoft Visual Studio **v140** or **v141** toolset (**cl** version 19) or another compiler that simulates it.

MSVC60
Discouraged. Use the *MSVC_VERSION* variable instead.

True when using Microsoft Visual C++ 6.0.

Set to **true** when the compiler is version 6.0 of Microsoft Visual C++.

MSVC70
Discouraged. Use the *MSVC_VERSION* variable instead.

True when using Microsoft Visual C++ 7.0.

Set to **true** when the compiler is version 7.0 of Microsoft Visual C++.

MSVC71
Discouraged. Use the *MSVC_VERSION* variable instead.

True when using Microsoft Visual C++ 7.1.

Set to **true** when the compiler is version 7.1 of Microsoft Visual C++.

MSVC80
Discouraged. Use the *MSVC_VERSION* variable instead.

True when using the Microsoft Visual Studio **v80** toolset (**cl** version 14) or another compiler that simulates it.

MSVC90
Discouraged. Use the *MSVC_VERSION* variable instead.

True when using the Microsoft Visual Studio **v90** toolset (**cl** version 15) or another compiler that simulates it.

DEPRECATED VARIABLES THAT CONTROL THE BUILD

CMAKE_IOS_INSTALL_COMBINED
Added in version 3.5.

Deprecated since version 3.28: This is deprecated because *IOS_INSTALL_COMBINED* is deprecated.

Default value for *IOS_INSTALL_COMBINED* of targets.

This variable is used to initialize the *IOS_INSTALL_COMBINED* property on all the targets. See that target property for additional information.

CMAKE_<LANG>_USING_LINKER_MODE
Added in version 3.29.

This controls how the value of the *CMAKE_<LANG>_USING_LINKER_<TYPE>* variable should be interpreted. The supported linker mode values are:

FLAG *CMAKE_<LANG>_USING_LINKER_<TYPE>* holds a *semicolon−separated list* of flags to be passed to the compiler frontend. This is also the default behavior if **CMAKE_<LANG>_USING_LINKER_MODE** is not set.

TOOL *CMAKE_<LANG>_USING_LINKER_<TYPE>* holds the path to the linker tool.

WARNING:
The variable must be set accordingly to how CMake manage the link step:

- value **TOOL** is expected and required when the linker is used directly for the link step.
- value **FLAGS** is expected or the variable not set when the compiler is used as driver for the link step.

Deprecated since version 4.0.

This variable is no longer used. The type of information stored in the *CMAKE_<LANG>_USING_LINKER_<TYPE>* variable is determined by the *CMAKE_<LANG>_LINK_MODE* variable.

CMAKE_USE_RELATIVE_PATHS
This variable has no effect. The partially implemented effect it had in previous releases was removed in CMake 3.4.

DEPRECATED VARIABLES FOR LANGUAGES

CMAKE_COMPILER_IS_GNUCC
True if the **C** compiler is GNU.

This variable is deprecated. Use *CMAKE_C_COMPILER_ID* instead.

CMAKE_COMPILER_IS_GNUCXX
True if the C++ (**CXX**) compiler is GNU.

This variable is deprecated. Use *CMAKE_CXX_COMPILER_ID* instead.

CMAKE_COMPILER_IS_GNUG77
True if the **Fortran** compiler is GNU.

This variable is deprecated. Use *CMAKE_Fortran_COMPILER_ID* instead.

DEPRECATED VARIABLES FOR CTEST

CTEST_CURL_OPTIONS
Deprecated since version 3.30: Use the *CTEST_TLS_VERIFY* variable instead.

Added in version 3.1.

Specify the CTest **CurlOptions** setting in a *ctest(1)* dashboard client script.

CTEST_CVS_CHECKOUT
Added in version 3.1.

Deprecated. Use*CTEST_CHECK OUT_COMMAND* instead.

CTEST_SCP_COMMAND
Added in version 3.1.

Legacy option. Not used.

CTEST_TRIGGER_SITE
Added in version 3.1.

Legacy option. Not used.

COPYRIGHT