NAME

cmake-commands − CMake Language Command Reference

SCRIPTING COMMANDS

These commands are always available.

block

Added in version 3.25.

Evaluate a group of commands with a dedicated variable and/or policy scope.

```
block([SCOPE_FOR [POLICIES] [VARIABLES]] [PROPAGATE <var-name>...])
  <commands>
endblock()
```

All commands between **block()** and the matching *endblock()* are recorded without being invoked. Once the *endblock()* is evaluated, the recorded list of commands is invoked inside the requested scopes, then the scopes created by the **block()** command are removed.

SCOPE_FOR

Specify which scopes must be created.

POLICIES

Create a new policy scope. This is equivalent to *cmake_policy(PUSH)* with an automatic *cmake_policy(POP)* when leaving the block scope.

VARIABLES

Create a new variable scope.

If **SCOPE_FOR** is not specified, this is equivalent to:

```
block(SCOPE_FOR VARIABLES POLICIES)
```

PROPAGATE

When a variable scope is created by the *block()* command, this option sets or unsets the specified variables in the parent scope. This is equivalent to *set(PARENT_SCOPE)* or *unset(PARENT_SCOPE)* commands.

```
set(var1 "INIT1")
set(var2 "INIT2")

block(PROPAGATE var1 var2)
  set(var1 "VALUE1")
  unset(var2)
endblock()

# Now var1 holds VALUE1, and var2 is unset
```

This option is only allowed when a variable scope is created. An error will be raised in the other cases.

When the **block()** is inside a *foreach()* or *while()* command, the *break()* and *continue()* commands can be used inside the block.

```
while(TRUE)
  block()
    ...
```

```
    # the break() command will terminate the while() command
    break()
  endblock()
endwhile()
```

See Also

- *endblock()*
- *return()*
- *cmake_policy()*

break

Break from an enclosing foreach or while loop.

```
break()
```

Breaks from an enclosing *foreach()* or *while()* loop.

See also the *continue()* command.

cmake_host_system_information

Query various host system information.

Synopsis

```
Query host system specific information
  cmake_host_system_information(RESULT <variable> QUERY <key> ...)

Query Windows registry
  cmake_host_system_information(RESULT <variable> QUERY WINDOWS_REGISTRY <key> ..
```

Query host system specific information

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

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

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

NUMBER_OF_LOGICAL_CORES
Number of logical cores

NUMBER_OF_PHYSICAL_CORES
Number of physical cores

HOSTNAME
Hostname

FQDN Fully qualified domain name

TOTAL_VIRTUAL_MEMORY
Total virtual memory in MiB [1]

AVAILABLE_VIRTUAL_MEMORY
Available virtual memory in MiB [1]

TOTAL_PHYSICAL_MEMORY
Total physical memory in MiB [1]

AVAILABLE_PHYSICAL_MEMORY
Available physical memory in MiB [1]

IS_64BIT
Added in version 3.10.

One if processor is 64Bit

HAS_FPU
Added in version 3.10.

One if processor has floating point unit

HAS_MMX
Added in version 3.10.

One if processor supports MMX instructions

HAS_MMX_PLUS
Added in version 3.10.

One if processor supports Ext. MMX instructions

HAS_SSE
Added in version 3.10.

One if processor supports SSE instructions

HAS_SSE2
Added in version 3.10.

One if processor supports SSE2 instructions

HAS_SSE_FP
Added in version 3.10.

One if processor supports SSE FP instructions

HAS_SSE_MMX
Added in version 3.10.

One if processor supports SSE MMX instructions

HAS_AMD_3DNOW
Added in version 3.10.

One if processor supports 3DNow instructions

HAS_AMD_3DNOW_PLUS
Added in version 3.10.

One if processor supports 3DNow+ instructions

HAS_IA64
Added in version 3.10.

One if IA64 processor emulating x86

HAS_SERIAL_NUMBER
Added in version 3.10.

One if processor has serial number

PROCESSOR_SERIAL_NUMBER
Added in version 3.10.

Processor serial number

PROCESSOR_NAME
Added in version 3.10.

Human readable processor name

PROCESSOR_DESCRIPTION
Added in version 3.10.

Human readable full processor description

OS_NAME
Added in version 3.10.

The host operating system name:

- On UNIX platforms, this is **uname −s**.
- On Apple platforms, this is **sw_vers −productName**.
- On Windows, this is **Windows**.

See also *CMAKE_HOST_SYSTEM_NAME*.

OS_RELEASE
Added in version 3.10.

The OS sub−type e.g. on Windows **Professional**

OS_VERSION
Added in version 3.10.

The OS build ID

OS_PLATFORM
Added in version 3.10.

See *CMAKE_HOST_SYSTEM_PROCESSOR*

MSYSTEM_PREFIX
Added in version 3.28.

Available only on Windows hosts. In a MSYS or MinGW development environment that sets the **MSYSTEM** environment variable, this is its installation prefix. Otherwise, this is the empty string.

DISTRIB_INFO
Added in version 3.22.

Read **/etc/os−release** file and define the given **<variable>** into a list of read variables

DISTRIB_<name>
Added in version 3.22.

Get the **<name>** variable (see *man 5 os−release*) if it exists in the **/etc/os−release** file

Example:

```
cmake_host_system_information(RESULT PRETTY_NAME QUERY DISTRIB_PRETTY_NAME)
message(STATUS "${PRETTY_NAME}")

cmake_host_system_information(RESULT DISTRO QUERY DISTRIB_INFO)

foreach(VAR IN LISTS DISTRO)
  message(STATUS "${VAR}=`${${VAR}}`")
endforeach()
```

Output:

```
-- Ubuntu 20.04.2 LTS
-- DISTRO_BUG_REPORT_URL=`https://bugs.launchpad.net/ubuntu/`
-- DISTRO_HOME_URL=`https://www.ubuntu.com/`
-- DISTRO_ID=`ubuntu`
-- DISTRO_ID_LIKE=`debian`
-- DISTRO_NAME=`Ubuntu`
-- DISTRO_PRETTY_NAME=`Ubuntu 20.04.2 LTS`
-- DISTRO_PRIVACY_POLICY_URL=`https://www.ubuntu.com/legal/terms-and-polici
-- DISTRO_SUPPORT_URL=`https://help.ubuntu.com/`
-- DISTRO_UBUNTU_CODENAME=`focal`
-- DISTRO_VERSION=`20.04.2 LTS (Focal Fossa)`
-- DISTRO_VERSION_CODENAME=`focal`
-- DISTRO_VERSION_ID=`20.04`
```

If **/etc/os−release** file is not found, the command tries to gather OS identification via fallback scripts. The fallback script can use *various distribution−specific files* to collect OS identification data and map it into *man 5 os−release* variables.

Fallback Interface Variables

CMAKE_GET_OS_RELEASE_FALLBACK_SCRIPTS
In addition to the scripts shipped with CMake, a user may append full paths to his script(s) to the this list. The script filename has the following format: **NNN−<name>.cmake**, where **NNN** is three digits used to apply collected scripts in a specific order.

CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_<varname>
Variables collected by the user provided fallback script ought to be assigned to CMake variables using this naming convention. Example, the**ID** v ariable from the manual becomes **CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_ID**.

CMAKE_GET_OS_RELEASE_FALLBACK_RESULT
The fallback script ought to store names of all assigned **CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_<varname>** variables in this list.

Example:

```
# Try to detect some old distribution
# See also
# - http://linuxmafia.com/faq/Admin/release-files.html
#
if(NOT EXISTS "${CMAKE_SYSROOT}/etc/foobar-release")
  return()
endif()
# Get the first string only
file(
    STRINGS "${CMAKE_SYSROOT}/etc/foobar-release" CMAKE_GET_OS_RELEASE_FALLBACK_C
    LIMIT_COUNT 1
  )
#
# Example:
#
#   Foobar distribution release 1.2.3 (server)
#
if(CMAKE_GET_OS_RELEASE_FALLBACK_CONTENT MATCHES "Foobar distribution release ([0
  set(CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_NAME Foobar)
  set(CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_PRETTY_NAME "${CMAKE_GET_OS_RELEASE_FA
  set(CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_ID foobar)
  set(CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_VERSION ${CMAKE_MATCH_1})
  set(CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_VERSION_ID ${CMAKE_MATCH_1})
  list(
      APPEND CMAKE_GET_OS_RELEASE_FALLBACK_RESULT
      CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_NAME
      CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_PRETTY_NAME
      CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_ID
      CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_VERSION
      CMAKE_GET_OS_RELEASE_FALLBACK_RESULT_VERSION_ID
    )
endif()
unset(CMAKE_GET_OS_RELEASE_FALLBACK_CONTENT)
```

FOOTNOTES

[1] One MiB (mebibyte) is equal to 1024x1024 bytes.

Query Windows registry
Added in version 3.24.

```
cmake_host_system_information(RESULT <variable>
                              QUERY WINDOWS_REGISTRY <key> [VALUE_NAMES|SUBKEYS|V
                              [VIEW (64|32|64_32|32_64|HOST|TARGET|BOTH)]
                              [SEPARATOR <separator>]
```

```
                                [ERROR_VARIABLE <result>])
```

Performs query operations on local computer registry subkey. Returns a list of subkeys or value names that are located under the specified subkey in the registry or the data of the specified value name. The result of the queried entity is stored in **<variable>**.

NOTE:
Querying registry for any other platforms than **Windows**, including **CYGWIN**, will always returns an empty string and sets an error message in the variable specified with sub−option **ERROR_VARIABLE**.

<key> specify the full path of a subkey on the local computer. The **<key>** must include a valid root key. Valid root keys for the local computer are:

- **HKLM** or **HKEY_LOCAL_MACHINE**
- **HKCU** or **HKEY_CURRENT_USER**
- **HKCR** or **HKEY_CLASSES_ROOT**
- **HKU** or **HKEY_USERS**
- **HKCC** or **HKEY_CURRENT_CONFIG**

And, optionally, the path to a subkey under the specified root key. The path separator can be the slash or the backslash. **<key>** is not case sensitive. For example:

```
cmake_host_system_information(RESULT result QUERY WINDOWS_REGISTRY "HKLM")
cmake_host_system_information(RESULT result QUERY WINDOWS_REGISTRY "HKLM/SOFTWARE
cmake_host_system_information(RESULT result QUERY WINDOWS_REGISTRY "HKCU\\SOFTWAR
```

VALUE_NAMES
Request the list of value names defined under **<key>**. If a default value is defined, it will be identified with the special name **(default)**.

SUBKEYS
Request the list of subkeys defined under **<key>**.

VALUE <name>
Request the data stored in value named **<name>**. If **VALUE** is not specified or argument is the special name **(default)**, the content of the default value, if any, will be returned.

```
# query default value for HKLM/SOFTWARE/Kitware key
cmake_host_system_information(RESULT result
                              QUERY WINDOWS_REGISTRY "HKLM/SOFTWARE/Kitware"

# query default value for HKLM/SOFTWARE/Kitware key using special value name
cmake_host_system_information(RESULT result
                              QUERY WINDOWS_REGISTRY "HKLM/SOFTWARE/Kitware"
                              VALUE "(default)")
```

Supported types are:

- **REG_SZ**.
- **REG_EXPAND_SZ**. The returned data is expanded.
- **REG_MULTI_SZ**. The returned is expressed as a CMake list. See also **SEPARATOR** sub−option.

- **REG_DWORD**.
- **REG_QWORD**.

For all other types, an empty string is returned.

VIEW Specify which registry views must be queried. When not specified, **BOTH** view is used.

64 Query the 64bit registry. On **32bit Windows**, returns always an empty string.

32 Query the 32bit registry.

64_32 For **VALUE** sub−option or default value, query the registry using view **64**, and if the request failed, query the registry using view **32**. For **VALUE_NAMES** and **SUBKEYS** sub−options, query both views (**64** and **32**) and merge the results (sorted and duplicates removed).

32_64 For **VALUE** sub−option or default value, query the registry using view **32**, and if the request failed, query the registry using view **64**. For **VALUE_NAMES** and **SUBKEYS** sub−options, query both views (**32** and **64**) and merge the results (sorted and duplicates removed).

HOST Query the registry matching the architecture of the host: **64** on **64bit Windows** and **32** on **32bit Windows**.

TARGET
Query the registry matching the architecture specified by *CMAKE_SIZEOF_VOID_P* variable. If not defined, fallback to **HOST** view.

BOTH Query both views (**32** and **64**). The order depends of the following rules: If *CMAKE_SIZEOF_VOID_P* variable is defined. Use the following view depending of the content of this variable:

- **8**: **64_32**
- **4**: **32_64**

If *CMAKE_SIZEOF_VOID_P* variable is not defined, rely on architecture of the host:

- **64bit**: **64_32**
- **32bit**: **32**

SEPARATOR
Specify the separator character for **REG_MULTI_SZ** type. When not specified, the character **\0** is used.

ERROR_VARIABLE <result>
Returns any error raised during query operation. In case of success, the variable holds an empty string.

cmake_language
Added in version 3.18.

Call meta−operations on CMake commands.

Synopsis

```
cmake_language(CALL <command> [<arg>...])
cmake_language(EVAL CODE <code>...)
cmake_language(DEFER <options>... CALL <command> [<arg>...])
cmake_language(SET_DEPENDENCY_PROVIDER <command> SUPPORTED_METHODS <methods>...)
cmake_language(GET_MESSAGE_LOG_LEVEL <out-var>)
cmake_language(EXIT <exit-code>)
```

Introduction

This command will call meta−operations on built−in CMake commands or those created via the *macro()* or *function()* commands.

cmake_language does not introduce a new variable or policy scope.

Calling Commands

cmake_language(CALL <command> [<arg>...])

Calls the named **<command>** with the given arguments (if any). For example, the code:

```
set(message_command "message")
cmake_language(CALL ${message_command} STATUS "Hello World!")
```

is equivalent to

```
message(STATUS "Hello World!")
```

NOTE:

To ensure consistency of the code, the following commands are not allowed:

- **if** / **elseif** / **else** / **endif**
- **block** / **endblock**
- **while** / **endwhile**
- **foreach** / **endforeach**
- **function** / **endfunction**
- **macro** / **endmacro**

Evaluating Code

cmake_language(EVAL CODE <code>...)

Evaluates the **<code>...** as CMake code.

For example, the code:

```
set(A TRUE)
set(B TRUE)
set(C TRUE)
set(condition "(A AND B) OR C")

cmake_language(EVAL CODE "
  if (${condition})
    message(STATUS TRUE)
  else()
    message(STATUS FALSE)
  endif()"
)
```

is equivalent to

```
set(A TRUE)
set(B TRUE)
set(C TRUE)
set(condition "(A AND B) OR C")

file(WRITE ${CMAKE_CURRENT_BINARY_DIR}/eval.cmake "
```

```
  if (${condition})
    message(STATUS TRUE)
  else()
    message(STATUS FALSE)
  endif()"
)

include(${CMAKE_CURRENT_BINARY_DIR}/eval.cmake)
```

Deferring Calls

Added in version 3.19.

cmake_language(DEFER <options>... CALL <command> [<arg>...])

Schedules a call to the named **<command>** with the given arguments (if any) to occur at a later time. By default, deferred calls are executed as if written at the end of the current directory's **CMakeLists.txt** file, except that they run even after a *return()* call. Variable references in arguments are evaluated at the time the deferred call is executed.

The options are:

DIRECTORY <dir>

Schedule the call for the end of the given directory instead of the current directory. The **<dir>** may reference either a source directory or its corresponding binary directory. Relative paths are treated as relative to the current source directory.

The given directory must be known to CMake, being either the top−level directory or one added by *add_subdirectory()*. Furthermore, the given directory must not yet be finished processing. This means it can be the current directory or one of its ancestors.

ID <id>

Specify an identification for the deferred call. The **<id>** may not be empty and may not begin with a capital letter **A−Z**. The**<id>** may be gin with an underscore (**_**) only if it was generated automatically by an earlier call that used **ID_VAR** to get the id.

ID_VAR <var>

Specify a variable in which to store the identification for the deferred call. If **ID <id>** is not given, a new identification will be generated and the generated id will start with an underscore (**_**).

The currently scheduled list of deferred calls may be retrieved:

```
cmake_language(DEFER [DIRECTORY <dir>] GET_CALL_IDS <var>)
```

This will store in **<var>** a *semicolon−separated list* of deferred call ids. The ids are for the directory scope in which the calls have been deferred to (i.e. where they will be executed), which can be different to the scope in which they were created. The **DIRECTORY** option can be used to specify the scope for which to retrieve the call ids. If that option is not given, the call ids for the current directory scope will be returned.

Details of a specific call may be retrieved from its id:

```
cmake_language(DEFER [DIRECTORY <dir>] GET_CALL <id> <var>)
```

This will store in **<var>** a *semicolon−separated list* in which the first element is the name of the command to be called, and the remaining elements are its unevaluated arguments (any contained **;** characters are included literally and cannot be distinguished from multiple arguments). If multiple

calls are scheduled with the same id, this retrieves the first one. If no call is scheduled with the given id in the specified **DIRECTORY** scope (or the current directory scope if no **DIRECTORY** option is given), this stores an empty string in the variable.

Deferred calls may be canceled by their id:

```
cmake_language(DEFER [DIRECTORY <dir>] CANCEL_CALL <id>...)
```

This cancels all deferred calls matching any of the given ids in the specified **DIRECTORY** scope (or the current directory scope if no **DIRECTORY** option is given). Unknown ids are silently ignored.

Deferred Call Examples

For example, the code:

```
cmake_language(DEFER CALL message "${deferred_message}")
cmake_language(DEFER ID_VAR id CALL message "Canceled Message")
cmake_language(DEFER CANCEL_CALL ${id})
message("Immediate Message")
set(deferred_message "Deferred Message")
```

prints:

```
Immediate Message
Deferred Message
```

The **Canceled Message** is never printed because its command is canceled. The **deferred_message** variable reference is not evaluated until the call site, so it can be set after the deferred call is scheduled.

In order to evaluate variable references immediately when scheduling a deferred call, wrap it using **cmake_language(EVAL)**. However, note that arguments will be re−evaluated in the deferred call, though that can be avoided by using bracket arguments. For example:

```
set(deferred_message "Deferred Message 1")
set(re_evaluated [[${deferred_message}]])
cmake_language(EVAL CODE "
  cmake_language(DEFER CALL message [[${deferred_message}]])
  cmake_language(DEFER CALL message \"${re_evaluated}\")
")
message("Immediate Message")
set(deferred_message "Deferred Message 2")
```

also prints:

```
Immediate Message
Deferred Message 1
Deferred Message 2
```

Dependency Providers

Added in version 3.24.

NOTE:

A high−level introduction to this feature can be found in the *Using Dependencies Guide*.

cmake_language(SET_DEPENDENCY_PROVIDER <command> SUPPORTED_METHODS <methods>...)

When a call is made to *find_package()* or *FetchContent_MakeAvailable()*, the call may be forwarded to a dependency provider which then has the opportunity to fulfill the request. If the request is for one of the **<methods>** specified when the provider was set, CMake calls the provider's **<command>** with a set of method−specific arguments. If the provider does not fulfill the request, or if the provider doesn't support the request's method, or no provider is set, the built−in *find_package()* or *FetchContent_MakeAvailable()* implementation is used to fulfill the request in the usual way.

One or more of the following values can be specified for the **<methods>** when setting the provider:

FIND_PACKAGE
The provider command accepts *find_package()* requests.

FETCHCONTENT_MAKEAVAILABLE_SERIAL
The provider command accepts *FetchContent_MakeAvailable()* requests. It expects each dependency to be fed to the provider command one at a time, not the whole list in one go.

Only one provider can be set at any point in time. If a provider is already set when **cmake_language(SET_DEPENDENCY_PROVIDER)** is called, the new provider replaces the previously set one. The specified **<command>** must already exist when **cmake_language(SET_DEPENDENCY_PROVIDER)** is called. As a special case, providing an empty string for the **<command>** and no **<methods>** will discard any previously set provider.

The dependency provider can only be set while processing one of the files specified by the *CMAKE_PROJECT_TOP_LEVEL_INCLUDES* variable. Thus, dependency providers can only be set as part of the first call to *project()*. Calling**cmak e_language(SET_DEPENDENCY_PROVIDER)** outside of that context will result in an error.

Added in version 3.30: The *PROPAGATE_TOP_LEVEL_INCLUDES_TO_TRY_COMPILE* global property can be set if the dependency provider also wants to be enabled in whole−project calls to *try_compile()*.

NOTE:
The choice of dependency provider should always be under the user's control. As a convenience, a project may choose to provide a file that users can list in their *CMAKE_PROJECT_TOP_LEVEL_INCLUDES* variable, but the use of such a file should always be the user's choice.

Provider commands

Providers define a single **<command>** to accept requests. The name of the command should be specific to that provider, not something overly generic that another provider might also use. This enables users to compose different providers in their own custom provider. The recommended form is**xxx_pr ovide_dependency()**, where **xxx** is the provider−specific part (e.g. **vcpkg_provide_dependency()**, **conan_provide_dependency()**, **ourcompany_provide_dependency()**, and so on).

```
xxx_provide_dependency(<method> [<method-specific-args>...])
```

Because some methods expect certain variables to be set in the calling scope, the provider command should typically be implemented as a macro rather than a function. This ensures it does not introduce a new variable scope.

The arguments CMake passes to the dependency provider depend on the type of request. The first

argument is always the method, and it will only ever be one of the **<methods>** that was specified when setting the provider.

FIND_PACKAGE

The **<method−specific−args>** will be everything passed to the *find_package()* call that requested the dependency. The first of these**<method−specific−ar gs>** will therefore always be the name of the dependency. Dependency names are case−sensitive for this method because *find_package()* treats them case−sensitively too.

If the provider command fulfills the request, it must set the same variable that *find_package()* expects to be set. For a dependency named **depName**, the provider must set **depName_FOUND** to true if it fulfilled the request. If the provider returns without setting this variable, CMake will assume the request was not fulfilled and will fall back to the built−in implementation.

If the provider needs to call the built−in *find_package()* implementation as part of its processing, it can do so by including the **BYPASS_PROVIDER** keyword as one of the arguments.

FETCHCONTENT_MAKEAVAILABLE_SERIAL

The **<method−specific−args>** will be everything passed to the *FetchContent_Declare()* call that corresponds to the requested dependency, with the following exceptions:

- If **SOURCE_DIR** or **BINARY_DIR** were not part of the original declared arguments, they will be added with their default values.
- If *FETCHCONTENT_TRY_FIND_PACKAGE_MODE* is set to **NEVER**, any **FIND_PACKAGE_ARGS** will be omitted.
- The **OVERRIDE_FIND_PACKAGE** keyword is always omitted.

The first of the **<method−specific−args>** will always be the name of the dependency. Dependency names are case−insensitive for this method because *FetchContent* also treats them case−insensitively.

If the provider fulfills the request, it should call *FetchContent_SetPopulated()*, passing the name of the dependency as the first argument. The**SOURCE_DIR** and **BIN ARY_DIR** arguments to that command should only be given if the provider makes the dependency's source and build directories available in exactly the same way as the built−in *FetchContent_MakeAvailable()* command.

If the provider returns without calling *FetchContent_SetPopulated()* for the named dependency, CMake will assume the request was not fulfilled and will fall back to the built−in implementation.

Note that empty arguments may be significant for this method (e.g. an empty string following a **GIT_SUBMODULES** keyword). Therefore, if forwarding these arguments on to another command, extra care must be taken to avoid such arguments being silently dropped.

If **FETCHCONTENT_SOURCE_DIR_<uppercaseDepName>** is set, then the dependency provider will never see requests for the **<depName>** dependency for this method. When the user sets such a variable, they are explicitly overriding where to get that dependency from and are taking on the responsibility that their overriding version meets any requirements for that dependency and is compatible with whatever else in the project uses it. Depending on the value of *FETCHCONTENT_TRY_FIND_PACKAGE_MODE* and whether the **OVERRIDE_FIND_PACKAGE** option was given to*F etchContent_Declare()*, having **FETCHCONTENT_SOURCE_DIR_<uppercaseDepName>** set may also prevent the dependency provider from seeing requests for a **find_package(depName)** call too.

Provider Examples

This first example only intercepts *find_package()* calls. The provider command runs an external tool which copies the relevant artifacts into a provider−specific directory, if that tool knows about the dependency. It

then relies on the built−in implementation to then find those artifacts. *FetchContent_MakeAvailable()* calls would not go through the provider.

mycomp_provider.cmake

```
# Always ensure we have the policy settings this provider expects
cmake_minimum_required(VERSION 3.24)

set(MYCOMP_PROVIDER_INSTALL_DIR ${CMAKE_BINARY_DIR}/mycomp_packages
  CACHE PATH "The directory this provider installs packages to"
)
# Tell the built−in implementation to look in our area first, unless
# the find_package() call uses NO_..._PATH options to exclude it
list(APPEND CMAKE_MODULE_PATH ${MYCOMP_PROVIDER_INSTALL_DIR}/cmake)
list(APPEND CMAKE_PREFIX_PATH ${MYCOMP_PROVIDER_INSTALL_DIR})

macro(mycomp_provide_dependency method package_name)
  execute_process(
    COMMAND some_tool ${package_name} −−installdir ${MYCOMP_PROVIDER_INSTALL_DIR}
    COMMAND_ERROR_IS_FATAL ANY
  )
endmacro()

cmake_language(
  SET_DEPENDENCY_PROVIDER mycomp_provide_dependency
  SUPPORTED_METHODS FIND_PACKAGE
)
```

The user would then typically use the above file like so:

```
cmake −DCMAKE_PROJECT_TOP_LEVEL_INCLUDES=/path/to/mycomp_provider.cmake ...
```

The next example demonstrates a provider that accepts both methods, but only handles one specific dependency. It enforces providing Google Test using *FetchContent*, but leaves all other dependencies to be fulfilled by CMake's built−in implementation. It accepts a few different names, which demonstrates one way of working around projects that hard−code an unusual or undesirable way of adding this particular dependency to the build. The example also demonstrates how to use the *list()* command to preserve variables that may be overwritten by a call to *FetchContent_MakeAvailable()*.

mycomp_provider.cmake

```
cmake_minimum_required(VERSION 3.24)

# Because we declare this very early, it will take precedence over any
# details the project might declare later for the same thing
include(FetchContent)
FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG        e2239ee6043f73722e7aa812a459f54a28552929 # release−1.11.0
)

# Both FIND_PACKAGE and FETCHCONTENT_MAKEAVAILABLE_SERIAL methods provide
# the package or dependency name as the first method−specific argument.
```

```
macro(mycomp_provide_dependency method dep_name)
  if("${dep_name}" MATCHES "^(gtest|googletest)$")
    # Save our current command arguments in case we are called recursively
    list(APPEND mycomp_provider_args ${method} ${dep_name})

    # This will forward to the built-in FetchContent implementation,
    # which detects a recursive call for the same thing and avoids calling
    # the provider again if dep_name is the same as the current call.
    FetchContent_MakeAvailable(googletest)

    # Restore our command arguments
    list(POP_BACK mycomp_provider_args dep_name method)

    # Tell the caller we fulfilled the request
    if("${method}" STREQUAL "FIND_PACKAGE")
      # We need to set this if we got here from a find_package() call
      # since we used a different method to fulfill the request.
      # This example assumes projects only use the gtest targets,
      # not any of the variables the FindGTest module may define.
      set(${dep_name}_FOUND TRUE)
    elseif(NOT "${dep_name}" STREQUAL "googletest")
      # We used the same method, but were given a different name to the
      # one we populated with. Tell the caller about the name it used.
      FetchContent_SetPopulated(${dep_name}
        SOURCE_DIR "${googletest_SOURCE_DIR}"
        BINARY_DIR "${googletest_BINARY_DIR}"
      )
    endif()
  endif()
endmacro()

cmake_language(
  SET_DEPENDENCY_PROVIDER mycomp_provide_dependency
  SUPPORTED_METHODS
    FIND_PACKAGE
    FETCHCONTENT_MAKEAVAILABLE_SERIAL
)
```

The final example demonstrates how to modify arguments to a *find_package()* call. It forces all such calls to have the **QUIET** keyword. It uses the**BYP ASS_PROVIDER** keyword to prevent calling the provider command recursively for the same dependency.

mycomp_provider.cmake

```
cmake_minimum_required(VERSION 3.24)

macro(mycomp_provide_dependency method)
  find_package(${ARGN} BYPASS_PROVIDER QUIET)
endmacro()

cmake_language(
  SET_DEPENDENCY_PROVIDER mycomp_provide_dependency
  SUPPORTED_METHODS FIND_PACKAGE
)
```

Getting current message log level
Added in version 3.25.

cmake_language(GET_MESSAGE_LOG_LEVEL <output_variable>)
Writes the current *message()* logging level into the given **<output_variable>**.

See *message()* for the possible logging levels.

The current message logging level can be set either using the *−−log−level* command line option of the *cmake(1)* program or using the *CMAKE_MESSAGE_LOG_LEVEL* variable.

If both the command line option and the variable are set, the command line option takes precedence. If neither are set, the default logging level is returned.

Terminating Scripts
Added in version 3.29.

cmake_language(EXIT <exit−code>)
Terminate the current *cmake −P* script and exit with **<exit−code>**.

This command works only in *script mode*. If used outside of that context, it will cause a fatal error.

The **<exit−code>** should be non−negative. If**<exit−code>** is ne gative, then the behavior is unspecified (e.g., on Windows the error code −1 becomes **0xffffffff**, and on Linux it becomes 255). Exit codes above 255 may not be supported by the underlying shell or platform, and some shells may interpret values above 125 specially. Therefore, it is advisable to only specify an **<exit−code>** in the range 0 to 125.

cmake_minimum_required
Require a minimum version of cmake.

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

Added in version 3.12: The optional **<policy_max>** version behavior; ignored in older CMake.

Sets the minimum required version of cmake for a project. Also updates the policy settings as explained below.

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

If the running version of CMake is lower than the **<min>** required version it will stop processing the project and report an error. The optional**<policy_max>** v ersion, if specified, must be at least the **<min>** version and sets the *Policy Version*. If the running version of CMake is older than 3.12, the extra **...** dots will be seen as version component separators, resulting in the **...<max>** part being ignored and preserving the pre−3.12 behavior of basing policies on **<min>**.

This command will set the value of the *CMAKE_MINIMUM_REQUIRED_VERSION* variable to **<min>**.

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

NOTE:

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

Calling **cmake_minimum_required()** inside a *function()* limits some effects to the function scope when invoked. For example, the *CMAKE_MINIMUM_REQUIRED_VERSION* variable won't be set in the calling scope. Functions do not introduce their own policy scope though, so policy settings of the caller *will* be affected (see below). Due to this mix of things that do and do not affect the calling scope, calling **cmake_minimum_required()** inside a function is generally discouraged.

Policy Version

cmake_minimum_required(VERSION <min>[...<max>]) implicitly invokes

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

This specifies that the current CMake code is written for the given range of CMake versions, **<min>[...<max>]**. It sets the "policy version" to:

- the range's **<max>** version, if specified, or to
- the **<min>** version, or to
- the value of the *CMAKE_POLICY_VERSION_MINIMUM* variable if it is higher than the other two versions.

The policy version effectively requests behavior preferred as of a given CMake version and tells newer CMake versions to warn about their new policies. All policies known to the running version of CMake and introduced in that version or earlier will be set to use **NEW** behavior. All policies introduced in later versions will be unset (unless the *CMAKE_POLICY_DEFAULT_CMP<NNNN>* variable sets a default). This effectively requests behavior preferred as of a given CMake version and tells newer CMake versions to warn about their new policies.

Changed in version 4.0: Compatibility with versions of CMake older than 3.5 is removed. Calls to *cmake_minimum_required(VERSION)* or *cmake_policy(VERSION)* that do not specify at least 3.5 as their policy version (optionally via **...<max>**) will produce an error in CMake 4.0 and above.

Changed in version 3.31: Compatibility with versions of CMake older than 3.10 is deprecated. Calls to *cmake_minimum_required(VERSION)* or *cmake_policy(VERSION)* that do not specify at least 3.10 as their policy version (optionally via **...<max>**) will produce a deprecation warning in CMake 3.31 and above.

Changed in version 3.27: Compatibility with versions of CMake older than 3.5 is deprecated. Calls to *cmake_minimum_required(VERSION)* or *cmake_policy(VERSION)* that do not specify at least 3.5 as their policy version (optionally via **...<max>**) will produce a deprecation warning in CMake 3.27 and above.

Changed in version 3.19: Compatibility with versions of CMake older than 2.8.12 is deprecated. Calls to *cmake_minimum_required(VERSION)* or *cmake_policy(VERSION)* that do not specify at least 2.8.12 as their policy version (optionally via **...<max>**) will produce a deprecation warning in CMake 3.19 and above.

See Also

- *cmake_policy()*

cmake_parse_arguments

Parse function or macro arguments.

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

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

Added in version 3.5: This command is implemented natively. Previously, it has been defined in the module *CMakeParseArguments*.

This command is for use in macros or functions. It processes the arguments given to that macro or function, and defines a set of variables which hold the values of the respective options.

The first signature reads arguments passed in the **<args>...**. This may be used in either a*macr o()* or a *function()*.

Added in version 3.7: The **PARSE_ARGV** signature is only for use in a *function()* body. In this case, the arguments that are parsed come from the **ARGV#** variables of the calling function. The parsing starts with the **<N>**−th argument, where **<N>** is an unsigned integer. This allows for the values to have special characters like **;** in them.

The **<options>** argument contains all options for the respective function or macro. These are keywords that have no value following them, like the **OPTIONAL** keyword of the *install()* command.

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

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

Changed in version 3.5: All keywords must be unique. Each keyword can only be specified once in any of the **<options>**, **<one_value_keywords>**, or **<multi_value_keywords>**. A warning will be emitted if uniqueness is violated.

When done, **cmake_parse_arguments** will consider for each of the keywords listed in **<options>**, **<one_value_keywords>**, and **<multi_value_keywords>**, a variable composed of the given **<prefix>** followed by **"_"** and the name of the respective keyword. For **<one_value_keywords>** and **<multi_value_keywords>**, these variables will then hold the respective value(s) from the argument list, or be undefined if the associated keyword was not given (policy *CMP0174* can also affect the behavior for **<one_value_keywords>**). For the **<options>** keywords, these variables will always be defined, and they will be set to **TRUE** if the keyword is present, or **FALSE** if it is not.

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

Added in version 3.15: **<one_value_keywords>** and **<multi_value_keywords>** that were given no values at all are collected in a variable **<prefix>_KEYWORDS_MISSING_VALUES** that will be undefined if all keywords received values. This can be checked to see if there were keywords without any values given.

Changed in version 3.31: If a **<one_value_keyword>** is followed by an empty string as its value, policy *CMP0174* controls whether a corresponding **<prefix>_<keyword>** variable is defined or not.

Choose a **<prefix>** carefully to avoid clashing with existing variable names. When used inside a function, it is usually suitable to use the prefix **arg**. There is a very strong convention that all keywords are fully uppercase, so this prefix results in variables of the form **arg_SOME_KEYWORD**. This makes the code more readable, and it minimizes the chance of clashing with cache variables, which also have a strong convention of being all uppercase.

```
function(my_install)
    set(options OPTIONAL FAST)
    set(oneValueArgs DESTINATION RENAME)
    set(multiValueArgs TARGETS CONFIGURATIONS)
    cmake_parse_arguments(PARSE_ARGV 0 arg
        "${options}" "${oneValueArgs}" "${multiValueArgs}"
    )

    # The above will set or unset variables with the following names:
    #   arg_OPTIONAL
    #   arg_FAST
    #   arg_DESTINATION
    #   arg_RENAME
    #   arg_TARGETS
    #   arg_CONFIGURATIONS
    #
    # The following will also be set or unset:
    #   arg_UNPARSED_ARGUMENTS
    #   arg_KEYWORDS_MISSING_VALUES
```

When used inside a macro, **arg** might not be a suitable prefix because the code will affect the calling scope. If another macro also called in the same scope were to use **arg** in its own call to **cmake_parse_arguments()**, and if there are any common keywords between the two macros, the later call's variables can overwrite or remove those of the earlier macro's call. Therefore, it is advisable to incorporate something unique from the macro name in the **<prefix>**, such as **arg_lowercase_macro_name**.

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

macro(my_special_install)
    # NOTE: Has the same keywords as my_install()
    set(options OPTIONAL FAST)
    set(oneValueArgs DESTINATION RENAME)
    set(multiValueArgs TARGETS CONFIGURATIONS)
    cmake_parse_arguments(arg_my_special_install
        "${options}" "${oneValueArgs}" "${multiValueArgs}"
```

```
        ${ARGN}
    )
    # ...
endmacro()
```

Suppose the above macros are called one after the other, like so:

```
my_install(TARGETS foo bar DESTINATION bin OPTIONAL blub CONFIGURATIONS)
my_special_install(TARGETS barry DESTINATION sbin RENAME FAST)
```

After these two calls, the following describes the variables that will be set or unset:

```
arg_my_install_OPTIONAL = TRUE
arg_my_install_FAST = FALSE # was not present in call to my_install
arg_my_install_DESTINATION = "bin"
arg_my_install_RENAME <UNSET> # was not present
arg_my_install_TARGETS = "foo;bar"
arg_my_install_CONFIGURATIONS <UNSET> # was not present
arg_my_install_UNPARSED_ARGUMENTS = "blub" # nothing expected after "OPTIONAL"
arg_my_install_KEYWORDS_MISSING_VALUES = "CONFIGURATIONS" # value was missing

arg_my_special_install_OPTIONAL = FALSE # was not present
arg_my_special_install_FAST = TRUE
arg_my_special_install_DESTINATION = "sbin"
arg_my_special_install_RENAME <UNSET> # value was missing
arg_my_special_install_TARGETS = "barry"
arg_my_special_install_CONFIGURATIONS <UNSET> # was not present
arg_my_special_install_UNPARSED_ARGUMENTS <UNSET>
arg_my_special_install_KEYWORDS_MISSING_VALUES = "RENAME"
```

Keywords terminate lists of values. If a keyword is given directly after a **<one_value_keyword>**, that preceding **<one_value_keyword>** receives no value and the keyword is added to the **<prefix>_KEYWORDS_MISSING_VALUES** variable. In the above example, the call to **my_special_install()** contains the **RENAME** keyword immediately followed by the **FAST** keyword. In this case,**F AST** terminates processing of the **RENAME** keyword. **arg_my_special_install_FAST** is set to **TRUE**, **arg_my_special_install_RENAME** is unset, and **arg_my_special_install_KEYWORDS_MISSING_VALUES** contains the value **RENAME**.

See Also

- *function()*
- *macro()*

cmake_path

Added in version 3.20.

This command is for the manipulation of paths. Only syntactic aspects of paths are handled, there is no interaction of any kind with any underlying file system. The path may represent a non−existing path or even one that is not allowed to exist on the current file system or platform. For operations that do interact with the filesystem, see the *file()* command.

NOTE:

The **cmake_path** command handles paths in the format of the build system (i.e. the host platform), not the target system. When cross−compiling, if the path contains elements that are not representable on the host platform (e.g. a drive letter when the host is not Windows), the results will be unpredictable.

Synopsis

```
  Conventions

  Path Structure And Terminology

  Normalization

  Decomposition
    cmake_path(GET <path-var> ROOT_NAME <out-var>)
    cmake_path(GET <path-var> ROOT_DIRECTORY <out-var>)
    cmake_path(GET <path-var> ROOT_PATH <out-var>)
    cmake_path(GET <path-var> FILENAME <out-var>)
    cmake_path(GET <path-var> EXTENSION [LAST_ONLY] <out-var>)
    cmake_path(GET <path-var> STEM [LAST_ONLY] <out-var>)
    cmake_path(GET <path-var> RELATIVE_PART <out-var>)
    cmake_path(GET <path-var> PARENT_PATH <out-var>)

  Query
    cmake_path(HAS_ROOT_NAME <path-var> <out-var>)
    cmake_path(HAS_ROOT_DIRECTORY <path-var> <out-var>)
    cmake_path(HAS_ROOT_PATH <path-var> <out-var>)
    cmake_path(HAS_FILENAME <path-var> <out-var>)
    cmake_path(HAS_EXTENSION <path-var> <out-var>)
    cmake_path(HAS_STEM <path-var> <out-var>)
    cmake_path(HAS_RELATIVE_PART <path-var> <out-var>)
    cmake_path(HAS_PARENT_PATH <path-var> <out-var>)
    cmake_path(IS_ABSOLUTE <path-var> <out-var>)
    cmake_path(IS_RELATIVE <path-var> <out-var>)
    cmake_path(IS_PREFIX <path-var> <input> [NORMALIZE] <out-var>)
    cmake_path(COMPARE <input1> <OP> <input2> <out-var>)

  Modification
    cmake_path(SET <path-var> [NORMALIZE] <input>)
    cmake_path(APPEND <path-var> [<input>...] [OUTPUT_VARIABLE <out-var>])
    cmake_path(APPEND_STRING <path-var> [<input>...] [OUTPUT_VARIABLE <out-var>])
    cmake_path(REMOVE_FILENAME <path-var> [OUTPUT_VARIABLE <out-var>])
    cmake_path(REPLACE_FILENAME <path-var> <input> [OUTPUT_VARIABLE <out-var>])
    cmake_path(REMOVE_EXTENSION <path-var> [LAST_ONLY] [OUTPUT_VARIABLE <out-var>])
    cmake_path(REPLACE_EXTENSION <path-var> [LAST_ONLY] <input> [OUTPUT_VARIABLE <o

  Generation
    cmake_path(NORMAL_PATH <path-var> [OUTPUT_VARIABLE <out-var>])
    cmake_path(RELATIVE_PATH <path-var> [BASE_DIRECTORY <input>] [OUTPUT_VARIABLE <
    cmake_path(ABSOLUTE_PATH <path-var> [BASE_DIRECTORY <input>] [NORMALIZE] [OUTPU

  Native Conversion
    cmake_path(NATIVE_PATH <path-var> [NORMALIZE] <out-var>)
    cmake_path(CONVERT <input> TO_CMAKE_PATH_LIST <out-var> [NORMALIZE])
    cmake_path(CONVERT <input> TO_NATIVE_PATH_LIST <out-var> [NORMALIZE])

  Hashing
    cmake_path(HASH <path-var> <out-var>)
```

Conventions

The following conventions are used in this command's documentation:

<path−var>
Always the name of a variable. For commands that expect a **<path−var>** as input, the variable must exist and it is expected to hold a single path.

<input>
A string literal which may contain a path, path fragment, or multiple paths with a special separator depending on the command. See the description of each command to see how this is interpreted.

<input>...
Zero or more string literal arguments.

<out−var>
The name of a variable into which the result of a command will be written.

Path Structure And Terminology

A path has the following structure (all components are optional, with some constraints):

```
root−name root−directory−separator (item−name directory−separator)* filename
```

root−name
Identifies the root on a filesystem with multiple roots (such as **"C:"** or **"//myserver"**). It is optional.

root−directory−separator
A directory separator that, if present, indicates that this path is absolute. If it is missing and the first element other than the **root−name** is an **item−name**, then the path is relative.

item−name
A sequence of characters that aren't directory separators. This name may identify a file, a hard link, a symbolic link, or a directory. Two special cases are recognized:

- The item name consisting of a single dot character **.** is a directory name that refers to the current directory.
- The item name consisting of two dot characters **..** is a directory name that refers to the parent directory.

The **(...)*** pattern shown above is to indicate that there can be zero or more item names, with multiple items separated by a **directory−separator**. The**()*** characters are not part of the path.

directory−separator
The only recognized directory separator is a forward slash character **/**. If this character is repeated, it is treated as a single directory separator. In other words, **/usr///lib** is the same as **/usr/lib**.

filename
A path has a **filename** if it does not end with a **directory−separator**. The**filename** is ef fectively the last **item−name** of the path, so it can also be a hard link, symbolic link or a directory.

A **filename** can have an *extension*. By default, the extension is defined as the sub−string beginning at the left−most period (including the period) and until the end of the **filename**. In commands that accept a **LAST_ONLY** keyword, **LAST_ONLY** changes the interpretation to the sub−string beginning at the right−most period.

The following exceptions apply to the above interpretation:

- If the first character in the **filename** is a period, that period is ignored (i.e. a **filename** like **".profile"** is treated as having no extension).

- If the **filename** is either **.** or **..**, it has no extension.

The *stem* is the part of the **filename** before the extension.

Some commands refer to a **root−path**. This is the concatenation of**r oot−name** and **root−directory−separator**, either or both of which can be empty. A**r elative−part** refers to the full path with any **root−path** removed.

Creating A Path Variable

While a path can be created with care using an ordinary *set()* command, it is recommended to use *cmake_path(SET)* instead, as it automatically converts the path to the required form where required. The *cmake_path(APPEND)* subcommand may be another suitable alternative where a path needs to be constructed by joining fragments. The following example compares the three methods for constructing the same path:

```
set(path1 "${CMAKE_CURRENT_SOURCE_DIR}/data")

cmake_path(SET path2 "${CMAKE_CURRENT_SOURCE_DIR}/data")

cmake_path(APPEND path3 "${CMAKE_CURRENT_SOURCE_DIR}" "data")
```

Modification and *Generation* sub−commands can either store the result in−place, or in a separate variable named after an **OUTPUT_VARIABLE** keyword. All other sub−commands store the result in a mandatory **<out−var>** variable.

Normalization

Some sub−commands support *normalizing* a path. The algorithm used to normalize a path is as follows:

1. If the path is empty, stop (the normalized form of an empty path is also an empty path).
2. Replace each **directory−separator**, which may consist of multiple separators, with a single / (**/a///b −−> /a/b**).
3. Remove each solitary period (**.**) and any immediately following **directory−separator** (**/a/./b/. −−> /a/b**).
4. Remove each **item−name** (other than **..**) that is immediately followed by a **directory−separator** and a **..**, along with any immediately following **directory−separator** (**/a/b/../c −−> a/c**).
5. If there is a **root−directory**, remove any **..** and any **directory−separators** immediately following them. The parent of the root directory is treated as still the root directory (**/../a −−> /a**).
6. If the last **item−name** is **..**, remove any trailing **directory−separator** (**../ −−> ..**).
7. If the path is empty by this stage, add a **dot** (normal form of **./** is **.**).

Decomposition

The following forms of the **GET** subcommand each retrieve a different component or group of components from a path. See *Path Structure And Terminology* for the meaning of each path component.

```
cmake_path(GET <path−var> ROOT_NAME <out−var>)
cmake_path(GET <path−var> ROOT_DIRECTORY <out−var>)
cmake_path(GET <path−var> ROOT_PATH <out−var>)
cmake_path(GET <path−var> FILENAME <out−var>)
cmake_path(GET <path−var> EXTENSION [LAST_ONLY] <out−var>)
cmake_path(GET <path−var> STEM [LAST_ONLY] <out−var>)
cmake_path(GET <path−var> RELATIVE_PART <out−var>)
cmake_path(GET <path−var> PARENT_PATH <out−var>)
```

If a requested component is not present in the path, an empty string will be stored in **<out−var>**. For

example, only Windows systems have the concept of a **root−name**, so when the host machine is non−Windows, the **ROOT_NAME** subcommand will always return an empty string.

For **PARENT_PATH**, if the *HAS_RELATIVE_PART* subcommand returns false, the result is a copy of **<path−var>**. Note that this implies that a root directory is considered to have a parent, with that parent being itself. Where *HAS_RELATIVE_PART* returns true, the result will essentially be **<path−var>** with one less element.

Root examples

```
set(path "c:/a")

cmake_path(GET path ROOT_NAME rootName)
cmake_path(GET path ROOT_DIRECTORY rootDir)
cmake_path(GET path ROOT_PATH rootPath)

message("Root name is \"${rootName}\"")
message("Root directory is \"${rootDir}\"")
message("Root path is \"${rootPath}\"")

Root name is "c:"
Root directory is "/"
Root path is "c:/"
```

Filename examples

```
set(path "/a/b")
cmake_path(GET path FILENAME filename)
message("First filename is \"${filename}\"")

# Trailing slash means filename is empty
set(path "/a/b/")
cmake_path(GET path FILENAME filename)
message("Second filename is \"${filename}\"")

First filename is "b"
Second filename is ""
```

Extension and stem examples

```
set(path "name.ext1.ext2")

cmake_path(GET path EXTENSION fullExt)
cmake_path(GET path STEM fullStem)
message("Full extension is \"${fullExt}\"")
message("Full stem is \"${fullStem}\"")

# Effect of LAST_ONLY
cmake_path(GET path EXTENSION LAST_ONLY lastExt)
cmake_path(GET path STEM LAST_ONLY lastStem)
message("Last extension is \"${lastExt}\"")
message("Last stem is \"${lastStem}\"")

# Special cases
set(dotPath "/a/.")
set(dotDotPath "/a/..")
set(someMorePath "/a/.some.more")
cmake_path(GET dotPath EXTENSION dotExt)
cmake_path(GET dotPath STEM dotStem)
```

```
cmake_path(GET dotDotPath EXTENSION dotDotExt)
cmake_path(GET dotDotPath STEM dotDotStem)
cmake_path(GET dotMorePath EXTENSION someMoreExt)
cmake_path(GET dotMorePath STEM someMoreStem)
message("Dot extension is \"${dotExt}\"")
message("Dot stem is \"${dotStem}\"")
message("Dot-dot extension is \"${dotDotExt}\"")
message("Dot-dot stem is \"${dotDotStem}\"")
message(".some.more extension is \"${someMoreExt}\"")
message(".some.more stem is \"${someMoreStem}\"")

Full extension is ".ext1.ext2"
Full stem is "name"
Last extension is ".ext2"
Last stem is "name.ext1"
Dot extension is ""
Dot stem is "."
Dot-dot extension is ""
Dot-dot stem is ".."
.some.more extension is ".more"
.some.more stem is ".some"
```

Relative part examples

```
set(path "c:/a/b")
cmake_path(GET path RELATIVE_PART result)
message("Relative part is \"${result}\"")

set(path "c/d")
cmake_path(GET path RELATIVE_PART result)
message("Relative part is \"${result}\"")

set(path "/")
cmake_path(GET path RELATIVE_PART result)
message("Relative part is \"${result}\"")

Relative part is "a/b"
Relative part is "c/d"
Relative part is ""
```

Path traversal examples

```
set(path "c:/a/b")
cmake_path(GET path PARENT_PATH result)
message("Parent path is \"${result}\"")

set(path "c:/")
cmake_path(GET path PARENT_PATH result)
message("Parent path is \"${result}\"")

Parent path is "c:/a"
Parent path is "c:/"
```

Query

Each of the **GET** subcommands has a corresponding **HAS_...** subcommand which can be used to discover whether a particular path component is present. See *Path Structure And Terminology* for the meaning of each path component.

```
cmake_path(HAS_ROOT_NAME <path-var> <out-var>)
cmake_path(HAS_ROOT_DIRECTORY <path-var> <out-var>)
cmake_path(HAS_ROOT_PATH <path-var> <out-var>)
cmake_path(HAS_FILENAME <path-var> <out-var>)
cmake_path(HAS_EXTENSION <path-var> <out-var>)
cmake_path(HAS_STEM <path-var> <out-var>)
cmake_path(HAS_RELATIVE_PART <path-var> <out-var>)
cmake_path(HAS_PARENT_PATH <path-var> <out-var>)
```

Each of the above follows the predictable pattern of setting <**out-var**> to true if the path has the associated component, or false otherwise. Note the following special cases:

- For **HAS_ROOT_PATH**, a true result will only be returned if at least one of **root-name** or **root-directory** is non-empty.
- For **HAS_PARENT_PATH**, the root directory is also considered to have a parent, which will be itself. The result is true except if the path consists of just a *filename*.

```
cmake_path(IS_ABSOLUTE <path-var> <out-var>)
```

Sets <**out-var**> to true if <**path-var**> is absolute. An absolute path is a path that unambiguously identifies the location of a file without reference to an additional starting location. On Windows, this means the path must have both a **root-name** and a **root-directory-separator** to be considered absolute. On other platforms, just a **root-directory-separator** is sufficient. Note that this means on Windows, **IS_ABSOLUTE** can be false while **HAS_ROOT_DIRECTORY** can be true.

```
cmake_path(IS_RELATIVE <path-var> <out-var>)
```

This will store the opposite of **IS_ABSOLUTE** in <**out-var**>.

```
cmake_path(IS_PREFIX <path-var> <input> [NORMALIZE] <out-var>)
```

Checks if <**path-var**> is the prefix of <**input**>.

When the **NORMALIZE** option is specified, <**path-var**> and <**input**> are *normalized* before the check.

```
set(path "/a/b/c")
cmake_path(IS_PREFIX path "/a/b/c/d" result) # result = true
cmake_path(IS_PREFIX path "/a/b" result)     # result = false
cmake_path(IS_PREFIX path "/x/y/z" result)   # result = false

set(path "/a/b")
cmake_path(IS_PREFIX path "/a/c/../b" NORMALIZE result)   # result = true

cmake_path(COMPARE <input1> EQUAL <input2> <out-var>)
cmake_path(COMPARE <input1> NOT_EQUAL <input2> <out-var>)
```

Compares the lexical representations of two paths provided as string literals. No normalization is performed on either path, except multiple consecutive directory separators are effectively collapsed into a single separator. Equality is determined according to the following pseudo-code logic:

```
if(NOT <input1>.root_name() STREQUAL <input2>.root_name())
  return FALSE

if(<input1>.has_root_directory() XOR <input2>.has_root_directory())
  return FALSE
```

```
Return FALSE if a relative portion of <input1> is not lexicographically
equal to the relative portion of <input2>. This comparison is performed path
component-wise. If all of the components compare equal, then return TRUE.
```

NOTE:
Unlike most other **cmake_path()** subcommands, the **COMPARE** subcommand takes literal strings as input, not the names of variables.

Modification

```
cmake_path(SET <path-var> [NORMALIZE] <input>)
```

Assign the **<input>** path to **<path-var>**. If**<input>** is a nati ve path, it is converted into a cmake-style path with forward-slashes (*/*). On Windows, the long filename marker is taken into account.

When the **NORMALIZE** option is specified, the path is *normalized* after the conversion.

For example:

```
set(native_path "c:\\a\\b/..\\c")
cmake_path(SET path "${native_path}")
message("CMake path is \"${path}\"")

cmake_path(SET path NORMALIZE "${native_path}")
message("Normalized CMake path is \"${path}\"")
```

Output:

```
CMake path is "c:/a/b/../c"
Normalized CMake path is "c:/a/c"
```

```
cmake_path(APPEND <path-var> [<input>...] [OUTPUT_VARIABLE <out-var>])
```

Append all the **<input>** arguments to the **<path-var>** using / as the **directory-separator**. Depending on the **<input>**, the previous contents of **<path-var>** may be discarded. For each **<input>** argument, the following algorithm (pseudo-code) applies:

```
# <path> is the contents of <path-var>

if(<input>.is_absolute() OR
   (<input>.has_root_name() AND
    NOT <input>.root_name() STREQUAL <path>.root_name()))
  replace <path> with <input>
  return()
endif()

if(<input>.has_root_directory())
  remove any root-directory and the entire relative path from <path>
elseif(<path>.has_filename() OR
       (NOT <path-var>.has_root_directory() OR <path>.is_absolute()))
  append directory-separator to <path>
endif()

append <input> omitting any root-name to <path>
```

```
cmake_path(APPEND_STRING <path-var> [<input>...] [OUTPUT_VARIABLE <out-var>])
```

Append all the **<input>** arguments to the **<path-var>** without adding any **directory-separator**.

```
cmake_path(REMOVE_FILENAME <path-var> [OUTPUT_VARIABLE <out-var>])
```

Removes the *filename* component (as returned by *GET ... FILENAME*) from **<path-var>**. After removal, any trailing **directory-separator** is left alone, if present.

If **OUTPUT_VARIABLE** is not given, then after this function returns, *HAS_FILENAME* returns false for **<path-var>**.

For example:

```
set(path "/a/b")
cmake_path(REMOVE_FILENAME path)
message("First path is \"${path}\"")

# filename is now already empty, the following removes nothing
cmake_path(REMOVE_FILENAME path)
message("Second path is \"${path}\"")
```

Output:

```
First path is "/a/"
Second path is "/a/"
```

```
cmake_path(REPLACE_FILENAME <path-var> <input> [OUTPUT_VARIABLE <out-var>])
```

Replaces the *filename* component from **<path-var>** with **<input>**. If**<path-v ar>** has no filename component (i.e. *HAS_FILEN AME* returns false), the path is unchanged. The operation is equivalent to the following:

```
cmake_path(HAS_FILENAME path has_filename)
if(has_filename)
  cmake_path(REMOVE_FILENAME path)
  cmake_path(APPEND path input);
endif()
```

```
cmake_path(REMOVE_EXTENSION <path-var> [LAST_ONLY]
                                       [OUTPUT_VARIABLE <out-var>])
```

Removes the *extension*, if any, from **<path-var>**.

```
cmake_path(REPLACE_EXTENSION <path-var> [LAST_ONLY] <input>
                             [OUTPUT_VARIABLE <out-var>])
```

Replaces the *extension* with **<input>**. Its effect is equivalent to the following:

```
cmake_path(REMOVE_EXTENSION path)
if(NOT "input" MATCHES "^\\.")
  cmake_path(APPEND_STRING path ".")
endif()
cmake_path(APPEND_STRING path "input")
```

Generation

```
cmake_path(NORMAL_PATH <path-var> [OUTPUT_VARIABLE <out-var>])
```

Normalize **<path-var>** according the steps described in *Normalization*.

```
cmake_path(RELATIVE_PATH <path-var> [BASE_DIRECTORY <input>]
                                    [OUTPUT_VARIABLE <out-var>])
```

Modifies **<path-var>** to make it relative to the **BASE_DIRECTORY** argument. If**B ASE_DIRECTORY** is not specified, the default base directory will be *CMAKE_CURRENT_SOURCE_DIR*.

For reference, the algorithm used to compute the relative path is the same as that used by C++ - *std::filesystem::path::lexically_relative*.

```
cmake_path(ABSOLUTE_PATH <path-var> [BASE_DIRECTORY <input>] [NORMALIZE]
                                    [OUTPUT_VARIABLE <out-var>])
```

If **<path-var>** is a relative path (*IS_RELATIVE* is true), it is evaluated relative to the given base directory specified by **BASE_DIRECTORY** option. If **BASE_DIRECTORY** is not specified, the default base directory will be *CMAKE_CURRENT_SOURCE_DIR*.

When the **NORMALIZE** option is specified, the path is *normalized* after the path computation.

Because **cmake_path()** does not access the filesystem, symbolic links are not resolved and any leading tilde is not expanded. To compute a real path with symbolic links resolved and leading tildes expanded, use the *file(REAL_PATH)* command instead.

Native Conversion

For commands in this section, *native* refers to the host platform, not the target platform when cross-compiling.

```
cmake_path(NATIVE_PATH <path-var> [NORMALIZE] <out-var>)
```

Converts a cmake-style **<path-var>** into a native path with platform-specific slashes (\ on Windows hosts and / elsewhere).

When the **NORMALIZE** option is specified, the path is *normalized* before the conversion.

```
cmake_path(CONVERT <input> TO_CMAKE_PATH_LIST <out-var> [NORMALIZE])
```

Converts a native **<input>** path into a cmake-style path with forward slashes (/). On Windows hosts, the long filename marker is taken into account. The input can be a single path or a system search path like **$ENV{PATH}**. A search path will be converted to a cmake-style list separated by **;** characters (on non-Windows platforms, this essentially means **:** separators are replaced with **;**). The result of the conversion is stored in the **<out-var>** variable.

When the **NORMALIZE** option is specified, the path is *normalized* before the conversion.

NOTE:

Unlike most other **cmake_path()** subcommands, the **CONVERT** subcommand takes a literal string as input, not the name of a variable.

```
cmake_path(CONVERT <input> TO_NATIVE_PATH_LIST <out-var> [NORMALIZE])
```

Converts a cmake-style **<input>** path into a native path with platform-specific slashes (\ on Windows hosts

and / elsewhere). The input can be a single path or a cmake−style list. A list will be converted into a native search path (**;**−separated on Windows, **:**−separated on other platforms). The result of the conversion is stored in the **<out−var>** variable.

When the **NORMALIZE** option is specified, the path is *normalized* before the conversion.

NOTE:
Unlike most other **cmake_path()** subcommands, the **CONVERT** subcommand takes a literal string as input, not the name of a variable.

For example:

```
set(paths "/a/b/c" "/x/y/z")
cmake_path(CONVERT "${paths}" TO_NATIVE_PATH_LIST native_paths)
message("Native path list is \"${native_paths}\"")
```

Output on Windows:

```
Native path list is "\a\b\c;\x\y\z"
```

Output on all other platforms:

```
Native path list is "/a/b/c:/x/y/z"
```

Hashing

```
cmake_path(HASH <path−var> <out−var>)
```

Compute a hash value of **<path−var>** such that for two paths **p1** and **p2** that compare equal (*COMPARE ... EQUAL*), the hash value of **p1** is equal to the hash value of **p2**. The path is always *normalized* before the hash is computed.

cmake_pkg_config

Added in version 3.31.

Process pkg−config format package files.

Synopsis

```
cmake_pkg_config(EXTRACT <package> [<version>] [...])
```

Introduction

This command generates CMake variables and targets from pkg−config format package files natively, without needing to invoke or even require the presence of a pkg−config implementation. A **<package>** is either an absolute path to a package file, or a package name to be searched for using the typical pkg−config search patterns. The optional **<version>** string has the same format and semantics as a pkg−config style version specifier, with the exception that if no comparison operator is specified **=** is assumed.

There are multiple signatures for this command, and some of the options are common between them. They are:

EXACT / QUIET / REQUIRED

The **EXACT** option requests that the version string be matched exactly (including empty string, if no version is provided), overriding the typical pkg−config version comparison algorithm. This will ignore any comparison operator attached to the version string.

The **QUIET** option disables informational messages, including those indicating that the package cannot be found if it is not **REQUIRED**. The **REQUIRED** option stops processing with an error

message if the package cannot be found.

STRICTNESS <mode>
Specify how strictly the contents of the package files will be verified during parsing and resolution. An invalid file, under the provided strictness mode, will cause the command to fail. Possible modes are:

- **STRICT**: Closely mirrors the behavior of the original FDO pkg−config. Variables and keywords must be unique. Variables must be defined before they are used. The Name, Description, and Version keywords must be present. The overall structure of the file must be valid and parsable.
- **PERMISSIVE**: Closely mirrors the behavior of the pkgconf implementation. Duplicate variables are overridden. Duplicate keywords are appended. Undefined variables resolve to empty strings. The Name, Description, and Version keywords must be present. The overall structure of the file must be valid and parsable.
- **BEST_EFFORT**: Same behavior as **PERMISSIVE** with regards to duplicate or uninitialized variables and keywords, but will not fail under any conditions. Package files which require BEST_EFFORT will fail validation under all other major implementations and should be fixed.

The default strictness is **PERMISSIVE**.

ENV_MODE
Specifies which environment variables will be queried when running a given command. Possible modes are:

- **FDO**: Queries only the original set of **PKG_CONFIG_*** environment variables used by the freedesktop.org **pkg−config** implementation.
- **PKGCONF**: Queries the more extensive set of environment variables used by the **pkgconf** implementation.
- **IGNORE**: Ignores the presence, absence, and value of environment variables entirely. In all cases an environment variable would be queried its treated as defined, but with a value of empty string for the purpose of the operation. This does not modify the current environment. For boolean environment variables, such as **PKG_CONFIG_ALLOW_***, this means they are evaluated as truthy.

  PKG_CONFIG_SYSROOT_PATH is a minor exception. When **ENV_MODE IGNORE** is used, no root path prepending will occur by default and **pc_sysrootdir** remains defaulted to **/**.

Target−generating subcommands always ignore flag−filtering environment variables. The default environment mode is **PKGCONF**.

PC_LIBDIR <path>...
Overrides the default search location for package files; also used to derive the **pc_path** package variable.

When this option is not provided, the default library directory is the first available of the following values:

1. **CMAKE_PKG_CONFIG_PC_LIB_DIRS**
2. The **PKG_CONFIG_LIBDIR** environment variable
3. The output of **pkg−config −−variable pc_path pkg−config**
4. A platform−dependent default value

PC_PATH <path>...
Overrides the supplemental package file directories which will be prepended to the search path; also used to derive the **pc_path** package variable.

When this option is not provided, the default paths are the first available of the following values:

1. **CMAKE_PKG_CONFIG_PC_PATH**
2. The **PKG_CONFIG_PATH** environment variable
3. Empty list

DISABLE_UNINSTALLED <bool>
Overrides the search behavior for "uninstalled" package files. These are package files with an "−uninstalled" suffix which describe packages integrated directly from a build tree.

Normally such package files have higher priority than "installed" packages. When **DISABLE_UNINSTALLED** is true, searching for "uninstalled" packages is disabled.

When this option is not provided, the default search behavior is determined by the first available of the following values:

1. **CMAKE_PKG_CONFIG_DISABLE_UNINSTALLED**
2. If the **PKG_CONFIG_DISABLE_UNINSTALLED** environment variable is defined the search is disabled, otherwise it is enabled.

PC_SYSROOT_DIR <path>
Overrides the root path which will be prepended to paths specified by **−I** compile flags and **−L** library search locations; also used to derive the **pc_sysrootdir** package variable.

When this option is not provided, the default root path is provided by the first available of the following values:

1. **CMAKE_PKG_CONFIG_SYSROOT_DIR**
2. The **PKG_CONFIG_SYSROOT_DIR** environment variable
3. If no root path is available, nothing will be prepended to include or library directory paths and **pc_sysrootdir** will be set to **/**

TOP_BUILD_DIR <path>
Overrides the top build directory path used to derived the **pc_top_builddir** package variable.

When this option is not provided, the default top build directory path is the first available of the following values:

1. **CMAKE_PKG_CONFIG_TOP_BUILD_DIR**
2. The **PKG_CONFIG_TOP_BUILD_DIR** environment variable
3. If no top build directory path is available, the **pc_top_builddir** package variable is not set

Signatures

cmake_pkg_config(EXTRACT <package> [<version>] [...])
Extract the contents of the package into variables.

```
cmake_pkg_config(EXTRACT <package> [<version>]
                 [REQUIRED] [EXACT] [QUIET]
                 [STRICTNESS <mode>]
                 [ENV_MODE <mode>]
                 [PC_LIBDIR <path>...]
                 [PC_PATH <path>...]
                 [DISABLE_UNINSTALLED <bool>]
                 [PC_SYSROOT_DIR <path>]
                 [TOP_BUILD_DIR <path>]
                 [SYSTEM_INCLUDE_DIRS <path>...]
```

```
[SYSTEM_LIBRARY_DIRS <path>...]
[ALLOW_SYSTEM_INCLUDES <bool>]
[ALLOW_SYSTEM_LIBS <bool>])
```

The following variables will be populated from the contents of package file:

Variable	Type	Definition
CMAKE_PKG_CONFIG_NAME	String	Value of the **Name** keyword
CMAKE_PKG_CONFIG_DESCRIPTION	String	Value of the **Description** keyword
CMAKE_PKG_CONFIG_VERSION	String	Value of the **Version** keyword
CMAKE_PKG_CONFIG_PROVIDES	List	Value of the **Provides** keyword
CMAKE_PKG_CONFIG_REQUIRES	List	Value of the **Requires** keyword
CMAKE_PKG_CONFIG_CONFLICTS	List	Value of the **Conflicts** keyword
CMAKE_PKG_CONFIG_CFLAGS	String	Value of the **CFlags** / **Cflags** keyword
CMAKE_PKG_CONFIG_INCLUDES	List	All **−I** prefixed flags from **CMAKE_PKG_CONFIG_CFLAGS**
CMAKE_PKG_CONFIG_COMPILE_OPTIONS	List	All flags not prefixed with **−I** from **CMAKE_PKG_CONFIG_CFLAGS**
CMAKE_PKG_CONFIG_LIBS	String	Value of the **Libs** keyword
CMAKE_PKG_CONFIG_LIBDIRS	List	All **−L** prefixed flags from **CMAKE_PKG_CONFIG_LIBS**
CMAKE_PKG_CONFIG_LIBNAMES	List	All **−l** prefixed flags from **CMAKE_PKG_CONFIG_LIBS**
CMAKE_PKG_CONFIG_LINK_OPTIONS	List	All flags not prefixed with **−L** or **−l** from **CMAKE_PKG_CONFIG_LIBS**
CMAKE_PKG_CONFIG_*_PRIVATE	*	**CFLAGS** / **LIBS** / **REQUIRES** and derived, but in their **.private** suffix forms

SYSTEM_INCLUDE_DIRS

Overrides the "system" directories for the purpose of flag mangling include directories in **CMAKE_PKG_CONFIG_CFLAGS** and derived variables.

When this option is not provided, the default directories are provided by the first available of the following values:

1. **CMAKE_PKG_CONFIG_SYS_INCLUDE_DIRS**

2. The **PKG_CONFIG_SYSTEM_INCLUDE_PATH** environment variable
3. The output of **pkgconf −−variable pc_system_includedirs pkg−config**
4. A platform−dependent default value

Additionally, when the **ENV_MODE** is **PKGCONF** the **CMAKE_PKG_CONFIG_PKGCONF_INCLUDES** variable will be concatenated to the list if available. If it is not available, the following environment variables will be queried and concatenated:

- **CPATH**
- **C_INCLUDE_PATH**
- **CPLUS_INCLUDE_PATH**
- **OBJC_INCLUDE_PATH**
- **INCLUDE** (Windows Only)

SYSTEM_LIBRARY_DIRS
Overrides the "system" directories for the purpose of flag mangling library directories in **CMAKE_PKG_CONFIG_LIBS** and derived variables.

When this option is not provided, the default directories are provided by the first available of the following values:

1. **CMAKE_PKG_CONFIG_SYS_LIB_DIRS**
2. The **PKG_CONFIG_SYSTEM_LIBRARY_PATH** environment variable
3. The output of **pkgconf −−variable pc_system_libdirs pkg−config**
4. A platform−dependent default value

Additionally, when the **ENV_MODE** is **PKGCONF** the **CMAKE_PKG_CONFIG_PKGCONF_LIB_DIRS** variable will be concatenated to the list if available. If it is not available, the **LIBRARY_PATH** environment variable will be queried and concatenated.

ALLOW_SYSTEM_INCLUDES
Preserves "system" directories during flag mangling of include directories in **CMAKE_PKG_CONFIG_CFLAGS** and derived variables.

When this option is not provided, the default value is determined by the first available of the following values:

1. **CMAKE_PKG_CONFIG_ALLOW_SYS_INCLUDES**
2. If the **PKG_CONFIG_ALLOW_SYSTEM_CFLAGS** environment variable is defined the flags are preserved, otherwise they are filtered during flag mangling.

ALLOW_SYSTEM_LIBS
Preserves "system" directories during flag mangling of library directories in **CMAKE_PKG_CONFIG_LIBS** and derived variables.

When this option is not provided, the default value is determined by the first available of the following values:

1. **CMAKE_PKG_CONFIG_ALLOW_SYS_LIBS**
2. If the **PKG_CONFIG_ALLOW_SYSTEM_LIBS** environment variable is defined the flags are preserved, otherwise they are filtered during flag mangling.

cmake_policy
Manage CMake Policy settings. See the *cmake−policies(7)* manual for defined policies.

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

Setting Policies by CMake Version

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

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

Added in version 3.12: The optional **<max>** version.

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

This specifies that the current CMake code is written for the given range of CMake versions, **<min>[...<max>]**. It sets the "policy version" to:

- the range's **<max>** version, if specified, or to
- the **<min>** version, or to
- the value of the *CMAKE_POLICY_VERSION_MINIMUM* variable if it is higher than the other two versions.

The policy version effectively requests behavior preferred as of a given CMake version and tells newer CMake versions to warn about their new policies. All policies known to the running version of CMake and introduced in that version or earlier will be set to use **NEW** behavior. All policies introduced in later versions will be unset (unless the *CMAKE_POLICY_DEFAULT_CMP<NNNN>* variable sets a default). This effectively requests behavior preferred as of a given CMake version and tells newer CMake versions to warn about their new policies.

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

Changed in version 4.0: Compatibility with versions of CMake older than 3.5 is removed. Calls to *cmake_minimum_required(VERSION)* or *cmake_policy(VERSION)* that do not specify at least 3.5 as their policy version (optionally via **...<max>**) will produce an error in CMake 4.0 and above.

Changed in version 3.31: Compatibility with versions of CMake older than 3.10 is deprecated. Calls to *cmake_minimum_required(VERSION)* or *cmake_policy(VERSION)* that do not specify at least 3.10 as their policy version (optionally via **...<max>**) will produce a deprecation warning in CMake 3.31 and above.

Changed in version 3.27: Compatibility with versions of CMake older than 3.5 is deprecated. Calls to *cmake_minimum_required(VERSION)* or *cmake_policy(VERSION)* that do not specify at least 3.5 as their

policy version (optionally via **...<max>**) will produce a deprecation warning in CMake 3.27 and above.

Changed in version 3.19: Compatibility with versions of CMake older than 2.8.12 is deprecated. Calls to *cmake_minimum_required(VERSION)* or *cmake_policy(VERSION)* that do not specify at least 2.8.12 as their policy version (optionally via **...<max>**) will produce a deprecation warning in CMake 3.19 and above.

Setting Policies Explicitly

cmake_policy(SET CMP<NNNN> NEW|OLD)

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

NOTE:

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

Checking Policy Settings

cmake_policy(GET CMP<NNNN> <variable>)

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

CMake Policy Stack

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

cmake_policy(PUSH)

Create a new entry on the policy stack.

cmake_policy(POP)

Remove the last policy stack entry created with **cmake_policy(PUSH)**.

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

Added in version 3.25: The *block(SCOPE_FOR POLICIES)* command offers a more flexible and more secure way to manage the policy stack. The pop action is done automatically when leaving the block scope, so there is no need to precede each *return()* with a call to *cmake_policy(POP)*.

```
# stack management with cmake_policy()
function(my_func)
  cmake_policy(PUSH)
  cmake_policy(SET ...)
  if (<cond1>)
    ...
    cmake_policy(POP)
    return()
  elseif(<cond2>)
```

```
    ...
    cmake_policy(POP)
    return()
  endif()
  ...
  cmake_policy(POP)
endfunction()

# stack management with block()/endblock()
function(my_func)
  block(SCOPE_FOR POLICIES)
    cmake_policy(SET ...)
    if (<cond1>)
      ...
      return()
    elseif(<cond2>)
      ...
      return()
    endif()
    ...
  endblock()
endfunction()
```

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

See Also

- *cmake_minimum_required()*

configure_file

Copy a file to another location and modify its contents.

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

Copies an **<input>** file to an **<output>** file while performing *transformations* of the input file content.

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

Options

The options are:

<input>

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

<output>

Path to the output file or directory. A relative path is treated with respect to the value of *CMAKE_CURRENT_BINARY_DIR*. If the path names an existing directory the output file is

placed in that directory with the same file name as the input file. If the path contains non−existent directories, they are created.

NO_SOURCE_PERMISSIONS
Added in version 3.19.

Do not transfer the permissions of the input file to the output file. The copied file permissions default to the standard 644 value (−rw−r−−r−−).

USE_SOURCE_PERMISSIONS
Added in version 3.20.

Transfer the permissions of the input file to the output file. This is already the default behavior if none of the three permissions−related keywords are given (**NO_SOURCE_PERMISSIONS**, **USE_SOURCE_PERMISSIONS** or **FILE_PERMISSIONS**). The**USE_SOURCE_PERMISSIONS** keyword mostly serves as a way of making the intended behavior clearer at the call site.

FILE_PERMISSIONS <permissions>...
Added in version 3.20.

Ignore the input file's permissions and use the specified **<permissions>** for the output file instead.

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

ESCAPE_QUOTES
Escape any substituted quotes with backslashes (C−style).

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

NEWLINE_STYLE <style>
Specify the newline style for the output file. Specify **UNIX** or **LF** for **\n** newlines, or specify **DOS**, **WIN32**, or **CRLF** for **\r\n** newlines. This option may not be used with**COPY ONLY**.

Transformations
Variables referenced in the input file content as **@VAR@**, **${VAR}**, **$CACHE{VAR}**, and *environment variables* referenced as **$ENV{VAR}**, will each be replaced with the current value of the variable, or the empty string if the variable is not defined. Furthermore, input lines of the form

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

will be replaced with either

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

or

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

depending on whether **VAR** is set in CMake to any value not considered a false constant by the *if()* command. The "..." content on the line after the variable name, if any, is processed as above.

Unlike lines of the form **#cmakedefine VAR ...**, in lines of the form **#cmakedefine01 VAR**, **VAR** itself will

expand to **VAR 0** or **VAR 1** rather than being assigned the value **...**. Therefore, input lines of the form

```
#cmakedefine01 VAR
```

will be replaced with either

```
#define VAR 0
```

or

```
#define VAR 1
```

Input lines of the form **#cmakedefine01 VAR ...** will expand as **#cmakedefine01 VAR ... 0** or **#cmakedefine01 VAR ... 1**, which may lead to undefined behavior.

Added in version 3.10: The result lines (with the exception of the **#undef** comments) can be indented using spaces and/or tabs between the **#** character and the **cmakedefine** or **cmakedefine01** words. This whitespace indentation will be preserved in the output lines:

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

will be replaced, if **VAR** is defined, with

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

Example

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

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

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

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

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

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

Otherwise it will contain:

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

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

```
target_include_directories(<target> [SYSTEM] <INTERFACE|PUBLIC|PRIVATE> "${CMAKE_
```

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

See Also

- *file(GENERATE)*

continue

Added in version 3.2.

Continue to the top of enclosing foreach or while loop.

```
continue()
```

The **continue()** command allows a cmake script to abort the rest of the current iteration of a *foreach()* or *while()* loop, and start at the top of the next iteration.

See also the *break()* command.

else

Starts the else portion of an if block.

```
else([<condition>])
```

See the *if()* command.

The optional <**condition**> argument is supported for backward compatibility only and is not evaluated.

elseif

Starts an elseif portion of an if block.

```
elseif(<condition>)
```

See the *if()* command, especially for the syntax and logic of the <**condition**>.

endblock

Added in version 3.25.

Ends a list of commands in a *block()* and removes the scopes created by the *block()* command.

```
endblock()
```

endforeach

Ends a list of commands in a foreach block.

```
endforeach([<loop_var>])
```

See the *foreach()* command.

The optional <**loop_var**> argument is supported for backward compatibility only. If used it must be a verbatim repeat of the <**loop_var**> argument of the opening **foreach** clause.

endfunction

Ends a list of commands in a function block.

```
endfunction([<name>])
```

See the *function()* command.

The optional <**name**> argument is supported for backward compatibility only. If used it must be a verbatim repeat of the <**name**> argument of the opening **function** command.

endif
Ends a list of commands in an if block.

```
endif([<condition>])
```

See the *if()* command.

The optional <**condition**> argument is supported for backward compatibility only. If used it must be a verbatim repeat of the argument of the opening **if** clause.

endmacro
Ends a list of commands in a macro block.

```
endmacro([<name>])
```

See the *macro()* command.

The optional <**name**> argument is supported for backward compatibility only. If used it must be a verbatim repeat of the <**name**> argument of the opening **macro** command.

endwhile
Ends a list of commands in a while block.

```
endwhile([<condition>])
```

See the *while()* command.

The optional <**condition**> argument is supported for backward compatibility only. If used it must be a verbatim repeat of the argument of the opening **while** clause.

execute_process
Execute one or more child processes.

```
execute_process(COMMAND <cmd1> [<arguments>]
                [COMMAND <cmd2> [<arguments>]]...
                [WORKING_DIRECTORY <directory>]
                [TIMEOUT <seconds>]
                [RESULT_VARIABLE <variable>]
                [RESULTS_VARIABLE <variable>]
                [OUTPUT_VARIABLE <variable>]
                [ERROR_VARIABLE <variable>]
                [INPUT_FILE <file>]
                [OUTPUT_FILE <file>]
                [ERROR_FILE <file>]
                [OUTPUT_QUIET]
                [ERROR_QUIET]
                [COMMAND_ECHO <where>]
                [OUTPUT_STRIP_TRAILING_WHITESPACE]
                [ERROR_STRIP_TRAILING_WHITESPACE]
                [ENCODING <name>]
                [ECHO_OUTPUT_VARIABLE]
                [ECHO_ERROR_VARIABLE]
```

```
[COMMAND_ERROR_IS_FATAL <ANY|LAST|NONE>])
```

Runs the given sequence of one or more commands.

Commands are executed concurrently as a pipeline, with the standard output of each process piped to the standard input of the next. A single standard error pipe is used for all processes.

execute_process runs commands while CMake is configuring the project, prior to build system generation. Use the *add_custom_target()* and *add_custom_command()* commands to create custom commands that run at build time.

Options:

COMMAND
A child process command line.

CMake executes the child process using operating system APIs directly:

- On POSIX platforms, the command line is passed to the child process in an **argv[]** style array. No intermediate shell is executed, so shell operators such as > are treated as normal arguments.
- On Windows platforms, the command line is encoded as a string such that child processes using *CommandLineToArgvW* will decode the original arguments.

  If the command runs a **.exe**, **.com**, or other executable, no intermediate command interpreter is executed, so shell operators such as > are treated as normal arguments.

  If the command runs a **.bat** or **.cmd** script, it is executed through the **cmd** command interpreter. The command interpreter does not use *CommandLineToArgvW*, so some arguments may be received by the script with extra quoting.

  Changed in version 4.0: **.bat** and **.cmd** scripts are now explicitly executed through the command interpreter by prepending **cmd /c call** to the command line. Previously, they were implicitly executed through **cmd /c**, without **call**, by undocumented behavior of *CreateProcessW*.

Use the **INPUT_***, **OUTPUT_***, and **ERROR_*** options to redirect stdin, stdout, and stderr.

For **sequential execution** of multiple commands use multiple **execute_process** calls each with a single **COMMAND** argument.

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

TIMEOUT
After the specified number of seconds (fractions allowed), all unfinished child processes will be terminated, and the **RESULT_VARIABLE** will be set to a string mentioning the "timeout".

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

RESULTS_VARIABLE <variable>
Added in version 3.10.

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

INPUT_FILE <file>
<file> is attached to the standard input pipe of the *first* **COMMAND** process.

OUTPUT_FILE <file>
<file> is attached to the standard output pipe of the *last* **COMMAND** process.

ERROR_FILE <file>
<file> is attached to the standard error pipe of *all* **COMMAND** processes.

Added in version 3.3: If the same **<file>** is named for both **OUTPUT_FILE** and **ERROR_FILE** then it will be used for both standard output and standard error pipes.

OUTPUT_QUIET, ERROR_QUIET
The standard output on **OUTPUT_VARIABLE** or standard error on **ERROR_VARIABLE** are not connected (no variable content). The ***_FILE** and **ECHO_*_VARIABLE** options are not affected.

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

ECHO_OUTPUT_VARIABLE, ECHO_ERROR_VARIABLE
Added in version 3.18.

The standard output or standard error will not be exclusively redirected to the specified variables.

The output will be duplicated into the specified variables and also onto standard output or standard error analogous to the **tee** Unix command.

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

COMMAND_ECHO <where>
Added in version 3.15.

The command being run will be echo'ed to **<where>** with **<where>** being set to one of **STDERR**, **STDOUT** or **NONE**. See the*CMAKE_EXECUTE_PR OCESS_COMMAND_ECHO* variable for a way to control the default behavior when this option is not present.

ENCODING <name>
Added in version 3.8.

On Windows, the encoding that is used to decode output from the process. Ignored on other platforms. Valid encoding names are:

NONE Perform no decoding. This assumes that the process output is encoded in the same way as CMake's internal encoding (UTF−8).

This was the default in CMake 3.14 and older.

AUTO Use the current active console's codepage or if that isn't available then use ANSI.

This was the default in CMake 3.15 through 3.30.

ANSI Use the ANSI codepage.

OEM Use the original equipment manufacturer (OEM) code page.

UTF−8 Added in version 3.11.

Use the UTF−8 codepage.

This is the default since CMake 3.31. See policy *CMP0176*.

UTF8 Use the UTF−8 codepage. Use of this name is discouraged in favor of **UTF−8** to match the *UTF−8 RFC* naming convention.

COMMAND_ERROR_IS_FATAL <ANY|LAST|NONE>
Added in version 3.19.

The option following **COMMAND_ERROR_IS_FATAL** determines the behavior when an error is encountered:

ANY If any of the commands in the list of commands fail, the **execute_process()** command halts with an error.

LAST If the last command in the list of commands fails, the **execute_process()** command halts with an error. Commands earlier in the list will not cause a fatal error.

NONE Added in version 4.0.

Regardless of any of the commands failing, the **execute_process()** command will not halt with an error.

Added in version 4.0: If not provided, the *CMAKE_EXECUTE_PROCESS_COMMAND_ERROR_IS_FATAL* variable is checked. If the variable is not set, the default is **NONE**. If**RESUL T_VARIABLE** or **RESULTS_VARIABLE** is supplied, *CMAKE_EXECUTE_PROCESS_COMMAND_ERROR_IS_FATAL* is ignored.

file

File manipulation command.

This command is dedicated to file and path manipulation requiring access to the filesystem.

For other path manipulation, handling only syntactic aspects, have a look at *cmake_path()* command.

NOTE:
The sub−commands *RELATIVE_PATH*, *TO_CMAKE_PATH* and *TO_NATIVE_PATH* has been superseded, respectively, by sub−commands *RELATIVE_PATH*, *CONVERT ... TO_CMAKE_PATH_LIST* and *CONVERT ... TO_NATIVE_PATH_LIST* of *cmake_path()* command.

Synopsis

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

Writing
  file({WRITE | APPEND} <filename> <content>...)
```

```
  file({TOUCH | TOUCH_NOCREATE} <file>...)
  file(GENERATE OUTPUT <output-file> [...])
  file(CONFIGURE OUTPUT <output-file> CONTENT <content> [...])

Filesystem
  file({GLOB | GLOB_RECURSE} <out-var> [...] <globbing-expr>...)
  file(MAKE_DIRECTORY <directories>... [...])
  file({REMOVE | REMOVE_RECURSE } <files>...)
  file(RENAME <oldname> <newname> [...])
  file(COPY_FILE <oldname> <newname> [...])
  file({COPY | INSTALL} <file>... DESTINATION <dir> [...])
  file(SIZE <filename> <out-var>)
  file(READ_SYMLINK <linkname> <out-var>)
  file(CREATE_LINK <original> <linkname> [...])
  file(CHMOD <files>... <directories>... PERMISSIONS <permissions>... [...])
  file(CHMOD_RECURSE <files>... <directories>... PERMISSIONS <permissions>... [..

Path Conversion
  file(REAL_PATH <path> <out-var> [BASE_DIRECTORY <dir>] [EXPAND_TILDE])
  file(RELATIVE_PATH <out-var> <directory> <file>)
  file({TO_CMAKE_PATH | TO_NATIVE_PATH} <path> <out-var>)

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

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

Archiving
  file(ARCHIVE_CREATE OUTPUT <archive> PATHS <paths>... [...])
  file(ARCHIVE_EXTRACT INPUT <archive> [...])

Handling Runtime Binaries
  file(GET_RUNTIME_DEPENDENCIES [...])
```

Reading

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

Read content from a file called **<filename>** and store it in a **<variable>**. Optionally start from the given **<offset>** and read at most **<max-in>** bytes. The **HEX** option causes data to be converted to a hexadecimal representation (useful for binary data). If the **HEX** option is specified, letters in the output (**a** through **f**) are in lowercase.

file(STRINGS <filename> <variable> <options>...)

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

LENGTH_MAXIMUM <max-len>

Consider only strings of at most a given length.

LENGTH_MINIMUM <min-len>

Consider only strings of at least a given length.

LIMIT_COUNT <max-num>

Limit the number of distinct strings to be extracted.

LIMIT_INPUT <max−in>
Limit the number of input bytes to read from the file.

LIMIT_OUTPUT <max−out>
Limit the number of total bytes to store in the **<variable>**.

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

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

REGEX <regex>
Consider only strings that match the given regular expression, as described under *string(REGEX)*.

Changed in version 3.29: Capture groups from the last match in the file are stored in *CMAKE_MATCH_<n>*, similar to *string(REGEX MATCHALL)*. See policy *CMP0159*.

ENCODING <encoding−type>
Added in version 3.1.

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

Added in version 3.2: Added the **UTF−16LE**, **UTF−16BE**, **UTF−32LE**, **UTF−32BE** encodings.

For example, the code

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

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

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

file(TIMESTAMP <filename> <variable> [<format>] [UTC])
Compute a string representation of the modification time of **<filename>** and store it in **<variable>**. Should the command be unable to obtain a timestamp variable will be set to the empty string ("").

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

Writing

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

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

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

file(TOUCH <files>...)

file(TOUCH_NOCREATE <files>...)
Added in version 3.12.

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

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

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

Changed in version 3.30: **<files>** can be an empty list. CMake 3.29 and earlier required at least one file to be given.

file(GENERATE [...])
Generate an output file for each build configuration supported by the current *CMake Generator*. Evaluate *generator expressions* from the input content to produce the output content.

```
file(GENERATE OUTPUT <output-file>
     <INPUT <input-file>|CONTENT <content>>
     [CONDITION <expression>] [TARGET <target>]
     [NO_SOURCE_PERMISSIONS | USE_SOURCE_PERMISSIONS |
      FILE_PERMISSIONS <permissions>...]
     [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF]])
```

The options are:

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

CONTENT <content>
Use the content given explicitly as input.

INPUT <input-file>
Use the content from a given file as input.

Changed in version 3.10: A relative path is treated with respect to the value of *CMAKE_CURRENT_SOURCE_DIR*. See policy *CMP0070*.

OUTPUT <output-file>
Specify the output file name to generate. Use generator expressions such as *$<CONFIG>* to specify a configuration-specific output file name. Multiple configurations may generate the same output file only if the generated content is identical. Otherwise, the **<output-file>** must evaluate to an unique name for each configuration.

Changed in version 3.10: A relative path (after evaluating generator expressions) is treated with respect to the value of *CMAKE_CURRENT_BINARY_DIR*. See policy *CMP0070*.

TARGET <target>
Added in version 3.19.

Specify which target to use when evaluating generator expressions that require a target for evaluation (e.g. *$<COMPILE_FEATURES:...>*, *$<TARGET_PROPERTY:prop>*).

NO_SOURCE_PERMISSIONS
Added in version 3.20.

The generated file permissions default to the standard 644 value (−rw−r−−r−−).

USE_SOURCE_PERMISSIONS
Added in version 3.20.

Transfer the file permissions of the **INPUT** file to the generated file. This is already the default behavior if none of the three permissions−related keywords are given (**NO_SOURCE_PERMISSIONS**, **USE_SOURCE_PERMISSIONS** or **FILE_PERMISSIONS**). The**USE_SOURCE_PERMISSIONS** keyword mostly serves as a way of making the intended behavior clearer at the call site. It is an error to specify this option without **INPUT**.

FILE_PERMISSIONS <permissions>...
Added in version 3.20.

Use the specified permissions for the generated file.

NEWLINE_STYLE <style>
Added in version 3.20.

Specify the newline style for the generated file. Specify **UNIX** or **LF** for **\n** newlines, or specify **DOS**, **WIN32**, or **CRLF** for **\r\n** newlines.

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

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

file(CONFIGURE OUTPUT <output−file> CONTENT <content> [ESCAPE_QUOTES] [@ONLY] [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF]])
Added in version 3.18.

Generate an output file using the input given by **CONTENT** and substitute variable values referenced as **@VAR@** or **${VAR}** contained therein. The substitution rules behave the same as the *configure_file()* command. In order to match *configure_file()*'s behavior, generator expressions are not supported for both **OUTPUT** and **CONTENT**, and the output file is only modified and its timestamp updated if the content is changed or the file previously didn't exist.

The arguments are:

OUTPUT <output−file>
Specify the output file name to generate. A relative path is treated with respect to the value of *CMAKE_CURRENT_BINARY_DIR*. **<output−file>** does not support generator expressions.

CONTENT <content>
Use the content given explicitly as input. **<content>** does not support generator e xpressions.

ESCAPE_QUOTES
Escape any substituted quotes with backslashes (C−style).

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

NEWLINE_STYLE <style>
Specify the newline style for the output file. Specify **UNIX** or **LF** for **\n** newlines, or specify **DOS**, **WIN32**, or **CRLF** for **\r\n** newlines.

Filesystem

file(GLOB <variable> [LIST_DIRECTORIES true|false] [RELATIVE <path>] [CONFIGURE_DEPENDS] <globbing−expressions>...)

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

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

Changed in version 3.6: The results will be ordered lexicographically.

On Windows and macOS, globbing is case−insensitive even if the underlying filesystem is case−sensitive (both filenames and globbing expressions are converted to lowercase before matching). On other platforms, globbing is case−sensitive.

Added in version 3.3: By default **GLOB** lists directories. Directories are omitted in the result if **LIST_DIRECTORIES** is set to false.

Added in version 3.12: If the **CONFIGURE_DEPENDS** flag is specified, CMake will add logic to the main build system check target to rerun the flagged **GLOB** commands at build time. If any of the outputs change, CMake will regenerate the build system.

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

Examples of globbing expressions include:

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

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

Added in version 3.3: By default **GLOB_RECURSE** omits directories from result list. Setting **LIST_DIRECTORIES** to true adds directories to result list. If **FOLLOW_SYMLINKS** is given or policy *CMP0009* is not set to **NEW** then **LIST_DIRECTORIES** treats symlinks as directories.

Examples of recursive globbing include:

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

file(MAKE_DIRECTORY <directories>... [RESULT <result>])

Create the given directories and their parents as needed. Relative input paths are evaluated with respect to the current source directory.

The options are:

RESULT <result>

Added in version 3.31.

Set **<result>** variable to **0** on success or an error message otherwise. If **RESULT** is not specified and the operation fails, an error is emitted.

Changed in version 3.30: **<directories>** can be an empty list. CMake 3.29 and earlier required at least one directory to be given.

file(REMOVE <files>...)

file(REMOVE_RECURSE <files>...)

Remove the given files. The **REMOVE_RECURSE** mode will remove the given files and directories, including non−empty directories. No error is emitted if a given file does not exist. Relative input paths are evaluated with respect to the current source directory.

Changed in version 3.15: Empty input paths are ignored with a warning. Previous versions of CMake interpreted empty strings as a relative path with respect to the current directory and removed its contents.

file(RENAME <oldname> <newname> [RESULT <result>] [NO_REPLACE])

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

The options are:

RESULT <result>

Added in version 3.21.

Set **<result>** variable to **0** on success or an error message otherwise. If **RESULT** is not specified and the operation fails, an error is emitted.

NO_REPLACE
Added in version 3.21.

If the **<newname>** path already exists, do not replace it. If **RESULT <result>** is used, the result variable will be set to **NO_REPLACE**. Otherwise, an error is emitted.

file(COPY_FILE <oldname> <newname> [RESULT <result>] [ONLY_IF_DIFFERENT] [INPUT_MAY_BE_RECENT])
Added in version 3.21.

Copy a file from **<oldname>** to **<newname>**. Directories are not supported. Symlinks are ignored and **<oldfile>**'s content is read and written to **<newname>** as a new file.

The options are:

RESULT <result>
Set **<result>** variable to **0** on success or an error message otherwise. If **RESULT** is not specified and the operation fails, an error is emitted.

ONLY_IF_DIFFERENT
If the **<newname>** path already exists, do not replace it if the file's contents are already the same as **<oldname>** (this avoids updating **<newname>**'s timestamp).

INPUT_MAY_BE_RECENT
Added in version 3.26.

Tell CMake that the input file may have been recently created. This is meaningful only on Windows, where files may be inaccessible for a short time after they are created. With this option, if permission is denied, CMake will retry reading the input a few times.

This sub−command has some similarities to *configure_file()* with the **COPYONLY** option. An important difference is that *configure_file()* creates a dependency on the source file, so CMake will be re−run if it changes. The **file(COPY_FILE)** sub−command does not create such a dependency.

See also the *file(COPY)* sub−command just below which provides further file−copying capabilities.

file(COPY [...])

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

```
file(<COPY|INSTALL> <files>... DESTINATION <dir>
     [NO_SOURCE_PERMISSIONS | USE_SOURCE_PERMISSIONS]
     [FILE_PERMISSIONS <permissions>...]
     [DIRECTORY_PERMISSIONS <permissions>...]
     [FOLLOW_SYMLINK_CHAIN]
```

```
[FILES_MATCHING]
[[PATTERN <pattern> | REGEX <regex>]
 [EXCLUDE] [PERMISSIONS <permissions>...]] [...])
```

NOTE:
For a simple file copying operation, the *file(COPY_FILE)* sub−command just above may be easier to use.

Added in version 3.15: If **FOLLOW_SYMLINK_CHAIN** is specified, **COPY** will recursively resolve the symlinks at the paths given until a real file is found, and install a corresponding symlink in the destination for each symlink encountered. For each symlink that is installed, the resolution is stripped of the directory, leaving only the filename, meaning that the new symlink points to a file in the same directory as the symlink. This feature is useful on some Unix systems, where libraries are installed as a chain of symlinks with version numbers, with less specific versions pointing to more specific versions. **FOLLOW_SYMLINK_CHAIN** will install all of these symlinks and the library itself into the destination directory. For example, if you have the following directory structure:

- **/opt/foo/lib/libfoo.so.1.2.3**
- **/opt/foo/lib/libfoo.so.1.2 −> libfoo.so.1.2.3**
- **/opt/foo/lib/libfoo.so.1 −> libfoo.so.1.2**
- **/opt/foo/lib/libfoo.so −> libfoo.so.1**

and you do:

```
file(COPY /opt/foo/lib/libfoo.so DESTINATION lib FOLLOW_SYMLINK_CHAIN)
```

This will install all of the symlinks and **libfoo.so.1.2.3** itself into **lib**.

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

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

Changed in version 3.22: The environment variable *CMAKE_INSTALL_MODE* can override the default copying behavior of *file(INSTALL)*.

file(SIZE <filename> <variable>)
Added in version 3.14.

Determine the file size of the **<filename>** and put the result in **<variable>** variable. Requires that **<filename>** is a valid path pointing to a file and is readable.

file(READ_SYMLINK <linkname> <variable>)
Added in version 3.14.

Query the symlink **<linkname>** and stores the path it points to in the result **<variable>**. If **<linkname>** does not exist or is not a symlink, CMake issues a fatal error.

Note that this command returns the raw symlink path and does not resolve a relative path. The following is an example of how to ensure that an absolute path is obtained:

```
set(linkname "/path/to/foo.sym")
file(READ_SYMLINK "${linkname}" result)
if(NOT IS_ABSOLUTE "${result}")
  get_filename_component(dir "${linkname}" DIRECTORY)
  set(result "${dir}/${result}")
endif()
```

file(CREATE_LINK <original> <linkname> [RESULT <result>] [COPY_ON_ERROR] [SYMBOLIC])

Added in version 3.14.

Create a link **<linkname>** that points to **<original>**. It will be a hard link by default, but providing the **SYMBOLIC** option results in a symbolic link instead. Hard links require that **original** exists and is a file, not a directory. If**<linkname>** already e xists, it will be overwritten.

The **<result>** variable, if specified, receives the status of the operation. It is set to **0** upon success or an error message otherwise. If **RESULT** is not specified and the operation fails, a fatal error is emitted.

Specifying **COPY_ON_ERROR** enables copying the file as a fallback if creating the link fails. It can be useful for handling situations such as **<original>** and **<linkname>** being on different drives or mount points, which would make them unable to support a hard link.

file(CHMOD <files>... <directories>... [PERMISSIONS <permissions>...] [FILE_PERMISSIONS <permissions>...] [DIRECTORY_PERMISSIONS <permissions>...])

Added in version 3.19.

Set the permissions for the **<files>...** and **<directories>...** specified. Valid permissions are **OWNER_READ**, **OWNER_WRITE**, **OWNER_EXECUTE**, **GROUP_READ**, **GROUP_WRITE**, **GROUP_EXECUTE**, **WORLD_READ**, **WORLD_WRITE**, **WORLD_EXECUTE**, **SETUID**, **SETGID**.

Valid combination of keywords are:

PERMISSIONS
: All items are changed.

FILE_PERMISSIONS
: Only files are changed.

DIRECTORY_PERMISSIONS
: Only directories are changed.

PERMISSIONS and FILE_PERMISSIONS
: **FILE_PERMISSIONS** overrides **PERMISSIONS** for files.

PERMISSIONS and DIRECTORY_PERMISSIONS
: **DIRECTORY_PERMISSIONS** overrides **PERMISSIONS** for directories.

FILE_PERMISSIONS and DIRECTORY_PERMISSIONS
: Use **FILE_PERMISSIONS** for files and **DIRECTORY_PERMISSIONS** for directories.

file(CHMOD_RECURSE <files>... <directories>... [PERMISSIONS <permissions>...] [FILE_PERMISSIONS <permissions>...] [DIRECTORY_PERMISSIONS <permissions>...])
Added in version 3.19.

Same as *CHMOD*, but change the permissions of files and directories present in the **<directories>...** recursively.

Path Conversion

file(REAL_PATH <path> <out−var> [BASE_DIRECTORY <dir>] [EXPAND_TILDE])
Added in version 3.19.

Compute the absolute path to an existing file or directory with symlinks resolved. The options are:

BASE_DIRECTORY <dir>
If the provided **<path>** is a relative path, it is evaluated relative to the given base directory **<dir>**. If no base directory is provided, the default base directory will be *CMAKE_CURRENT_SOURCE_DIR*.

EXPAND_TILDE
Added in version 3.21.

If the **<path>** is ~ or starts with ~/, the ~ is replaced by the user's home directory. The path to the home directory is obtained from environment variables. On Windows, the **USERPROFILE** environment variable is used, falling back to the **HOME** environment variable if **USERPROFILE** is not defined. On all other platforms, only **HOME** is used.

Changed in version 3.28: All symlinks are resolved before collapsing **../** components. See policy *CMP0152*.

file(RELATIVE_PATH <variable> <directory> <file>)
Compute the relative path from a **<directory>** to a **<file>** and store it in the **<variable>**.

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

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

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

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

Transfer

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

file(UPLOAD <file> <url> <options>...)
The **DOWNLOAD** subcommand downloads the given **<url>** to a local **<file>**. The**UPLO AD** mode uploads a local **<file>** to a given **<url>**.

Added in version 3.19: If **<file>** is not specified for **file(DOWNLOAD)**, the file is not saved. This can be useful if you want to know if a file can be downloaded (for example, to check that it exists)

without actually saving it anywhere.

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

INACTIVITY_TIMEOUT <seconds>
Terminate the operation after a period of inactivity.

LOG <variable>
Store a human−readable log of the operation in a variable.

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

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

TIMEOUT <seconds>
Terminate the operation after a given total time has elapsed.

USERPWD <username>:<password>
Added in version 3.7.

Set username and password for operation.

HTTPHEADER <HTTP−header>
Added in version 3.7.

HTTP header for **DOWNLOAD** and **UPLOAD** operations. **HTTPHEADER** can be repeated for multiple options:

```
file(DOWNLOAD <url>
     HTTPHEADER "Authorization: Bearer <auth-token>"
     HTTPHEADER "UserAgent: Mozilla/5.0")
```

NETRC <level>
Added in version 3.11.

Specify whether the .netrc file is to be used for operation. If this option is not specified, the value of the *CMAKE_NETRC* variable will be used instead.

Valid levels are:

IGNORED
The .netrc file is ignored. This is the default.

OPTIONAL
The .netrc file is optional, and information in the URL is preferred. The file will be scanned to find which ever information is not specified in the URL.

REQUIRED
The .netrc file is required, and information in the URL is ignored.

NETRC_FILE <file>
Added in version 3.11.

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

TLS_VERSION <min>
Added in version 3.30.

Specify minimum TLS version for **https://** URLs. If this option is not specified, the value of the *CMAKE_TLS_VERSION* variable or *CMAKE_TLS_VERSION* environment variable will be used instead. See *CMAKE_TLS_VERSION* for allowed values.

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

TLS_VERIFY <ON|OFF>
Specify whether to verify the server certificate for **https://** URLs. If this option is not specified, the value of the *CMAKE_TLS_VERIFY* variable or *CMAKE_TLS_VERIFY* environment variable will be 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.

Added in version 3.18: Added support to **file(UPLOAD)**.

TLS_CAINFO <file>
Specify a custom Certificate Authority file for **https://** URLs. If this option is not specified, the value of the *CMAKE_TLS_CAINFO* variable will be used instead.

Added in version 3.18: Added support to **file(UPLOAD)**.

For **https://** URLs CMake must be built with SSL/TLS support.

Additional options to **DOWNLOAD** are:

EXPECTED_HASH <algorithm>=<value>
Verify that the downloaded content hash matches the expected value, where **<algorithm>** is one of the algorithms supported by *<HASH>*. If the file already exists and matches the hash, the download is skipped. If the file already exists and does not match the hash, the file is downloaded again. If after download the file does not match the hash, the operation fails with an error. It is an error to specify this option if **DOWNLOAD** is not given a **<file>**.

EXPECTED_MD5 <value>
Historical short−hand for **EXPECTED_HASH MD5=<value>**. It is an error to specify this if **DOWNLOAD** is not given a **<file>**.

RANGE_START <value>
Added in version 3.24.

Offset of the start of the range in file in bytes. Could be omitted to download up to the specified **RANGE_END**.

RANGE_END <value>
Added in version 3.24.

Offset of the end of the range in file in bytes. Could be omitted to download everything from the specified **RANGE_START** to the end of file.

Locking

file(LOCK <path> [DIRECTORY] [RELEASE] [GUARD <FUNCTION|FILE|PROCESS>] [RESULT_VARIABLE <variable>] [TIMEOUT <seconds>])
Added in version 3.2.

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

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

Trying to lock the same file twice is not allowed. Any intermediate directories and the file itself will be created if they not exist. The**GU ARD** and **TIMEOUT** options are ignored on the **RELEASE** operation.

Archiving

file(ARCHIVE_CREATE OUTPUT <archive> PATHS <paths>... [FORMAT <format>] [COMPRESSION <compression> [COMPRESSION_LEVEL <compression−level>]] [MTIME <mtime>] [WORKING_DIRECTORY <dir>] [VERBOSE])
Added in version 3.18.

Creates the specified **<archive>** file with the files and directories listed in **<paths>**. Note that **<paths>** must list actual files or directories; wildcards are not supported.

The options are:

FORMAT <format>
Specify the archive format. Supported values for **<format>** are **7zip**, **gnutar**, **pax**, **paxr**, **raw** and **zip**. If**FORMA T** is not given, the default format is **paxr**.

COMPRESSION <compression>
Some archive formats allow the type of compression to be specified. The **7zip** and **zip** archive formats already imply a specific type of compression. The other formats use no compression by default, but can be directed to do so with the **COMPRESSION** option. Valid values for **<compression>** are **None**, **BZip2**, **GZip**, **XZ**, and **Zstd**.

NOTE:
With **FORMAT** set to **raw**, only one file will be compressed with the compression type specified by **COMPRESSION**.

COMPRESSION_LEVEL <compression−level>
Added in version 3.19.

The compression level can be specified with the **COMPRESSION_LEVEL** option. The **<compression−level>** should be between 0−9, with the default being 0. The **COMPRESSION** option must be present when **COMPRESSION_LEVEL** is given.

Added in version 3.26: The **<compression−level>** of the **Zstd** algorithm can be set between 0−19.

MTIME <mtime>
Specify the modification time recorded in tarball entries.

WORKING_DIRECTORY <dir>
Added in version 3.31.

Specify the directory in which the archive creation operation will be executed. Paths in the **<paths>** argument can be relative to this directory. If this option is not provided, the current working directory will be used by default.

VERBOSE
Enable verbose output from the archive operation.

file(ARCHIVE_EXTRACT INPUT <archive> [DESTINATION <dir>] [PATTERNS <pattern>...] [LIST_ONLY] [VERBOSE] [TOUCH])
Added in version 3.18.

Extracts or lists the content of the specified **<archive>**.

The options are:

DESTINATION <dir>
Specify the directory under which the content of the archive will be extracted. If the directory does not exist, it will be created. If **DESTINATION** is not given, the current binary directory will be used.

PATTERNS <pattern>...
Extract/list only files and directories that match one of the given patterns. Wildcards are supported. If the**P ATTERNS** option is not given, the entire archive will be listed or extracted.

LIST_ONLY
List the files in the archive rather than extract them.

TOUCH
Added in version 3.24.

Give extracted files a current local timestamp instead of extracting file timestamps from the archive.

VERBOSE
Enable verbose output from the extraction operation.

NOTE:
The working directory for this subcommand is the **DESTINATION** directory (provided or computed) except when **LIST_ONLY** is specified. Therefore, outside of script mode, it may be best to provide absolute paths to **INPUT** archives as they are unlikely to be extracted where a relative path works.

Handling Runtime Binaries

file(GET_RUNTIME_DEPENDENCIES [...])

Added in version 3.16.

Recursively get the list of libraries depended on by the given files:

```
file(GET_RUNTIME_DEPENDENCIES
  [RESOLVED_DEPENDENCIES_VAR <deps_var>]
  [UNRESOLVED_DEPENDENCIES_VAR <unresolved_deps_var>]
  [CONFLICTING_DEPENDENCIES_PREFIX <conflicting_deps_prefix>]
  [EXECUTABLES <executable_files>...]
  [LIBRARIES <library_files>...]
  [MODULES <module_files>...]
  [DIRECTORIES <directories>...]
  [BUNDLE_EXECUTABLE <bundle_executable_file>]
  [PRE_INCLUDE_REGEXES <regexes>...]
  [PRE_EXCLUDE_REGEXES <regexes>...]
  [POST_INCLUDE_REGEXES <regexes>...]
  [POST_EXCLUDE_REGEXES <regexes>...]
  [POST_INCLUDE_FILES <files>...]
  [POST_EXCLUDE_FILES <files>...]
  )
```

Please note that this sub−command is not intended to be used in project mode. It is intended for use at install time, either from code generated by the *install(RUNTIME_DEPENDENCY_SET)* command, or from code provided by the project via *install(CODE)* or *install(SCRIPT)*. For example:

```
install(CODE [[
  file(GET_RUNTIME_DEPENDENCIES
    # ...
    )
  ]])
```

The arguments are as follows:

RESOLVED_DEPENDENCIES_VAR <deps_var>
Name of the variable in which to store the list of resolved dependencies.

UNRESOLVED_DEPENDENCIES_VAR <unresolved_deps_var>
Name of the variable in which to store the list of unresolved dependencies. If this variable is not specified, and there are any unresolved dependencies, an error is issued.

CONFLICTING_DEPENDENCIES_PREFIX <conflicting_deps_prefix>
Variable prefix in which to store conflicting dependency information. Dependencies are conflicting if two files with the same name are found in two different directories. The list of filenames that conflict are stored in **<conflicting_deps_prefix>_FILENAMES**. For each filename, the list of paths that were found for that filename are stored in **<conflicting_deps_prefix>_<filename>**.

EXECUTABLES <executable_files>...
List of executable files to read for dependencies. These are executables that are typically created with *add_executable()*, but they do not have to be created by CMake. On Apple platforms, the paths to these files determine the value of **@executable_path** when recursively resolving the libraries. Specifying any kind of library (**STATIC**, **MODULE**, or **SHARED**) here will result in undefined behavior.

LIBRARIES <library_files>...
List of library files to read for dependencies. These are libraries that are typically created with *add_library(SHARED)*, but they do not have to be created by CMake. Specifying **STATIC** libraries, **MODULE** libraries, or executables here will result in undefined behavior.

MODULES <module_files>...
List of loadable module files to read for dependencies. These are modules that are typically created with *add_library(MODULE)*, but they do not have to be created by CMake. They are typically used by calling **dlopen()** at runtime rather than linked at link time with **ld −l**. Specifying**ST ATIC** libraries, **SHARED** libraries, or executables here will result in undefined behavior.

DIRECTORIES <directories>...
List of additional directories to search for dependencies. On Linux platforms, these directories are searched if the dependency is not found in any of the other usual paths. If it is found in such a directory, a warning is issued, because it means that the file is incomplete (it does not list all of the directories that contain its dependencies). On Windows platforms, these directories are searched if the dependency is not found in any of the other search paths, but no warning is issued, because searching other paths is a normal part of Windows dependency resolution. On Apple platforms, this argument has no effect.

BUNDLE_EXECUTABLE <bundle_executable_file>
Executable to treat as the "bundle executable" when resolving libraries. On Apple platforms, this argument determines the value of **@executable_path** when recursively resolving libraries for **LIBRARIES** and **MODULES** files. It has no effect on **EXECUTABLES** files. On other platforms, it has no effect. This is typically (but not always) one of the executables in the **EXECUTABLES** argument which designates the "main" executable of the package.

The following arguments specify filters for including or excluding libraries to be resolved. See below for a full description of how they work.

PRE_INCLUDE_REGEXES <regexes>...
List of pre−include regexes through which to filter the names of not−yet−resolved dependencies.

PRE_EXCLUDE_REGEXES <regexes>...
List of pre−exclude regexes through which to filter the names of not−yet−resolved dependencies.

POST_INCLUDE_REGEXES <regexes>...
List of post−include regexes through which to filter the names of resolved dependencies.

POST_EXCLUDE_REGEXES <regexes>...
List of post−exclude regexes through which to filter the names of resolved dependencies.

POST_INCLUDE_FILES <files>...
Added in version 3.21.

List of post−include filenames through which to filter the names of resolved dependencies. Symlinks are resolved when attempting to match these filenames.

POST_EXCLUDE_FILES <files>...
Added in version 3.21.

List of post−exclude filenames through which to filter the names of resolved dependencies. Symlinks are resolved when attempting to match these filenames.

These arguments can be used to exclude unwanted system libraries when resolving the dependencies, or to include libraries from a specific directory. The filtering works as follows:

1. If the not−yet−resolved dependency matches any of the **PRE_INCLUDE_REGEXES**, steps 2 and 3 are skipped, and the dependency resolution proceeds to step 4.
2. If the not−yet−resolved dependency matches any of the **PRE_EXCLUDE_REGEXES**, dependency resolution stops for that dependency.
3. Otherwise, dependency resolution proceeds.
4. **file(GET_RUNTIME_DEPENDENCIES)** searches for the dependency according to the linking rules of the platform (see below).
5. If the dependency is found, and its full path matches one of the **POST_INCLUDE_REGEXES** or **POST_INCLUDE_FILES**, the full path is added to the resolved dependencies, and **file(GET_RUNTIME_DEPENDENCIES)** recursively resolves that library's own dependencies. Otherwise, resolution proceeds to step 6.
6. If the dependency is found, but its full path matches one of the **POST_EXCLUDE_REGEXES** or **POST_EXCLUDE_FILES**, it is not added to the resolved dependencies, and dependency resolution stops for that dependency.
7. If the dependency is found, and its full path does not match either **POST_INCLUDE_REGEXES**, **POST_INCLUDE_FILES**, **POST_EXCLUDE_REGEXES**, or **POST_EXCLUDE_FILES**, the full path is added to the resolved dependencies, and **file(GET_RUNTIME_DEPENDENCIES)** recursively resolves that library's own dependencies.

Different platforms have different rules for how dependencies are resolved. These specifics are described here.

On Linux platforms, library resolution works as follows:

1. If the depending file does not have any **RUNPATH** entries, and the library exists in one of the depending file's **RPATH** entries, or its parents', in that order, the dependency is resolved to that file.
2. Otherwise, if the depending file has any **RUNPATH** entries, and the library exists in one of those entries, the dependency is resolved to that file.
3. Otherwise, if the library exists in one of the directories listed by **ldconfig**, the dependency is resolved to that file.
4. Otherwise, if the library exists in one of the **DIRECTORIES** entries, the dependency is resolved to that file. In this case, a warning is issued, because finding a file in one of the **DIRECTORIES** means that the depending file is not complete (it does not list all the directories from which it pulls dependencies).
5. Otherwise, the dependency is unresolved.

Changed in version 3.31: Resolution of each encountered library file name occurs at most once while processing a given root ELF file (executable or shared object). If a library file name is encountered again in the dependency tree, the original resolution is assumed. This behavior more closely matches the dynamic loader's behavior on Linux.

On Windows platforms, library resolution works as follows:

1. DLL dependency names are converted to lowercase for matching filters. Windows DLL names are case−insensitive, and some linkers mangle the case of the DLL dependency names. However, this makes it more difficult for **PRE_INCLUDE_REGEXES**, **PRE_EXCLUDE_REGEXES**, **POST_INCLUDE_REGEXES**, and **POST_EXCLUDE_REGEXES** to properly filter DLL names − every regex would have to check for both uppercase and lowercase letters. For example:

   ```
   file(GET_RUNTIME_DEPENDENCIES
     # ...
     PRE_INCLUDE_REGEXES "^[Mm][Yy][Ll][Ii][Bb][Rr][Aa][Rr][Yy]\\.[Dd][Ll][L
     )
   ```

   Converting the DLL name to lowercase allows the regexes to only match lowercase names, thus simplifying the regex. For example:

   ```
   file(GET_RUNTIME_DEPENDENCIES
     # ...
     PRE_INCLUDE_REGEXES "^mylibrary\\.dll$"
     )
   ```

   This regex will match **mylibrary.dll** regardless of how it is cased, either on disk or in the depending file. (For example, it will match **mylibrary.dll**, **MyLibrary.dll**, and **MYLIBRARY.DLL**.)

   Changed in version 3.27: The conversion to lowercase only applies while matching filters. Results reported after filtering case−preserve each DLL name as it is found on disk, if resolved, and otherwise as it is referenced by the dependent binary.

   Prior to CMake 3.27, the results were reported with lowercase DLL file names, but the directory portion retained its casing.

2. (**Not yet implemented**) If the depending file is a Windows Store app, and the dependency is listed as a dependency in the application's package manifest, the dependency is resolved to that file.
3. Otherwise, if the library exists in the same directory as the depending file, the dependency is resolved to that file.
4. Otherwise, if the library exists in either the operating system's **system32** directory or the **Windows** directory, in that order, the dependency is resolved to that file.
5. Otherwise, if the library exists in one of the directories specified by **DIRECTORIES**, in the order they are listed, the dependency is resolved to that file. In this case, a warning is not issued, because searching other directories is a normal part of Windows library resolution.
6. Otherwise, the dependency is unresolved.

On Apple platforms, library resolution works as follows:

1. If the dependency starts with **@executable_path/**, and an **EXECUTABLES** argument is in the process of being resolved, and replacing **@executable_path/** with the directory of the executable yields an existing file, the dependency is resolved to that file.
2. Otherwise, if the dependency starts with **@executable_path/**, and there is a **BUNDLE_EXECUTABLE** argument, and replacing **@executable_path/** with the directory of the bundle executable yields an existing file, the dependency is resolved to that file.
3. Otherwise, if the dependency starts with **@loader_path/**, and replacing **@loader_path/** with the directory of the depending file yields an existing file, the dependency is resolved to that file.
4. Otherwise, if the dependency starts with **@rpath/**, and replacing **@rpath/** with one of the **RPATH** entries of the depending file yields an existing file, the dependency is resolved to that file. Note that**RP ATH** entries that start with **@executable_path/** or **@loader_path/** also have these items replaced with the appropriate path.
5. Otherwise, if the dependency is an absolute file that exists, the dependency is resolved to that file.
6. Otherwise, the dependency is unresolved.

This function accepts several variables that determine which tool is used for dependency resolution:

CMAKE_GET_RUNTIME_DEPENDENCIES_PLATFORM
Determines which operating system and executable format the files are built for. This could be one of several values:

- **linux+elf**
- **windows+pe**
- **macos+macho**

If this variable is not specified, it is determined automatically by system introspection.

CMAKE_GET_RUNTIME_DEPENDENCIES_TOOL
Determines the tool to use for dependency resolution. It could be one of several values, depending on the value of *CMAKE_GET_RUNTIME_DEPENDENCIES_PLATFORM*:

CMAKE_GET_RUNTIME_DEPENDENCIES_PLATFORM	**CMAKE_GET_RUNTIME_DEPENDENCIES_TOOL**
linux+elf	**objdump**
windows+pe	**objdump** or **dumpbin**
macos+macho	**otool**

If this variable is not specified, it is determined automatically by system introspection.

CMAKE_GET_RUNTIME_DEPENDENCIES_COMMAND
Determines the path to the tool to use for dependency resolution. This is the actual path to **objdump**, **dumpbin**, or **otool**.

If this variable is not specified, it is determined by the value of *CMAKE_OBJDUMP* variable if set, else by system introspection.

Added in version 3.18: Uses *CMAKE_OBJDUMP* if set.

find_file
A short−hand signature is:

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

The general signature is:

```
find_file (
          <VAR>
          name | NAMES name1 [name2 ...]
          [HINTS [path | ENV var]...]
          [PATHS [path | ENV var]...]
          [REGISTRY_VIEW (64|32|64_32|32_64|HOST|TARGET|BOTH)]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [VALIDATOR function]
          [DOC "cache documentation string"]
          [NO_CACHE]
          [REQUIRED]
          [NO_DEFAULT_PATH]
          [NO_PACKAGE_ROOT_PATH]
          [NO_CMAKE_PATH]
          [NO_CMAKE_ENVIRONMENT_PATH]
          [NO_SYSTEM_ENVIRONMENT_PATH]
          [NO_CMAKE_SYSTEM_PATH]
          [NO_CMAKE_INSTALL_PREFIX]
          [CMAKE_FIND_ROOT_PATH_BOTH |
           ONLY_CMAKE_FIND_ROOT_PATH |
           NO_CMAKE_FIND_ROOT_PATH]
          )
```

This command is used to find a full path to named file. A cache entry, or a normal variable if **NO_CACHE** is specified, named by **<VAR>** is created to store the result of this command. If the full path to a file is found the result is stored in the variable and the search will not be repeated unless the variable is cleared. If nothing is found, the result will be **<VAR>−NOTFOUND**.

Options include:

NAMES
Specify one or more possible names for the full path to a file.

When using this to specify names with and without a version suffix, we recommend specifying the unversioned name first so that locally−built packages can be found before those provided by distributions.

HINTS, PATHS
Specify directories to search in addition to the default locations. The **ENV var** sub−option reads paths from a system environment variable.

Changed in version 3.24: On **Windows** platform, it is possible to include registry queries as part of the directories, using a *dedicated syntax*. Such specifications will be ignored on all other platforms.

REGISTRY_VIEW
Added in version 3.24.

Specify which registry views must be queried. This option is only meaningful on **Windows** platforms and will be ignored on other ones. When not specified, the **TARGET** view is used when the *CMP0134* policy is **NEW**. Refer to *CMP0134* for the default view when the policy is **OLD**.

64 Query the 64−bit registry. On 32−bit Windows, it always returns the string **/REGISTRY−NOTFOUND**.

32 Query the 32−bit registry.

64_32 Query both views (**64** and **32**) and generate a path for each.

32_64 Query both views (**32** and **64**) and generate a path for each.

HOST Query the registry matching the architecture of the host: **64** on 64−bit Windows and **32** on 32−bit Windows.

TARGET
Query the registry matching the architecture specified by the *CMAKE_SIZEOF_VOID_P* variable. If not defined, fall back to **HOST** view.

BOTH Query both views (**32** and **64**). The order depends on the following rules: If the *CMAKE_SIZEOF_VOID_P* variable is defined, use the following view depending on the content of this variable:

- **8**: **64_32**
- **4**: **32_64**

If the *CMAKE_SIZEOF_VOID_P* variable is not defined, rely on the architecture of the host:

- 64−bit: **64_32**
- 32−bit: **32**

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

VALIDATOR
Added in version 3.25.

Specify a *function()* to be called for each candidate item found (a *macro()* cannot be provided, that will result in an error). Two arguments will be passed to the validator function: the name of a result variable, and the absolute path to the candidate item. The item will be accepted and the search will end unless the function sets the value in the result variable to false in the calling scope. The result variable will hold a true value when the validator function is entered.

```
function(my_check validator_result_var item)
  if(NOT item MATCHES ...)
```

```
    set(${validator_result_var} FALSE PARENT_SCOPE)
  endif()
endfunction()

find_file (result NAMES ... VALIDATOR my_check)
```

Note that if a cached result is used, the search is skipped and any **VALIDATOR** is ignored. The cached result is not required to pass the validation function.

DOC Specify the documentation string for the **<VAR>** cache entry.

NO_CACHE
Added in version 3.21.

The result of the search will be stored in a normal variable rather than a cache entry.

NOTE:
If the variable is already set before the call (as a normal or cache variable) then the search will not occur.

WARNING:
This option should be used with caution because it can greatly increase the cost of repeated configure steps.

REQUIRED
Added in version 3.18.

Stop processing with an error message if nothing is found, otherwise the search will be attempted again the next time find_file is invoked with the same variable.

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

1. If called from within a find module or any other script loaded by a call to *find_package(<PackageName>)*, search prefixes unique to the current package being found. See policy *CMP0074*.

   Added in version 3.12.

   Specifically, search paths specified by the following variables, in order:

   a. *<PackageName>_ROOT* CMake variable, where **<PackageName>** is the case−preserved package name.

   b. *<PACKAGENAME>_ROOT* CMake variable, where **<PACKAGENAME>** is the upper−cased package name. See policy *CMP0144*.

      Added in version 3.27.

   c. *<PackageName>_ROOT* environment variable, where **<PackageName>** is the case−preserved package name.

   d. *<PACKAGENAME>_ROOT* environment variable, where **<PACKAGENAME>** is the upper−cased package name. See policy *CMP0144*.

Added in version 3.27.

The package root variables are maintained as a stack, so if called from nested find modules or config packages, root paths from the parent's find module or config package will be searched after paths from the current module or package. In other words, the search order would be **<CurrentPackage>_ROOT**, **ENV{<CurrentPackage>_ROOT}**, **<ParentPackage>_ROOT**, **ENV{<ParentPackage>_ROOT}**, etc. This can be skipped if**NO_P ACKAGE_ROOT_PATH** is passed or by setting the *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* to **FALSE**.

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

2. Search paths specified in cmake−specific cache variables. These are intended to be used on the command line with a **−DVAR=value**. The values are interpreted as *semicolon−separated lists*. This can be skipped if **NO_CMAKE_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_PATH* to **FALSE**.

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

3. Search paths specified in cmake−specific environment variables. These are intended to be set in the user's shell configuration, and therefore use the host's native path separator (**;** on Windows and **:** on UNIX). This can be skipped if**NO_CMAKE_ENVIR ONMENT_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH* to **FALSE**.

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

4. Search the paths specified by the **HINTS** option. These should be paths computed by system introspection, such as a hint provided by the location of another item already found. Hard−coded guesses should be specified with the **PATHS** option.

5. Search the standard system environment variables. This can be skipped if**NO_SYSTEM_ENVIR ONMENT_PATH** is passed or by setting the *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH* to **FALSE**.

   - The directories in **INCLUDE** and **PATH**.

   On Windows hosts, CMake 3.3 through 3.27 searched additional paths: **<prefix>/include/<arch>** if *CMAKE_LIBRARY_ARCHITECTURE* is set, and **<prefix>/include** for each **<prefix>/[s]bin** in **PATH**, and **<entry>/include** for other entries in **PATH**. This behavior was removed by CMake 3.28.

6. Search cmake variables defined in the Platform files for the current system. The searching of **CMAKE_INSTALL_PREFIX** and **CMAKE_STAGING_PREFIX** can be skipped if **NO_CMAKE_INSTALL_PREFIX** is passed or by setting the *CMAKE_FIND_USE_INSTALL_PREFIX* to **FALSE**. All these locations can be skipped if **NO_CMAKE_SYSTEM_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH* to **FALSE**.

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

- *CMAKE_SYSTEM_INCLUDE_PATH*
- *CMAKE_SYSTEM_FRAMEWORK_PATH*

The platform paths that these variables contain are locations that typically include installed software. An example being **/usr/local** for UNIX based platforms.

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

The *CMAKE_IGNORE_PATH*, *CMAKE_IGNORE_PREFIX_PATH*, *CMAKE_SYSTEM_IGNORE_PATH* and *CMAKE_SYSTEM_IGNORE_PREFIX_PATH* variables can also cause some of the above locations to be ignored.

Added in version 3.16: Added **CMAKE_FIND_USE_<CATEGORY>_PATH** variables to globally disable various search locations.

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

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

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

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

CMAKE_FIND_ROOT_PATH_BOTH
Search in the order described above.

NO_CMAKE_FIND_ROOT_PATH
Do not use the *CMAKE_FIND_ROOT_PATH* variable.

ONLY_CMAKE_FIND_ROOT_PATH
Search only the re−rooted directories and directories below *CMAKE_STAGING_PREFIX*.

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

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

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

find_library
A short−hand signature is:

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

The general signature is:

```
find_library (
          <VAR>
          name | NAMES name1 [name2 ...] [NAMES_PER_DIR]
          [HINTS [path | ENV var]...]
          [PATHS [path | ENV var]...]
          [REGISTRY_VIEW (64|32|64_32|32_64|HOST|TARGET|BOTH)]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [VALIDATOR function]
          [DOC "cache documentation string"]
          [NO_CACHE]
          [REQUIRED]
          [NO_DEFAULT_PATH]
          [NO_PACKAGE_ROOT_PATH]
          [NO_CMAKE_PATH]
          [NO_CMAKE_ENVIRONMENT_PATH]
          [NO_SYSTEM_ENVIRONMENT_PATH]
          [NO_CMAKE_SYSTEM_PATH]
          [NO_CMAKE_INSTALL_PREFIX]
          [CMAKE_FIND_ROOT_PATH_BOTH |
           ONLY_CMAKE_FIND_ROOT_PATH |
           NO_CMAKE_FIND_ROOT_PATH]
         )
```

This command is used to find a library. A cache entry, or a normal variable if **NO_CACHE** is specified, named by **<VAR>** is created to store the result of this command. If the library is found the result is stored in the variable and the search will not be repeated unless the variable is cleared. If nothing is found, the result will be **<VAR>-NOTFOUND**.

Options include:

NAMES
Specify one or more possible names for the library.

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

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

Changed in version 3.24: On **Windows** platform, it is possible to include registry queries as part of the directories, using a *dedicated syntax*. Such specifications will be ignored on all other platforms.

REGISTRY_VIEW
Added in version 3.24.

Specify which registry views must be queried. This option is only meaningful on **Windows** platforms and will be ignored on other ones. When not specified, the **TARGET** view is used when the *CMP0134* policy is **NEW**. Refer to *CMP0134* for the default view when the policy is **OLD**.

64 Query the 64−bit registry. On 32−bit Windows, it always returns the string **/REGISTRY−NOTFOUND**.

32 Query the 32−bit registry.

64_32 Query both views (**64** and **32**) and generate a path for each.

32_64 Query both views (**32** and **64**) and generate a path for each.

HOST Query the registry matching the architecture of the host: **64** on 64−bit Windows and **32** on 32−bit Windows.

TARGET
Query the registry matching the architecture specified by the *CMAKE_SIZEOF_VOID_P* variable. If not defined, fall back to **HOST** view.

BOTH Query both views (**32** and **64**). The order depends on the following rules: If the *CMAKE_SIZEOF_VOID_P* variable is defined, use the following view depending on the content of this variable:

- **8**: **64_32**
- **4**: **32_64**

If the *CMAKE_SIZEOF_VOID_P* variable is not defined, rely on the architecture of the host:

- 64−bit: **64_32**
- 32−bit: **32**

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

VALIDATOR
Added in version 3.25.

Specify a *function()* to be called for each candidate item found (a *macro()* cannot be provided, that will result in an error). Two arguments will be passed to the validator function: the name of a result variable, and the absolute path to the candidate item. The item will be accepted and the search will end unless the function sets the value in the result variable to false in the calling scope. The result variable will hold a true value when the validator function is entered.

```
function(my_check validator_result_var item)
  if(NOT item MATCHES ...)
    set(${validator_result_var} FALSE PARENT_SCOPE)
  endif()
endfunction()

find_library (result NAMES ... VALIDATOR my_check)
```

Note that if a cached result is used, the search is skipped and any **VALIDATOR** is ignored. The cached result is not required to pass the validation function.

DOC Specify the documentation string for the **<VAR>** cache entry.

NO_CACHE
Added in version 3.21.

The result of the search will be stored in a normal variable rather than a cache entry.

NOTE:
If the variable is already set before the call (as a normal or cache variable) then the search will not occur.

WARNING:
This option should be used with caution because it can greatly increase the cost of repeated configure steps.

REQUIRED
Added in version 3.18.

Stop processing with an error message if nothing is found, otherwise the search will be attempted again the next time find_library is invoked with the same variable.

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

1. If called from within a find module or any other script loaded by a call to *find_package(<PackageName>)*, search prefixes unique to the current package being found. See policy *CMP0074*.

   Added in version 3.12.

   Specifically, search paths specified by the following variables, in order:

   a. *<PackageName>_ROOT* CMake variable, where **<PackageName>** is the case−preserved package name.

   b. *<PACKAGENAME>_ROOT* CMake variable, where **<PACKAGENAME>** is the upper−cased package name. See policy *CMP0144*.

      Added in version 3.27.

   c. *<PackageName>_ROOT* environment variable, where **<PackageName>** is the case−preserved package name.

   d. *<PACKAGENAME>_ROOT* environment variable, where **<PACKAGENAME>** is the upper−cased package name. See policy *CMP0144*.

      Added in version 3.27.

   The package root variables are maintained as a stack, so if called from nested find modules or config packages, root paths from the parent's find module or config package will be searched after paths from the current module or package. In other words, the search order would be **<CurrentPackage>_ROOT**, **ENV{<CurrentPackage>_ROOT}**, **<ParentPackage>_ROOT**, **ENV{<ParentPackage>_ROOT}**, etc. This can be skipped if**NO_P ACKAGE_ROOT_PATH** is passed or by setting the *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* to **FALSE**.

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

2. Search paths specified in cmake−specific cache variables. These are intended to be used on the command line with a **−DVAR=value**. The values are interpreted as *semicolon−separated lists*. This can be skipped if **NO_CMAKE_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_PATH* to

FALSE.

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

3. Search paths specified in cmake−specific environment variables. These are intended to be set in the user's shell configuration, and therefore use the host's native path separator (**;** on Windows and **:** on UNIX). This can be skipped if**NO_CMAKE_ENVIR ONMENT_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH* to **FALSE**.

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

4. Search the paths specified by the **HINTS** option. These should be paths computed by system introspection, such as a hint provided by the location of another item already found. Hard−coded guesses should be specified with the **PATHS** option.

5. Search the standard system environment variables. This can be skipped if**NO_SYSTEM_ENVIR ONMENT_PATH** is passed or by setting the *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH* to **FALSE**.

   - The directories in **LIB** and **PATH**.

   On Windows hosts, CMake 3.3 through 3.27 searched additional paths: **<prefix>/lib/<arch>** if *CMAKE_LIBRARY_ARCHITECTURE* is set, and **<prefix>/lib** for each **<prefix>/[s]bin** in **PATH**, and **<entry>/lib** for other entries in **PATH**. This behavior was removed by CMake 3.28.

6. Search cmake variables defined in the Platform files for the current system. The searching of **CMAKE_INSTALL_PREFIX** and **CMAKE_STAGING_PREFIX** can be skipped if **NO_CMAKE_INSTALL_PREFIX** is passed or by setting the *CMAKE_FIND_USE_INSTALL_PREFIX* to **FALSE**. All these locations can be skipped if **NO_CMAKE_SYSTEM_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH* to **FALSE**.

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

   The platform paths that these variables contain are locations that typically include installed software. An example being **/usr/local** for UNIX based platforms.

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

The *CMAKE_IGNORE_PATH*, *CMAKE_IGNORE_PREFIX_PATH*, *CMAKE_SYSTEM_IGNORE_PATH* and *CMAKE_SYSTEM_IGNORE_PREFIX_PATH* variables can also cause some of the above locations to be ignored.

Added in version 3.16: Added **CMAKE_FIND_USE_<CATEGORY>_PATH** variables to globally disable various search locations.

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

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

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

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

CMAKE_FIND_ROOT_PATH_BOTH
Search in the order described above.

NO_CMAKE_FIND_ROOT_PATH
Do not use the *CMAKE_FIND_ROOT_PATH* variable.

ONLY_CMAKE_FIND_ROOT_PATH
Search only the re−rooted directories and directories below *CMAKE_STAGING_PREFIX*.

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

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

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

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

Each library name given to the **NAMES** option is first considered as is, if it contains a library suffix, and then considered with platform−specific prefixes (e.g. **lib**) and suffixes (e.g. **.so**), as defined by the variables *CMAKE_FIND_LIBRARY_PREFIXES* and *CMAKE_FIND_LIBRARY_SUFFIXES*. Therefore one may specify library file names such as **libfoo.a** directly. This can be used to locate static libraries on UNIX−like systems.

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

Added in version 3.28: The library found can now be a **.xcframework** folder.

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

FIND_LIBRARY_USE_LIBX32_PATHS, and *FIND_LIBRARY_USE_LIB64_PATHS* global properties.

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

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

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

find_package

NOTE:

The *Using Dependencies Guide* provides a high−level introduction to this general topic. It provides a broader overview of where the **find_package()** command fits into the bigger picture, including its relationship to the *FetchContent* module. The guide is recommended pre−reading before moving on to the details below.

Find a package (usually provided by something external to the project), and load its package−specific details. Calls to this command can also be intercepted by*dependency pr oviders*.

Typical Usage

Most calls to **find_package()** typically have the following form:

```
find_package(<PackageName> [<version>] [REQUIRED] [COMPONENTS <components>...])
```

The <**PackageName**> is the only mandatory argument. The<**v ersion**> is often omitted, and **REQUIRED** should be given if the project cannot be configured successfully without the package. Some more complicated packages support components which can be selected with the **COMPONENTS** keyword, but most packages don't have that level of complexity.

The above is a reduced form of the *basic signature*. Where possible, projects should find packages using this form. This reduces complexity and maximizes the ways in which the package can be found or provided.

Understanding the *basic signature* should be enough for general usage of **find_package()**. Project maintainers who intend to provide a package configuration file should understand the bigger picture, as explained in *Full Signature* and all subsequent sections on this page.

Search Modes

The command has a few modes by which it searches for packages:

Module mode

In this mode, CMake searches for a file called **Find<PackageName>.cmake**, looking first in the locations listed in the *CMAKE_MODULE_PATH*, then among the *Find Modules* provided by the CMake installation. If the file is found, it is read and processed by CMake. It is responsible for finding the package, checking the version, and producing any needed messages. Some Find modules provide limited or no support for versioning; check the Find module's documentation.

The **Find<PackageName>.cmake** file is not typically provided by the package itself. Rather, it is normally provided by something external to the package, such as the operating system, CMake

itself, or even the project from which the **find_package()** command was called. Being externally provided, *Find Modules* tend to be heuristic in nature and are susceptible to becoming out−of−date. They typically search for certain libraries, files and other package artifacts.

Module mode is only supported by the *basic command signature*.

Config mode

In this mode, CMake searches for a file called **<lowercasePackageName>−config.cmake** or **<PackageName>Config.cmake**. It will also look for**<lo wercasePackageName>−config−version.cmake** or **<PackageName>ConfigVersion.cmake** if version details were specified (see *Config Mode Version Selection* for an explanation of how these separate version files are used).

NOTE:

If the experimental **CMAKE_EXPERIMENTAL_FIND_CPS_PACKAGES** is enabled, files named **<PackageName>.cps** and **<lowercasePackageName>.cps** are also considered. These files provide package information according to the *Common Package Specification* (CPS), which is more portable than CMake script. Aside from any explicitly noted exceptions, any references to "config files", "config mode", "package configuration files", and so forth refer equally to both CPS and CMake−script files. This functionality is a work in progress, and some features may be missing.

Search is implemented in a manner that will tend to prefer Common Package Specification files over CMake−script config files in most cases. Specifying **CONFIGS** suppresses consideration of CPS files.

In config mode, the command can be given a list of names to search for as package names. The locations where CMake searches for the config and version files is considerably more complicated than for Module mode (see *Config Mode Search Procedure*).

The config and version files are typically installed as part of the package, so they tend to be more reliable than Find modules. They usually contain direct knowledge of the package contents, so no searching or heuristics are needed within the config or version files themselves.

Config mode is supported by both the *basic* and *full* command signatures.

FetchContent redirection mode

Added in version 3.24: A call to **find_package()** can be redirected internally to a package provided by the *FetchContent* module. To the caller, the behavior will appear similar to Config mode, except that the search logic is by−passed and the component information is not used. See *FetchContent_Declare()* and *FetchContent_MakeAvailable()* for further details.

When not redirected to a package provided by *FetchContent*, the command arguments determine whether Module or Config mode is used. When the *basic signature* is used, the command searches in Module mode first. If the package is not found, the search falls back to Config mode. A user may set the *CMAKE_FIND_PACKAGE_PREFER_CONFIG* variable to true to reverse the priority and direct CMake to search using Config mode first before falling back to Module mode. The basic signature can also be forced to use only Module mode with a **MODULE** keyword. If the*full signatur e* is used, the command only searches in Config mode.

Basic Signature

```
find_package(<PackageName> [version] [EXACT] [QUIET] [MODULE]
             [REQUIRED] [[COMPONENTS] [components...]]
             [OPTIONAL_COMPONENTS components...]
             [REGISTRY_VIEW  (64|32|64_32|32_64|HOST|TARGET|BOTH)]
             [GLOBAL]
```

```
                [NO_POLICY_SCOPE]
                [BYPASS_PROVIDER])
```

The basic signature is supported by both Module and Config modes. The **MODULE** keyword implies that only Module mode can be used to find the package, with no fallback to Config mode.

Regardless of the mode used, a **<PackageName>_FOUND** variable will be set to indicate whether the package was found. When the package is found, package−specific information may be provided through other variables and *Imported Targets* documented by the package itself. The **QUIET** option disables informational messages, including those indicating that the package cannot be found if it is not **REQUIRED**. The **REQUIRED** option stops processing with an error message if the package cannot be found.

A package−specific list of required components may be listed after the **COMPONENTS** keyword. If any of these components are not able to be satisfied, the package overall is considered to be not found. If the **REQUIRED** option is also present, this is treated as a fatal error, otherwise execution still continues. As a form of shorthand, if the **REQUIRED** option is present, the **COMPONENTS** keyword can be omitted and the required components can be listed directly after **REQUIRED**.

Additional optional components may be listed after **OPTIONAL_COMPONENTS**. If these cannot be satisfied, the package overall can still be considered found, as long as all required components are satisfied.

The set of available components and their meaning are defined by the target package. For CMake−script package configuration files, it is formally up to the target package how to interpret the component information given to it, but it should follow the expectations stated above. For calls where no components are specified, there is no single expected behavior and target packages should clearly define what occurs in such cases. Common arrangements include assuming it should find all components, no components or some well−defined subset of the available components.

NOTE:

If the experimental **CMAKE_EXPERIMENTAL_FIND_CPS_PACKAGES** is enabled, CMake currently imports all available components if the located package configuration file is a Common Package Specification file. At this time, **COMPONENTS** and **OPTIONAL_COMPONENTS** have no effect when considering a CPS file.

Added in version 3.24: The **REGISTRY_VIEW** keyword specifies which registry views should be queried. This keyword is only meaningful on **Windows** platforms and will be ignored on all others. Formally, it is up to the target package how to interpret the registry view information given to it.

Added in version 3.24: Specifying the **GLOBAL** keyword will promote all imported targets to a global scope in the importing project. Alternatively, this functionality can be enabled by setting the *CMAKE_FIND_PACKAGE_TARGETS_GLOBAL* variable.

The **[version]** argument requests a version with which the package found should be compatible. There are two possible forms in which it may be specified:

- A single version with the format **major[.minor[.patch[.tweak]]]**, where each component is a numeric value.
- A version range with the format **versionMin...[<]versionMax** where **versionMin** and **versionMax** have the same format and constraints on components being integers as the single version. By default, both end points are included. By specifying **<**, the upper end point will be excluded. Version ranges are only supported with CMake 3.19 or later.

NOTE:

With the exception of CPS packages, version support is currently provided only on a package−by−package basis. When a version range is specified but the package is only designed to expect a single version, the package will ignore the upper end point of the range and only take the single version at the lower end of the range into account. Non−CPS packages that do support version ranges do so in a manner that is determined by the individual package. See the *Version Selection* section below for details and important caveats.

The **EXACT** option requests that the version be matched exactly. This option is incompatible with the specification of a version range.

If no **[version]** and/or component list is given to a recursive invocation inside a find−module, the corresponding arguments are forwarded automatically from the outer call (including the **EXACT** flag for **[version]**).

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

Added in version 3.24: The **BYPASS_PROVIDER** keyword is only allowed when **find_package()** is being called by a *dependency provider*. It can be used by providers to call the built−in **find_package()** implementation directly and prevent that call from being re−routed back to itself. Future versions of CMake may detect attempts to use this keyword from places other than a dependency provider and halt with a fatal error.

Full Signature

```
find_package(<PackageName> [version] [EXACT] [QUIET]
             [REQUIRED] [[COMPONENTS] [components...]]
             [OPTIONAL_COMPONENTS components...]
             [CONFIG|NO_MODULE]
             [GLOBAL]
             [NO_POLICY_SCOPE]
             [BYPASS_PROVIDER]
             [NAMES name1 [name2 ...]]
             [CONFIGS config1 [config2 ...]]
             [HINTS path1 [path2 ...]]
             [PATHS path1 [path2 ...]]
             [REGISTRY_VIEW  (64|32|64_32|32_64|HOST|TARGET|BOTH)]
             [PATH_SUFFIXES suffix1 [suffix2 ...]]
             [NO_DEFAULT_PATH]
             [NO_PACKAGE_ROOT_PATH]
             [NO_CMAKE_PATH]
             [NO_CMAKE_ENVIRONMENT_PATH]
             [NO_SYSTEM_ENVIRONMENT_PATH]
             [NO_CMAKE_PACKAGE_REGISTRY]
             [NO_CMAKE_BUILDS_PATH] # Deprecated; does nothing.
             [NO_CMAKE_SYSTEM_PATH]
             [NO_CMAKE_INSTALL_PREFIX]
             [NO_CMAKE_SYSTEM_PACKAGE_REGISTRY]
             [CMAKE_FIND_ROOT_PATH_BOTH |
              ONLY_CMAKE_FIND_ROOT_PATH |
              NO_CMAKE_FIND_ROOT_PATH])
```

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

Config mode search attempts to locate a configuration file provided by the package to be found. A cache

entry called **<PackageName>_DIR** is created to hold the directory containing the file. By default, the command searches for a package with the name **<PackageName>**. If the**N AMES** option is given, the names following it are used instead of **<PackageName>**. The names are also considered when determining whether to redirect the call to a package provided by *FetchContent*.

The command searches for a file called **<PackageName>Config.cmake** or **<lowercasePackageName>−config.cmake** for each name specified. A replacement set of possible configuration file names may be given using the **CONFIGS** option. The *Config Mode Search Procedure* is specified below. Once found, any *version constraint* is checked, and if satisfied, the configuration file is read and processed by CMake. Since the file is provided by the package it already knows the location of package contents. The full path to the configuration file is stored in the CMake variable **<PackageName>_CONFIG**.

NOTE:

If the experimental **CMAKE_EXPERIMENTAL_FIND_CPS_PACKAGES** is enabled, files named **<PackageName>.cps** and **<lowercasePackageName>.cps** are also considered, unless **CONFIGS** is given.

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

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

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

Config Mode Search Procedure

NOTE:

When Config mode is used, this search procedure is applied regardless of whether the *full* or *basic* signature was given.

Added in version 3.24: All calls to **find_package()** (even in Module mode) first look for a config package file in the *CMAKE_FIND_PACKAGE_REDIRECTS_DIR* directory. The*F etchContent* module, or even the project itself, may write files to that location to redirect **find_package()** calls to content already provided by the project. If no config package file is found in that location, the search proceeds with the logic described below.

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

Entry	Convention
<prefix>/<name>/cps/ [2]	W
<prefix>/<name>/*/cps/ [2]	W
<prefix>/cps/<name>/ [2]	W
<prefix>/cps/<name>/*/ [2]	W
<prefix>/cps/ [2]	W
<prefix>/	W
<prefix>/(cmake|CMake)/	W
<prefix>/<name>*/	W
<prefix>/<name>*/(cmake|CMake)/	W
<prefix>/<name>*/(cmake|CMake)/<name>*/ [1]	W
<prefix>/(lib/<arch>|lib*|share)/cps/<name>/ [2]	U
<prefix>/(lib/<arch>|lib*|share)/cps/<name>/*/ [2]	U
<prefix>/(lib/<arch>|lib*|share)/cps/ [2]	U
<prefix>/(lib/<arch>|lib*|share)/cmake/<name>*/	U
<prefix>/(lib/<arch>|lib*|share)/<name>*/	U
<prefix>/(lib/<arch>|lib*|share)/<name>*/(cmake|CMake)/	U
<prefix>/<name>*/(lib/<arch>|lib*|share)/cmake/<name>*/	W/U
<prefix>/<name>*/(lib/<arch>|lib*|share)/<name>*/	W/U
<prefix>/<name>*/(lib/<arch>|lib*|share)/<name>*/(cmake|CMake)/	W/U

[1] Added in version 3.25.

[2] Added in version 4.0.

On systems supporting macOS *FRAMEWORK* and *BUNDLE*, the following directories are searched for Frameworks or Application Bundles containing a configuration file:

Entry	Convention
<prefix>/<name>.framework/Versions/*/Resources/CPS/ [3]	A
<prefix>/<name>.framework/Resources/CPS/ [3]	A
<prefix>/<name>.framework/Resources/	A
<prefix>/<name>.framework/Resources/CMake/	A
<prefix>/<name>.framework/Versions/*/Resources/	A
<prefix>/<name>.framework/Versions/*/Resources/CMake/	A
<prefix>/<name>.app/Contents/Resources/CPS/ [3]	A
<prefix>/<name>.app/Contents/Resources/	A
<prefix>/<name>.app/Contents/Resources/CMake/	A

[3] Added in version 4.0.

When searching the above paths, **find_package** will only look for **.cps** files in search paths which contain **/cps/**, and will only look for **.cmake** files otherwise. (This only applies to the paths as specified and does not consider the contents of **<prefix>** or **<name>**.)

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

If at least one compiled language has been enabled, the architecture−specific **lib/<arch>** and **lib*** directories may be searched based on the compiler's target architecture, in the following order:

lib/<arch>
Searched if the *CMAKE_LIBRARY_ARCHITECTURE* variable is set.

lib64 Searched on 64 bit platforms (*CMAKE_SIZEOF_VOID_P* is 8) and the *FIND_LIBRARY_USE_LIB64_PATHS* property is set to **TRUE**.

lib32 Searched on 32 bit platforms (*CMAKE_SIZEOF_VOID_P* is 4) and the *FIND_LIBRARY_USE_LIB32_PATHS* property is set to **TRUE**.

libx32 Searched on platforms using the x32 ABI if the *FIND_LIBRARY_USE_LIBX32_PATHS* property is set to **TRUE**.

lib Always searched.

Changed in version 3.24: On **Windows** platform, it is possible to include registry queries as part of the directories specified through **HINTS** and **PATHS** keywords, using a *dedicated syntax*. Such specifications will be ignored on all other platforms.

Added in version 3.24: **REGISTRY_VIEW** can be specified to manage **Windows** registry queries specified as part of **PATHS** and **HINTS**.

Specify which registry views must be queried. This option is only meaningful on **Windows** platforms and

will be ignored on other ones. When not specified, the **TARGET** view is used when the *CMP0134* policy is **NEW**. Refer to *CMP0134* for the default view when the policy is **OLD**.

64 Query the 64−bit registry. On 32−bit Windows, it always returns the string **/REGISTRY−NOT-FOUND**.

32 Query the 32−bit registry.

64_32 Query both views (**64** and **32**) and generate a path for each.

32_64 Query both views (**32** and **64**) and generate a path for each.

HOST Query the registry matching the architecture of the host: **64** on 64−bit Windows and **32** on 32−bit Windows.

TARGET
Query the registry matching the architecture specified by the *CMAKE_SIZEOF_VOID_P* variable. If not defined, fall back to **HOST** view.

BOTH Query both views (**32** and **64**). The order depends on the following rules: If the *CMAKE_SIZEOF_VOID_P* variable is defined, use the following view depending on the content of this variable:

- **8**: **64_32**
- **4**: **32_64**

If the *CMAKE_SIZEOF_VOID_P* variable is not defined, rely on the architecture of the host:

- 64−bit: **64_32**
- 32−bit: **32**

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

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

WARNING:
Setting *CMAKE_FIND_FRAMEWORK* or *CMAKE_FIND_APPBUNDLE* to values other than **FIRST** (the default) will cause CMake to search for Common Package Specification files in an order that is different from the order set forth in the specification.

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

1. Search prefixes unique to the current **<PackageName>** being found. See policy *CMP0074*.

   Added in version 3.12.

   Specifically, search prefixes specified by the following variables, in order:

a. *<PackageName>_ROOT* CMake variable, where **<PackageName>** is the case−preserved package name.

b. *<PACKAGENAME>_ROOT* CMake variable, where **<PACKAGENAME>** is the upper−cased package name. See policy *CMP0144*.

   Added in version 3.27.

c. *<PackageName>_ROOT* environment variable, where **<PackageName>** is the case−preserved package name.

d. *<PACKAGENAME>_ROOT* environment variable, where **<PACKAGENAME>** is the upper−cased package name. See policy *CMP0144*.

   Added in version 3.27.

The package root variables are maintained as a stack so if called from within a find module, root paths from the parent's find module will also be searched after paths for the current package. This can be skipped if **NO_PACKAGE_ROOT_PATH** is passed or by setting the *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* to **FALSE**.

2. Search paths specified in CMake−specific cache variables. These are intended to be used on the command line with a *−DVAR=VALUE*. The values are interpreted as *semicolon−separated lists*. This can be skipped if **NO_CMAKE_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_PATH* to **FALSE**:

   - *CMAKE_PREFIX_PATH*
   - *CMAKE_FRAMEWORK_PATH*
   - *CMAKE_APPBUNDLE_PATH*

3. Search paths specified in CMake−specific environment variables. These are intended to be set in the user's shell configuration, and therefore use the host's native path separator (**;** on Windows and **:** on UNIX). This can be skipped if**NO_CMAKE_ENVIR ONMENT_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH* to **FALSE**:

   - **<PackageName>_DIR**
   - *CMAKE_PREFIX_PATH*
   - *CMAKE_FRAMEWORK_PATH*
   - *CMAKE_APPBUNDLE_PATH*

4. Search paths specified by the **HINTS** option. These should be paths computed by system introspection, such as a hint provided by the location of another item already found. Hard−coded guesses should be specified with the **PATHS** option.

5. Search the standard system environment variables. This can be skipped if**NO_SYSTEM_ENVIR ONMENT_PATH** is passed or by setting the *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH* to **FALSE**. Path entries ending in **/bin** or **/sbin** are automatically converted to their parent directories:

   - **PATH**

6. Search paths stored in the CMake *User Package Registry*. This can be skipped if **NO_CMAKE_PACKAGE_REGISTRY** is passed or by setting the variable *CMAKE_FIND_USE_PACKAGE_REGISTRY* to **FALSE** or the deprecated variable *CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY* to **TRUE**.

   See the *cmake−packages(7)* manual for details on the user package registry.

7. Search CMake variables defined in the Platform files for the current system. The searching of *CMAKE_INSTALL_PREFIX* and *CMAKE_STAGING_PREFIX* can be skipped if **NO_CMAKE_INSTALL_PREFIX** is passed or by setting the *CMAKE_FIND_USE_INSTALL_PREFIX* to **FALSE**. All these locations can be skipped if **NO_CMAKE_SYSTEM_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH* to **FALSE**:

   - *CMAKE_SYSTEM_PREFIX_PATH*
   - *CMAKE_SYSTEM_FRAMEWORK_PATH*
   - *CMAKE_SYSTEM_APPBUNDLE_PATH*

   The platform paths that these variables contain are locations that typically include installed software. An example being **/usr/local** for UNIX based platforms.

8. Search paths stored in the CMake *System Package Registry*. This can be skipped if **NO_CMAKE_SYSTEM_PACKAGE_REGISTRY** is passed or by setting the *CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY* variable to **FALSE** or the deprecated variable *CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY* to **TRUE**.

   See the *cmake−packages(7)* manual for details on the system package registry.

9. Search paths specified by the **PATHS** option. These are typically hard−coded guesses.

The *CMAKE_IGNORE_PATH*, *CMAKE_IGNORE_PREFIX_PATH*, *CMAKE_SYSTEM_IGNORE_PATH* and *CMAKE_SYSTEM_IGNORE_PREFIX_PATH* variables can also cause some of the above locations to be ignored.

Paths are searched in the order described above. The first viable package configuration file found is used, even if a newer version of the package resides later in the list of search paths.

For search paths which contain glob expressions (*****), the order in which directories matching the glob are searched is unspecified unless the *CMAKE_FIND_PACKAGE_SORT_ORDER* variable is set. This variable, along with the *CMAKE_FIND_PACKAGE_SORT_DIRECTION* variable, determines the order in which CMake considers glob matches. For example, if the file system contains the package configuration files

```
<prefix>/example−1.2/example−config.cmake
<prefix>/example−1.10/example−config.cmake
<prefix>/share/example−2.0/example−config.cmake
```

it is unspecified (when the aforementioned variables are unset) whether **find_package(example)** will find **example−1.2** or **example−1.10** (assuming that both are viable), but **find_package** will *not* find **example−2.0**, because one of the other two will be found first.

To control the order in which **find_package** searches directories that match a glob expression, use *CMAKE_FIND_PACKAGE_SORT_ORDER* and *CMAKE_FIND_PACKAGE_SORT_DIRECTION*. For instance, to cause the above example to select **example−1.10**, one can set

```
set(CMAKE_FIND_PACKAGE_SORT_ORDER NATURAL)
set(CMAKE_FIND_PACKAGE_SORT_DIRECTION DEC)
```

before calling **find_package**.

Added in version 3.16: Added the **CMAKE_FIND_USE_<CATEGORY>** variables to globally disable various search locations.

Changed in version 4.0: The variables *CMAKE_FIND_PACKAGE_SORT_ORDER* and *CMAKE_FIND_PACKAGE_SORT_DIRECTION* now also control the order in which **find_package** searches directories matching the glob expression in the search paths **<prefix>/<name>.framework/Versions/*/Resources/** and **<prefix>/<name>.framework/Versions/*/Resources/CMake**. In previous versions of CMake, this order was unspecified.

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

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

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

CMAKE_FIND_ROOT_PATH_BOTH
Search in the order described above.

NO_CMAKE_FIND_ROOT_PATH
Do not use the *CMAKE_FIND_ROOT_PATH* variable.

ONLY_CMAKE_FIND_ROOT_PATH
Search only the re−rooted directories and directories below *CMAKE_STAGING_PREFIX*.

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

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

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

By default the value stored in the result variable will be the path at which the file is found. The *CMAKE_FIND_PACKAGE_RESOLVE_SYMLINKS* variable may be set to **TRUE** before calling **find_package** in order to resolve symbolic links and store the real path to the file.

Every non−REQUIRED **find_package** call can be disabled or made REQUIRED:

- Setting the *CMAKE_DISABLE_FIND_PACKAGE_<PackageName>* variable to **TRUE** disables the package. This also disables redirection to a package provided by *FetchContent*.
- Setting the *CMAKE_REQUIRE_FIND_PACKAGE_<PackageName>* variable to **TRUE** makes the package REQUIRED.

Setting both variables to **TRUE** simultaneously is an error.

Config Mode Version Selection

NOTE:
When Config mode is used, this version selection process is applied regardless of whether the *full* or *basic* signature was given.

When the **[version]** argument is given, Config mode will only find a version of the package that claims compatibility with the requested version (see *format specification*). If the**EXA CT** option is given, only a version of the package claiming an exact match of the requested version may be found. CMake does not establish any convention for the meaning of version numbers.

CMake−script

For CMake−script package configuration files, package version numbers are checked by "version" files provided by the packages themselves or by *FetchContent*. For a candidate package configuration file **<config−file>.cmake** the corresponding version file is located next to it and named either **<config−file>−version.cmake** or **<config−file>Version.cmake**. If no such version file is available then the configuration file is assumed to not be compatible with any requested version. A basic version file containing generic version matching code can be created using the *CMakePackageConfigHelpers* module. When a version file is found it is loaded to check the requested version number. The version file is loaded in a nested scope in which the following variables have been defined:

PACKAGE_FIND_NAME
The **<PackageName>**

PACKAGE_FIND_VERSION
Full requested version string

PACKAGE_FIND_VERSION_MAJOR
Major version if requested, else 0

PACKAGE_FIND_VERSION_MINOR
Minor version if requested, else 0

PACKAGE_FIND_VERSION_PATCH
Patch version if requested, else 0

PACKAGE_FIND_VERSION_TWEAK
Tweak version if requested, else 0

PACKAGE_FIND_VERSION_COUNT
Number of version components, 0 to 4

When a version range is specified, the above version variables will hold values based on the lower end of the version range. This is to preserve compatibility with packages that have not been implemented to expect version ranges. In addition, the version range will be described by the following variables:

PACKAGE_FIND_VERSION_RANGE
Full requested version range string

PACKAGE_FIND_VERSION_RANGE_MIN
This specifies whether the lower end point of the version range should be included or excluded. Currently, the only supported value for this variable is **INCLUDE**.

PACKAGE_FIND_VERSION_RANGE_MAX
This specifies whether the upper end point of the version range should be included or excluded. The supported values for this variable are **INCLUDE** and **EXCLUDE**.

PACKAGE_FIND_VERSION_MIN
Full requested version string of the lower end point of the range

PACKAGE_FIND_VERSION_MIN_MAJOR
Major version of the lower end point if requested, else 0

PACKAGE_FIND_VERSION_MIN_MINOR
Minor version of the lower end point if requested, else 0

PACKAGE_FIND_VERSION_MIN_PATCH
Patch version of the lower end point if requested, else 0

PACKAGE_FIND_VERSION_MIN_TWEAK
Tweak version of the lower end point if requested, else 0

PACKAGE_FIND_VERSION_MIN_COUNT
Number of version components of the lower end point, 0 to 4

PACKAGE_FIND_VERSION_MAX
Full requested version string of the upper end point of the range

PACKAGE_FIND_VERSION_MAX_MAJOR
Major version of the upper end point if requested, else 0

PACKAGE_FIND_VERSION_MAX_MINOR
Minor version of the upper end point if requested, else 0

PACKAGE_FIND_VERSION_MAX_PATCH
Patch version of the upper end point if requested, else 0

PACKAGE_FIND_VERSION_MAX_TWEAK
Tweak version of the upper end point if requested, else 0

PACKAGE_FIND_VERSION_MAX_COUNT
Number of version components of the upper end point, 0 to 4

Regardless of whether a single version or a version range is specified, the variable **PACKAGE_FIND_VERSION_COMPLETE** will be defined and will hold the full requested version string as specified.

The version file checks whether it satisfies the requested version and sets these variables:

PACKAGE_VERSION
Full provided version string

PACKAGE_VERSION_EXACT
True if version is exact match

PACKAGE_VERSION_COMPATIBLE
True if version is compatible

PACKAGE_VERSION_UNSUITABLE
True if unsuitable as any version

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

<PackageName>_VERSION
Full provided version string

<PackageName>_VERSION_MAJOR
Major version if provided, else 0

<PackageName>_VERSION_MINOR
Minor version if provided, else 0

<PackageName>_VERSION_PATCH
Patch version if provided, else 0

<PackageName>_VERSION_TWEAK
Tweak version if provided, else 0

<PackageName>_VERSION_COUNT
Number of version components, 0 to 4

and the corresponding package configuration file is loaded.

NOTE:

While the exact behavior of version matching is determined by the individual package, many packages use *write_basic_package_version_file()* to supply this logic. The version check scripts this produces have some notable caveats with respect to version ranges:

- The upper end of a version range acts as a hard limit on what versions will be accepted. Thus, while a request for version **1.4.0** might be satisfied by a package whose version is **1.6.0** and which advertises 'same major version' compatibility, the same package will be rejected if the requested version range is **1.4.0...1.5.0**.
- Both ends of the version range must match the package's advertised compatibility level. For example, if a package advertises 'same major and minor version' compatibility, requesting the version range **1.4.0...<1.5.5** or **1.4.0...1.5.0** will result in that package being rejected, even if the package version is **1.4.1**.

As a result, it is not possible to use a version range to extend the range of compatible package versions that will be accepted.

Common Package Specification

For Common Package Specification package configuration files, package version numbers are checked by CMake according to the set of recognized version schemas. At present, the following schemas are recognized:

simple Version numbers are a tuple of integers followed by an optional trailing segment which is ignored with respect to version comparisons.

custom The mechanism for interpreting version numbers is unspecified. The version strings must match exactly for the package to be accepted.

Refer to *version_schema* for a more detailed explanation of each schema and how comparisons for each are performed. Note that the specification may include schemas that are not supported by CMake.

In addition to the package's **version**, CPS allows packages to optionally specify a *compat_version*, which is the oldest version for which the package provides compatibility. That is, the package warrants that a consumer expecting the **compat_version** should be able to use the package, even if the package's actual version is newer. If not specified, the**compat_v ersion** is implicitly equal to the package version, i.e. no backwards compatibility is provided.

When a package uses a recognized schema, CMake will determine the package's acceptability according to the following rules:

- If **EXACT** was specified, or if the package does not supply a **compat_version**, the package's **version** must equal the requested version.
- Otherwise:
  - The package's **version** must be greater than or equal to the requested (minimum) version, and
  - the package's **compat_version** must be less than or equal to the requested (minimum) version, and
  - if a requested maximum version was given, it must be greater than (or equal to, depending on whether the maximum version is specified as inclusive or exclusive) the package's **version**.

NOTE:

This implementation of range matching was chosen in order to most closely match the behavior of *write_basic_package_version_file()*, albeit without the case where an overly broad range matches nothing.

For packages using the **simple** version schema, if the version is acceptable, the following variables are set:

<PackageName>_VERSION
Full provided version string

<PackageName>_VERSION_MAJOR
Major version if provided, else 0

<PackageName>_VERSION_MINOR
Minor version if provided, else 0

<PackageName>_VERSION_PATCH
Patch version if provided, else 0

<PackageName>_VERSION_TWEAK
Tweak version if provided, else 0

<PackageName>_VERSION_COUNT
Number of version components, non−negative

Package File Interface Variables

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

CMAKE_FIND_PACKAGE_NAME
The **<PackageName>** which is searched for

<PackageName>_FIND_REQUIRED
True if **REQUIRED** option was given

<PackageName>_FIND_QUIETLY
True if **QUIET** option was given

<PackageName>_FIND_REGISTRY_VIEW
The requested view if **REGISTRY_VIEW** option was given

<PackageName>_FIND_VERSION
Full requested version string

<PackageName>_FIND_VERSION_MAJOR
Major version if requested, else 0

<PackageName>_FIND_VERSION_MINOR
Minor version if requested, else 0

<PackageName>_FIND_VERSION_PATCH
Patch version if requested, else 0

<PackageName>_FIND_VERSION_TWEAK
Tweak version if requested, else 0

<PackageName>_FIND_VERSION_COUNT
Number of version components, 0 to 4

<PackageName>_FIND_VERSION_EXACT
True if **EXACT** option was given

<PackageName>_FIND_COMPONENTS
List of specified components (required and optional)

<PackageName>_FIND_REQUIRED_<c>
True if component **<c>** is required, false if component **<c>** is optional

When a version range is specified, the above version variables will hold values based on the lower end of the version range. This is to preserve compatibility with packages that have not been implemented to expect version ranges. In addition, the version range will be described by the following variables:

<PackageName>_FIND_VERSION_RANGE
Full requested version range string

<PackageName>_FIND_VERSION_RANGE_MIN
This specifies whether the lower end point of the version range is included or excluded. Currently, **INCLUDE** is the only supported value.

<PackageName>_FIND_VERSION_RANGE_MAX
This specifies whether the upper end point of the version range is included or excluded. The possible values for this variable are **INCLUDE** or **EXCLUDE**.

<PackageName>_FIND_VERSION_MIN
Full requested version string of the lower end point of the range

<PackageName>_FIND_VERSION_MIN_MAJOR
Major version of the lower end point if requested, else 0

<PackageName>_FIND_VERSION_MIN_MINOR
Minor version of the lower end point if requested, else 0

<PackageName>_FIND_VERSION_MIN_PATCH
Patch version of the lower end point if requested, else 0

<PackageName>_FIND_VERSION_MIN_TWEAK
Tweak version of the lower end point if requested, else 0

<PackageName>_FIND_VERSION_MIN_COUNT
Number of version components of the lower end point, 0 to 4

<PackageName>_FIND_VERSION_MAX
Full requested version string of the upper end point of the range

<PackageName>_FIND_VERSION_MAX_MAJOR
Major version of the upper end point if requested, else 0

<PackageName>_FIND_VERSION_MAX_MINOR
Minor version of the upper end point if requested, else 0

<PackageName>_FIND_VERSION_MAX_PATCH
Patch version of the upper end point if requested, else 0

<PackageName>_FIND_VERSION_MAX_TWEAK
Tweak version of the upper end point if requested, else 0

<PackageName>_FIND_VERSION_MAX_COUNT
Number of version components of the upper end point, 0 to 4

Regardless of whether a single version or a version range is specified, the variable **<PackageName>_FIND_VERSION_COMPLETE** will be defined and will hold the full requested version string as specified.

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

find_path
A short−hand signature is:

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

The general signature is:

```
find_path (
          <VAR>
          name | NAMES name1 [name2 ...]
          [HINTS [path | ENV var]...]
          [PATHS [path | ENV var]...]
          [REGISTRY_VIEW (64|32|64_32|32_64|HOST|TARGET|BOTH)]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [VALIDATOR function]
          [DOC "cache documentation string"]
          [NO_CACHE]
          [REQUIRED]
          [NO_DEFAULT_PATH]
          [NO_PACKAGE_ROOT_PATH]
          [NO_CMAKE_PATH]
          [NO_CMAKE_ENVIRONMENT_PATH]
          [NO_SYSTEM_ENVIRONMENT_PATH]
          [NO_CMAKE_SYSTEM_PATH]
          [NO_CMAKE_INSTALL_PREFIX]
          [CMAKE_FIND_ROOT_PATH_BOTH |
           ONLY_CMAKE_FIND_ROOT_PATH |
           NO_CMAKE_FIND_ROOT_PATH]
         )
```

This command is used to find a directory containing the named file. A cache entry, or a normal variable if **NO_CACHE** is specified, named by **<VAR>** is created to store the result of this command. If the file in a directory is found the result is stored in the variable and the search will not be repeated unless the variable is cleared. If nothing is found, the result will be **<VAR>-NOTFOUND**.

Options include:

NAMES
Specify one or more possible names for the file in a directory.

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

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

Changed in version 3.24: On **Windows** platform, it is possible to include registry queries as part of the directories, using a *dedicated syntax*. Such specifications will be ignored on all other platforms.

REGISTRY_VIEW
Added in version 3.24.

Specify which registry views must be queried. This option is only meaningful on **Windows** platforms and will be ignored on other ones. When not specified, the **TARGET** view is used when the *CMP0134* policy is **NEW**. Refer to *CMP0134* for the default view when the policy is **OLD**.

64 Query the 64-bit registry. On 32-bit Windows, it always returns the string **/REGISTRY-NOTFOUND**.

32 Query the 32−bit registry.

64_32 Query both views (**64** and **32**) and generate a path for each.

32_64 Query both views (**32** and **64**) and generate a path for each.

HOST Query the registry matching the architecture of the host: **64** on 64−bit Windows and **32** on 32−bit Windows.

TARGET
Query the registry matching the architecture specified by the *CMAKE_SIZEOF_VOID_P* variable. If not defined, fall back to **HOST** view.

BOTH Query both views (**32** and **64**). The order depends on the following rules: If the *CMAKE_SIZEOF_VOID_P* variable is defined, use the following view depending on the content of this variable:

- **8**: **64_32**
- **4**: **32_64**

If the *CMAKE_SIZEOF_VOID_P* variable is not defined, rely on the architecture of the host:

- 64−bit: **64_32**
- 32−bit: **32**

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

VALIDATOR
Added in version 3.25.

Specify a *function()* to be called for each candidate item found (a *macro()* cannot be provided, that will result in an error). Two arguments will be passed to the validator function: the name of a result variable, and the absolute path to the candidate item. The item will be accepted and the search will end unless the function sets the value in the result variable to false in the calling scope. The result variable will hold a true value when the validator function is entered.

```
function(my_check validator_result_var item)
  if(NOT item MATCHES ...)
    set(${validator_result_var} FALSE PARENT_SCOPE)
  endif()
endfunction()

find_path (result NAMES ... VALIDATOR my_check)
```

Note that if a cached result is used, the search is skipped and any **VALIDATOR** is ignored. The cached result is not required to pass the validation function.

DOC Specify the documentation string for the **<VAR>** cache entry.

NO_CACHE
Added in version 3.21.

The result of the search will be stored in a normal variable rather than a cache entry.

NOTE:
If the variable is already set before the call (as a normal or cache variable) then the search will

not occur.

WARNING:
This option should be used with caution because it can greatly increase the cost of repeated configure steps.

REQUIRED
Added in version 3.18.

Stop processing with an error message if nothing is found, otherwise the search will be attempted again the next time find_path is invoked with the same variable.

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

1. If called from within a find module or any other script loaded by a call to *find_package(<PackageName>)*, search prefixes unique to the current package being found. See policy *CMP0074*.

   Added in version 3.12.

   Specifically, search paths specified by the following variables, in order:

   a. *<PackageName>_ROOT* CMake variable, where **<PackageName>** is the case−preserved package name.

   b. *<PACKAGENAME>_ROOT* CMake variable, where **<PACKAGENAME>** is the upper−cased package name. See policy *CMP0144*.

      Added in version 3.27.

   c. *<PackageName>_ROOT* environment variable, where **<PackageName>** is the case−preserved package name.

   d. *<PACKAGENAME>_ROOT* environment variable, where **<PACKAGENAME>** is the upper−cased package name. See policy *CMP0144*.

      Added in version 3.27.

   The package root variables are maintained as a stack, so if called from nested find modules or config packages, root paths from the parent's find module or config package will be searched after paths from the current module or package. In other words, the search order would be **<CurrentPackage>_ROOT**, **ENV{<CurrentPackage>_ROOT}**, **<ParentPackage>_ROOT**, **ENV{<ParentPackage>_ROOT}**, etc. This can be skipped if**NO_P ACKAGE_ROOT_PATH** is passed or by setting the *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* to **FALSE**.

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

2. Search paths specified in cmake−specific cache variables. These are intended to be used on the command line with a **−DVAR=value**. The values are interpreted as *semicolon−separated lists*. This can be skipped if **NO_CMAKE_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_PATH* to **FALSE**.

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

3. Search paths specified in cmake−specific environment variables. These are intended to be set in the user's shell configuration, and therefore use the host's native path separator (**;** on Windows and **:** on UNIX). This can be skipped if**NO_CMAKE_ENVIR ONMENT_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH* to **FALSE**.
   - **<prefix>/include/<arch>** if *CMAKE_LIBRARY_ARCHITECTURE* is set, and **<prefix>/include** for each **<prefix>** in *CMAKE_PREFIX_PATH*
   - *CMAKE_INCLUDE_PATH*
   - *CMAKE_FRAMEWORK_PATH*
4. Search the paths specified by the **HINTS** option. These should be paths computed by system introspection, such as a hint provided by the location of another item already found. Hard−coded guesses should be specified with the **PATHS** option.
5. Search the standard system environment variables. This can be skipped if**NO_SYSTEM_ENVIR ONMENT_PATH** is passed or by setting the *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH* to **FALSE**.
   - The directories in **INCLUDE** and **PATH**.

   On Windows hosts, CMake 3.3 through 3.27 searched additional paths: **<prefix>/include/<arch>** if *CMAKE_LIBRARY_ARCHITECTURE* is set, and **<prefix>/include** for each **<prefix>/[s]bin** in **PATH**, and **<entry>/include** for other entries in **PATH**. This behavior was removed by CMake 3.28.
6. Search cmake variables defined in the Platform files for the current system. The searching of **CMAKE_INSTALL_PREFIX** and **CMAKE_STAGING_PREFIX** can be skipped if **NO_CMAKE_INSTALL_PREFIX** is passed or by setting the *CMAKE_FIND_USE_INSTALL_PREFIX* to **FALSE**. All these locations can be skipped if **NO_CMAKE_SYSTEM_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH* to **FALSE**.
   - **<prefix>/include/<arch>** if *CMAKE_LIBRARY_ARCHITECTURE* is set, and **<prefix>/include** for each **<prefix>** in *CMAKE_SYSTEM_PREFIX_PATH*
   - *CMAKE_SYSTEM_INCLUDE_PATH*
   - *CMAKE_SYSTEM_FRAMEWORK_PATH*

   The platform paths that these variables contain are locations that typically include installed software. An example being **/usr/local** for UNIX based platforms.
7. Search the paths specified by the PATHS option or in the short−hand version of the command. These are typically hard−coded guesses.

The *CMAKE_IGNORE_PATH*, *CMAKE_IGNORE_PREFIX_PATH*, *CMAKE_SYSTEM_IGNORE_PATH* and *CMAKE_SYSTEM_IGNORE_PREFIX_PATH* variables can also cause some of the above locations to be ignored.

Added in version 3.16: Added **CMAKE_FIND_USE_<CATEGORY>_PATH** variables to globally disable various search locations.

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

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

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

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

CMAKE_FIND_ROOT_PATH_BOTH
Search in the order described above.

NO_CMAKE_FIND_ROOT_PATH
Do not use the *CMAKE_FIND_ROOT_PATH* variable.

ONLY_CMAKE_FIND_ROOT_PATH
Search only the re−rooted directories and directories below *CMAKE_STAGING_PREFIX*.

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

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

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

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

find_program
A short−hand signature is:

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

The general signature is:

```
find_program (
          <VAR>
          name | NAMES name1 [name2 ...] [NAMES_PER_DIR]
          [HINTS [path | ENV var]...]
          [PATHS [path | ENV var]...]
          [REGISTRY_VIEW (64|32|64_32|32_64|HOST|TARGET|BOTH)]
          [PATH_SUFFIXES suffix1 [suffix2 ...]]
          [VALIDATOR function]
          [DOC "cache documentation string"]
          [NO_CACHE]
          [REQUIRED]
          [NO_DEFAULT_PATH]
          [NO_PACKAGE_ROOT_PATH]
```

```
 [NO_CMAKE_PATH]
 [NO_CMAKE_ENVIRONMENT_PATH]
 [NO_SYSTEM_ENVIRONMENT_PATH]
 [NO_CMAKE_SYSTEM_PATH]
 [NO_CMAKE_INSTALL_PREFIX]
 [CMAKE_FIND_ROOT_PATH_BOTH |
  ONLY_CMAKE_FIND_ROOT_PATH |
  NO_CMAKE_FIND_ROOT_PATH]
)
```

This command is used to find a program. A cache entry, or a normal variable if **NO_CACHE** is specified, named by **<VAR>** is created to store the result of this command. If the program is found the result is stored in the variable and the search will not be repeated unless the variable is cleared. If nothing is found, the result will be **<VAR>−NOTFOUND**.

Options include:

NAMES
Specify one or more possible names for the program.

When using this to specify names with and without a version suffix, we recommend specifying the unversioned name first so that locally−built packages can be found before those provided by distributions.

HINTS, PATHS
Specify directories to search in addition to the default locations. The **ENV var** sub−option reads paths from a system environment variable.

Changed in version 3.24: On **Windows** platform, it is possible to include registry queries as part of the directories, using a *dedicated syntax*. Such specifications will be ignored on all other platforms.

REGISTRY_VIEW
Added in version 3.24.

Specify which registry views must be queried. This option is only meaningful on **Windows** platforms and will be ignored on other ones. When not specified, the **BOTH** view is used when the *CMP0134* policy is **NEW**. Refer to *CMP0134* for the default view when the policy is **OLD**.

64 Query the 64−bit registry. On 32−bit Windows, it always returns the string **/REGISTRY−NOTFOUND**.

32 Query the 32−bit registry.

64_32 Query both views (**64** and **32**) and generate a path for each.

32_64 Query both views (**32** and **64**) and generate a path for each.

HOST Query the registry matching the architecture of the host: **64** on 64−bit Windows and **32** on 32−bit Windows.

TARGET
Query the registry matching the architecture specified by the *CMAKE_SIZEOF_VOID_P* variable. If not defined, fall back to **HOST** view.

BOTH Query both views (**32** and **64**). The order depends on the following rules: If the *CMAKE_SIZEOF_VOID_P* variable is defined, use the following view depending on the content of this variable:

- **8**: **64_32**
- **4**: **32_64**

If the *CMAKE_SIZEOF_VOID_P* variable is not defined, rely on the architecture of the host:

- 64−bit: **64_32**
- 32−bit: **32**

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

VALIDATOR
Added in version 3.25.

Specify a *function()* to be called for each candidate item found (a *macro()* cannot be provided, that will result in an error). Two arguments will be passed to the validator function: the name of a result variable, and the absolute path to the candidate item. The item will be accepted and the search will end unless the function sets the value in the result variable to false in the calling scope. The result variable will hold a true value when the validator function is entered.

```
function(my_check validator_result_var item)
  if(NOT item MATCHES ...)
    set(${validator_result_var} FALSE PARENT_SCOPE)
  endif()
endfunction()

find_program (result NAMES ... VALIDATOR my_check)
```

Note that if a cached result is used, the search is skipped and any **VALIDATOR** is ignored. The cached result is not required to pass the validation function.

DOC Specify the documentation string for the **<VAR>** cache entry.

NO_CACHE
Added in version 3.21.

The result of the search will be stored in a normal variable rather than a cache entry.

NOTE:
If the variable is already set before the call (as a normal or cache variable) then the search will not occur.

WARNING:
This option should be used with caution because it can greatly increase the cost of repeated configure steps.

REQUIRED
Added in version 3.18.

Stop processing with an error message if nothing is found, otherwise the search will be attempted again the next time find_program is invoked with the same variable.

If **NO_DEFAULT_PATH** is specified, then no additional paths are added to the search. If

NO_DEFAULT_PATH is not specified, the search process is as follows:

1. If called from within a find module or any other script loaded by a call to *find_package(<PackageName>)*, search prefixes unique to the current package being found. See policy *CMP0074*.

   Added in version 3.12.

   Specifically, search paths specified by the following variables, in order:

   a. *<PackageName>_ROOT* CMake variable, where **<PackageName>** is the case−preserved package name.

   b. *<PACKAGENAME>_ROOT* CMake variable, where **<PACKAGENAME>** is the upper−cased package name. See policy *CMP0144*.

      Added in version 3.27.

   c. *<PackageName>_ROOT* environment variable, where **<PackageName>** is the case−preserved package name.

   d. *<PACKAGENAME>_ROOT* environment variable, where **<PACKAGENAME>** is the upper−cased package name. See policy *CMP0144*.

      Added in version 3.27.

   The package root variables are maintained as a stack, so if called from nested find modules or config packages, root paths from the parent's find module or config package will be searched after paths from the current module or package. In other words, the search order would be **<CurrentPackage>_ROOT**, **ENV{<CurrentPackage>_ROOT}**, **<ParentPackage>_ROOT**, **ENV{<ParentPackage>_ROOT}**, etc. This can be skipped if**NO_P ACKAGE_ROOT_PATH** is passed or by setting the *CMAKE_FIND_USE_PACKAGE_ROOT_PATH* to **FALSE**.

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

2. Search paths specified in cmake−specific cache variables. These are intended to be used on the command line with a **−DVAR=value**. The values are interpreted as *semicolon−separated lists*. This can be skipped if **NO_CMAKE_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_PATH* to **FALSE**.

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

3. Search paths specified in cmake−specific environment variables. These are intended to be set in the user's shell configuration, and therefore use the host's native path separator (**;** on Windows and **:** on UNIX). This can be skipped if**NO_CMAKE_ENVIR ONMENT_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH* to **FALSE**.

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

4. Search the paths specified by the **HINTS** option. These should be paths computed by system introspection, such as a hint provided by the location of another item already found. Hard−coded guesses should be specified with the **PATHS** option.
5. Search the standard system environment variables. This can be skipped if**NO_SYSTEM_ENVIR ONMENT_PATH** is passed or by setting the *CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH* to **FALSE**.
   - The directories in **PATH** itself.
6. Search cmake variables defined in the Platform files for the current system. The searching of **CMAKE_INSTALL_PREFIX** and **CMAKE_STAGING_PREFIX** can be skipped if **NO_CMAKE_INSTALL_PREFIX** is passed or by setting the *CMAKE_FIND_USE_INSTALL_PREFIX* to **FALSE**. All these locations can be skipped if **NO_CMAKE_SYSTEM_PATH** is passed or by setting the *CMAKE_FIND_USE_CMAKE_SYSTEM_PATH* to **FALSE**.
   - **<prefix>/[s]bin** for each **<prefix>** in *CMAKE_SYSTEM_PREFIX_PATH*
   - *CMAKE_SYSTEM_PROGRAM_PATH*
   - *CMAKE_SYSTEM_APPBUNDLE_PATH*

   The platform paths that these variables contain are locations that typically include installed software. An example being **/usr/local** for UNIX based platforms.
7. Search the paths specified by the PATHS option or in the short−hand version of the command. These are typically hard−coded guesses.

The *CMAKE_IGNORE_PATH*, *CMAKE_IGNORE_PREFIX_PATH*, *CMAKE_SYSTEM_IGNORE_PATH* and *CMAKE_SYSTEM_IGNORE_PREFIX_PATH* variables can also cause some of the above locations to be ignored.

Added in version 3.16: Added **CMAKE_FIND_USE_<CATEGORY>_PATH** variables to globally disable various search locations.

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

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

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

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

CMAKE_FIND_ROOT_PATH_BOTH
Search in the order described above.

NO_CMAKE_FIND_ROOT_PATH
Do not use the *CMAKE_FIND_ROOT_PATH* variable.

ONLY_CMAKE_FIND_ROOT_PATH
Search only the re−rooted directories and directories below *CMAKE_STAGING_PREFIX*.

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

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

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

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

The set of files considered to be programs is platform−specific:

- On Windows, filename suffixes are considered in order **.com**, **.exe**, and no suffix.
- On non−Windows systems, no filename suffix is considered, but files must have execute permission (see policy *CMP0109*).

To search for scripts, specify an extension explicitly:

```
if(WIN32)
  set(_script_suffix .bat)
else()
  set(_script_suffix .sh)
endif()

find_program(MY_SCRIPT NAMES my_script${_script_suffix})
```

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

```
foreach(<loop_var> <items>)
  <commands>
endforeach()
```

where **<items>** is a list of items that are separated by semicolon or whitespace. All commands between **foreach** and the matching **endforeach** are recorded without being invoked. Once the**endf oreach** is evaluated, the recorded list of commands is invoked once for each item in **<items>**. At the beginning of each iteration the variable **<loop_var>** will be set to the value of the current item.

The scope of **<loop_var>** is restricted to the loop scope. See policy *CMP0124* for details.

The commands *break()* and *continue()* provide means to escape from the normal control flow.

Per legacy, the *endforeach()* command admits an optional **<loop_var>** argument. If used, it must be a verbatim repeat of the argument of the opening **foreach** command.

```
foreach(<loop_var> RANGE <stop>)
```

In this variant, **foreach** iterates over the numbers 0, 1, ... up to (and including) the nonnegative integer <**stop**>.

```
foreach(<loop_var> RANGE <start> <stop> [<step>])
```

In this variant, **foreach** iterates over the numbers from <**start**> up to at most <**stop**> in steps of <**step**>. If <**step**> is not specified, then the step size is 1. The three arguments <**start**> <**stop**> <**step**> must all be nonnegative integers, and <**stop**> must not be smaller than <**start**>; otherwise you enter the danger zone of undocumented behavior that may change in future releases.

```
foreach(<loop_var> IN [LISTS [<lists>]] [ITEMS [<items>]])
```

In this variant, <**lists**> is a whitespace or semicolon separated list of list−valued variables. The **foreach** command iterates over each item in each given list. The <**items**> following the **ITEMS** keyword are processed as in the first variant of the **foreach** command. The forms **LISTS A** and **ITEMS ${A}** are equivalent.

The following example shows how the **LISTS** option is processed:

```
set(A 0;1)
set(B 2 3)
set(C "4 5")
set(D 6;7 8)
set(E "")
foreach(X IN LISTS A B C D E)
    message(STATUS "X=${X}")
endforeach()
```

yields:

```
-- X=0
-- X=1
-- X=2
-- X=3
-- X=4 5
-- X=6
-- X=7
-- X=8
```

```
foreach(<loop_var>... IN ZIP_LISTS <lists>)
```

Added in version 3.17.

In this variant, <**lists**> is a whitespace or semicolon separated list of list−valued variables. The **foreach** command iterates over each list simultaneously setting the iteration variables as follows:

- if the only **loop_var** given, then it sets a series of **loop_var_N** variables to the current item from the corresponding list;
- if multiple variable names passed, their count should match the lists variables count;

- if any of the lists are shorter, the corresponding iteration variable is not defined for the current iteration.

```
list(APPEND English one two three four)
list(APPEND Bahasa satu dua tiga)

foreach(num IN ZIP_LISTS English Bahasa)
    message(STATUS "num_0=${num_0}, num_1=${num_1}")
endforeach()

foreach(en ba IN ZIP_LISTS English Bahasa)
    message(STATUS "en=${en}, ba=${ba}")
endforeach()
```

yields:

```
-- num_0=one, num_1=satu
-- num_0=two, num_1=dua
-- num_0=three, num_1=tiga
-- num_0=four, num_1=
-- en=one, ba=satu
-- en=two, ba=dua
-- en=three, ba=tiga
-- en=four, ba=
```

See Also

- *break()*
- *continue()*
- *endforeach()*
- *while()*

function

Start recording a function for later invocation as a command.

```
function(<name> [<arg1> ...])
  <commands>
endfunction()
```

Defines a function named **<name>** that takes arguments named **<arg1>**, ... The **<commands>** in the function definition are recorded; they are not executed until the function is invoked.

Per legacy, the *endfunction()* command admits an optional **<name>** argument. If used, it must be a verbatim repeat of the argument of the opening **function** command.

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

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

See the *macro()* command documentation for differences between CMake functions and macros.

Invocation

The function invocation is case−insensitive. A function defined as

```
function(foo)
  <commands>
endfunction()
```

can be invoked through any of

```
foo()
Foo()
FOO()
cmake_language(CALL foo)
```

and so on. However, it is strongly recommended to stay with the case chosen in the function definition. Typically functions use all−lowercase names.

Added in version 3.18: The *cmake_language(CALL ...)* command can also be used to invoke the function.

Arguments

When the function is invoked, the recorded **<commands>** are first modified by replacing formal parameters (**${arg1}**, ...) with the arguments passed, and then invoked as normal commands.

In addition to referencing the formal parameters you can reference the **ARGC** variable which will be set to the number of arguments passed into the function as well as **ARGV0**, **ARGV1**, **ARGV2**, ... which will have the actual values of the arguments passed in. This facilitates creating functions with optional arguments.

Furthermore, **ARGV** holds the list of all arguments given to the function and **ARGN** holds the list of arguments past the last expected argument. Referencing to**ARGV#** ar guments beyond **ARGC** have undefined behavior. Checking that**ARGC** is greater than **#** is the only w ay to ensure that **ARGV#** was passed to the function as an extra argument.

See Also

- *cmake_parse_arguments()*
- *endfunction()*
- *return()*

get_cmake_property

Get a global property of the CMake instance.

```
get_cmake_property(<variable> <property>)
```

Gets a global property from the CMake instance. The value of the **<property>** is stored in the specified **<variable>**. If the property is not found,**<v ariable>** will be set to **NOTFOUND**. See the *cmake−properties(7)* manual for available properties.

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

See Also

- the *get_property()* command **GLOBAL** option

get_directory_property

Get a property of **DIRECTORY** scope.

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

Stores a property of directory scope in the named **<variable>**.

The **DIRECTORY** argument specifies another directory from which to retrieve the property value instead of the current directory. Relative paths are treated as relative to the current source directory. CMake must

already know about the directory, either by having added it through a call to *add_subdirectory()* or being the top level directory.

Added in version 3.19: **<dir>** may reference a binary directory.

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

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

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

See Also

- *define_property()*
- the more general *get_property()* command

get_filename_component

Get a specific component of a full filename.

Changed in version 3.20: This command has been superseded by the *cmake_path()* command, except for **REALPATH**, which is now offered by *file(REAL_PATH)*, and **PROGRAM**, now available in *separate_arguments(PROGRAM)*.

Changed in version 3.24: The undocumented feature offering the capability to query the **Windows** registry is superseded by *cmake_host_system_information(QUERY WINDOWS_REGISTRY)* command.

```
get_filename_component(<var> <FileName> <mode> [CACHE])
```

Sets **<var>** to a component of **<FileName>**, where **<mode>** is one of:

- **DIRECTORY** − directory without file name.
- **NAME** − file name without directory.
- **EXT** − file name longest extension (**.b.c** from **d/a.b.c**).
- **NAME_WE** − file name with neither the directory nor the longest extension.
- **LAST_EXT** − file name last extension (**.c** from **d/a.b.c**).
- **NAME_WLE** − file name with neither the directory nor the last extension.
- **PATH** − legacy alias for **DIRECTORY** (use for CMake <= 2.8.11).

Added in version 3.14: Added the **LAST_EXT** and **NAME_WLE** modes.

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

```
get_filename_component(<var> <FileName> <mode> [BASE_DIR <dir>] [CACHE])
```

Added in version 3.4.

Sets **<var>** to the absolute path of **<FileName>**, where **<mode>** is one of:

- **ABSOLUTE** − full path to file.
- **REALPATH** − full path to existing file with symlinks resolved.

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

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

```
get_filename_component(<var> <FileName> PROGRAM [PROGRAM_ARGS <arg_var>] [CACHE])
```

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

See Also

- *cmake_path()*

get_property

Get a property.

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

Gets one property from one object in a scope.

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

GLOBAL

Scope is unique and does not accept a name.

DIRECTORY

Scope defaults to the current directory, but another directory (already processed by CMake) may be named by the full or relative path **<dir>**. Relative paths are treated as relative to the current source directory. See also the*g et_directory_property()* command.

Added in version 3.19: **<dir>** may reference a binary directory.

TARGET

Scope must name one existing target. See also the*g et_target_property()* command.

SOURCE
Scope must name one source file. By default, the source file's property will be read from the current source directory's scope.

Added in version 3.18: Directory scope can be overridden with one of the following sub−options:

DIRECTORY <dir>
The source file property will be read from the **<dir>** directory's scope. CMake must already know about the directory, either by having added it through a call to *add_subdirectory()* or **<dir>** being the top level directory. Relative paths are treated as relative to the current source directory.

Added in version 3.19: **<dir>** may reference a binary directory.

TARGET_DIRECTORY <target>
The source file property will be read from the directory scope in which **<target>** was created (**<target>** must therefore already exist).

See also the *get_source_file_property()* command.

INSTALL
Added in version 3.1.

Scope must name one installed file path.

TEST Scope must name one existing test. See also the *get_test_property()* command.

Added in version 3.28: Directory scope can be overridden with the following sub−option:

DIRECTORY <dir>
The test property will be read from the **<dir>** directory's scope. CMake must already know about the directory, either by having added it through a call to *add_subdirectory()* or **<dir>** being the top level directory. Relative paths are treated as relative to the current source directory. **<dir>** may reference a binary directory.

CACHE
Scope must name one cache entry.

VARIABLE
Scope is unique and does not accept a name.

The required **PROPERTY** option is immediately followed by the name of the property to get. If the property is not set, the named **<variable>** will be unset in the calling scope upon return, although some properties support inheriting from a parent scope if defined to behave that way (see *define_property()*).

If the **SET** option is given, the variable is set to a boolean value indicating whether the property has been set. If the**DEFINED** option is gi ven, the variable is set to a boolean value indicating whether the property has been defined, such as with the *define_property()* command.

If **BRIEF_DOCS** or **FULL_DOCS** is given, then the variable is set to a string containing documentation for the requested property. If documentation is requested for a property that has not been defined,**NO T-FOUND** is returned.

NOTE:
The *GENERATED* source file property may be globally visible. See its documentation for details.

See Also

- *define_property()*
- *set_property()*

if

Conditionally execute a group of commands.

Synopsis

```
if(<condition>)
  <commands>
elseif(<condition>) # optional block, can be repeated
  <commands>
else()              # optional block
  <commands>
endif()
```

Evaluates the **condition** argument of the **if** clause according to the *Condition syntax* described below. If the result is true, then the **commands** in the **if** block are executed. Otherwise, optional**elseif** blocks are processed in the same way. Finally, if no **condition** is true, **commands** in the optional **else** block are executed.

Per legacy, the *else()* and *endif()* commands admit an optional **<condition>** argument. If used, it must be a verbatim repeat of the argument of the opening **if** command.

Condition Syntax

The following syntax applies to the **condition** argument of the **if**, **elseif** and *while()* clauses.

Compound conditions are evaluated in the following order of precedence:

1. *Parentheses*.
2. Unary tests such as *COMMAND*, *POLICY*, *TARGET*, *TEST*, *EXISTS*, *IS_READABLE*, *IS_WRITABLE*, *IS_EXECUTABLE*, *IS_DIRECTORY*, *IS_SYMLINK*, *IS_ABSOLUTE*, and *DEFINED*.
3. Binary tests such as *EQUAL*, *LESS*, *LESS_EQUAL*, *GREATER*, *GREATER_EQUAL*, *STREQUAL*, *STRLESS*, *STRLESS_EQUAL*, *STRGREATER*, *STRGREATER_EQUAL*, *VERSION_EQUAL*, *VERSION_LESS*, *VERSION_LESS_EQUAL*, *VERSION_GREATER*, *VERSION_GREATER_EQUAL*, *PATH_EQUAL*, *IN_LIST*, *IS_NEWER_THAN*, and *MATCHES*.
4. Unary logical operator *NOT*.
5. Binary logical operators *AND* and *OR*, from left to right, without any short−circuit.

Basic Expressions

if(<constant>)

True if the constant is **1**, **ON**, **YES**, **TRUE**, **Y**, or a non−zero number (including floating point numbers). False if the constant is **0**, **OFF**, **NO**, **FALSE**, **N**, **IGNORE**, **NOTFOUND**, the empty string, or ends in the suffix **−NOTFOUND**. Named boolean constants are case−insensitive. If the argument is not one of these specific constants, it is treated as a variable or string (see *Variable Expansion* further below) and one of the following two forms applies.

if(<variable>)

True if given a variable that is defined to a value that is not a false constant. False otherwise, including if the variable is undefined. Note that macro arguments are not variables. *Environment Variables* also cannot be tested this way, e.g. **if(ENV{some_var})** will always evaluate to false.

if(<string>)

A quoted string always evaluates to false unless:

- The string's value is one of the true constants, or
- in CMake versions prior to 4.0, policy *CMP0054* is not set to **NEW** and the string's value happens to be a variable name that is affected by *CMP0054*'s behavior.

Logic Operators

if(NOT <condition>)
True if the condition is not true.

if(<cond1> AND <cond2>)
True if both conditions would be considered true individually.

if(<cond1> OR <cond2>)
True if either condition would be considered true individually.

if((condition) AND (condition OR (condition)))
The conditions inside the parenthesis are evaluated first and then the remaining condition is evaluated as in the other examples. Where there are nested parenthesis the innermost are evaluated as part of evaluating the condition that contains them.

Existence Checks

if(COMMAND <command−name>)
True if the given name is a command, macro or function that can be invoked.

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

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

if(TEST <test−name>)
Added in version 3.3.

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

if(DEFINED <name>|CACHE{<name>}|ENV{<name>})
True if a variable, cache variable or environment variable with given **<name>** is defined. The value of the variable does not matter. Note the following caveats:

- Macro arguments are not variables.
- It is not possible to test directly whether a **<name>** is a non−cache variable. The expression **if(DEFINED someName)** will evaluate to true if either a cache or non−cache variable **someName** exists. In comparison, the expression **if(DEFINED CACHE{someName})** will only evaluate to true if a cache variable **someName** exists. Both expressions need to be tested if you need to know whether a non−cache variable exists: **if(DEFINED someName AND NOT DEFINED CACHE{someName})**.

Added in version 3.14: Added support for **CACHE{<name>}** variables.

if(<variable|string> IN_LIST <variable>)
Added in version 3.3.

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

File Operations

if(EXISTS <path−to−file−or−directory>)
True if the named file or directory exists and is readable. Behavior is well−defined only for explicit full paths (a leading ~/ is not expanded as a home directory and is considered a relative path).

Resolves symbolic links, i.e. if the named file or directory is a symbolic link, returns true if the target of the symbolic link exists.

False if the given path is an empty string.

NOTE:
Prefer **if(IS_READABLE)** to check file readability. **if(EXISTS)** may be changed in the future to only check file existence.

if(IS_READABLE <path−to−file−or−directory>)
Added in version 3.29.

True if the named file or directory is readable. Behavior is well−defined only for explicit full paths (a leading ˜/ is not expanded as a home directory and is considered a relative path). Resolves symbolic links, i.e. if the named file or directory is a symbolic link, returns true if the target of the symbolic link is readable.

False if the given path is an empty string.

if(IS_WRITABLE <path−to−file−or−directory>)
Added in version 3.29.

True if the named file or directory is writable. Behavior is well−defined only for explicit full paths (a leading ˜/ is not expanded as a home directory and is considered a relative path). Resolves symbolic links, i.e. if the named file or directory is a symbolic link, returns true if the target of the symbolic link is writable.

False if the given path is an empty string.

if(IS_EXECUTABLE <path−to−file−or−directory>)
Added in version 3.29.

True if the named file or directory is executable. Behavior is well−defined only for explicit full paths (a leading ˜/ is not expanded as a home directory and is considered a relative path). Resolves symbolic links, i.e. if the named file or directory is a symbolic link, returns true if the target of the symbolic link is executable.

False if the given path is an empty string.

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

if(IS_DIRECTORY <path>)
True if **path** is a directory. Behavior is well−defined only for full paths.

False if the given path is an empty string.

if(IS_SYMLINK <path>)
True if the given path is a symbolic link. Behavior is well−defined only for full paths.

if(IS_ABSOLUTE <path>)
True if the given path is an absolute path. Note the following special cases:

- An empty **path** evaluates to false.
- On Windows hosts, any **path** that begins with a drive letter and colon (e.g. **C:**), a forward slash or a backslash will evaluate to true. This means a path like **C:no\base\dir** will evaluate to true, even though the non−drive part of the path is relative.
- On non−Windows hosts, any **path** that begins with a tilde (˜) evaluates to true.

Comparisons

if(<variable|string> MATCHES <regex>)
True if the given string or variable's value matches the given regular expression. See*Re gex Specification* for regex format.

Added in version 3.9: **()** groups are captured in *CMAKE_MATCH_*<*n*> variables.

if(<variable|string> LESS <variable|string>)
True if the given string or variable's value parses as a real number (like a C **double**) and less than that on the right.

if(<variable|string> GREATER <variable|string>)
True if the given string or variable's value parses as a real number (like a C **double**) and greater than that on the right.

if(<variable|string> EQUAL <variable|string>)
True if the given string or variable's value parses as a real number (like a C **double**) and equal to that on the right.

if(<variable|string> LESS_EQUAL <variable|string>)
Added in version 3.7.

True if the given string or variable's value parses as a real number (like a C **double**) and less than or equal to that on the right.

if(<variable|string> GREATER_EQUAL <variable|string>)
Added in version 3.7.

True if the given string or variable's value parses as a real number (like a C **double**) and greater than or equal to that on the right.

if(<variable|string> STRLESS <variable|string>)
True if the given string or variable's value is lexicographically less than the string or variable on the right.

if(<variable|string> STRGREATER <variable|string>)
True if the given string or variable's value is lexicographically greater than the string or variable on the right.

if(<variable|string> STREQUAL <variable|string>)
True if the given string or variable's value is lexicographically equal to the string or variable on the right.

if(<variable|string> STRLESS_EQUAL <variable|string>)
Added in version 3.7.

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

if(<variable|string> STRGREATER_EQUAL <variable|string>)
Added in version 3.7.

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

Version Comparisons

if(<variable|string> VERSION_LESS <variable|string>)
Component−wise integer version number comparison (version format is **major[.minor[.patch[.tweak]]]**, omitted components are treated as zero). Any non−integer version component or non−integer trailing part of a version component effectively truncates the string at that point.

if(<variable|string> VERSION_GREATER <variable|string>)
Component−wise integer version number comparison (version format is **major[.minor[.patch[.tweak]]]**, omitted components are treated as zero). Any non−integer version component or non−integer trailing part of a version component effectively truncates the string at that point.

if(<variable|string> VERSION_EQUAL <variable|string>)
Component−wise integer version number comparison (version format is **major[.minor[.patch[.tweak]]]**, omitted components are treated as zero). Any non−integer version component or non−integer trailing part of a version component effectively truncates the string at that point.

if(<variable|string> VERSION_LESS_EQUAL <variable|string>)
Added in version 3.7.

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

if(<variable|string> VERSION_GREATER_EQUAL <variable|string>)
Added in version 3.7.

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

Path Comparisons

if(<variable|string> PATH_EQUAL <variable|string>)
Added in version 3.24.

Lexicographically compares two CMake paths component−by−component without accessing the filesystem. Only if every component of both paths match will the two paths compare equal. Multiple path separators are effectively collapsed into a single separator, but note that backslashes are not converted to forward slashes. No other *path normalization* is performed. Trailing slashes are preserved, thus **/a/b** and **/a/b/** are not equal.

Component−wise comparison is superior to string−based comparison due to the handling of multiple path separators. In the following example, the expression evaluates to true using **PATH_EQUAL**, but false with **STREQUAL**:

```
# comparison is TRUE
if ("/a//b/c" PATH_EQUAL "/a/b/c")
   ...
endif()

# comparison is FALSE
if ("/a//b/c" STREQUAL "/a/b/c")
   ...
endif()
```

See *cmake_path(COMPARE)* for more details.

Variable Expansion

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

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

appears to the if command as

```
if(var1)
```

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

```
if(var2)
```

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

Automatic evaluation applies in the other cases whenever the above−documented condition syntax accepts **<variable|string>**:

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

- The left and right hand arguments to *AND* and *OR* are independently tested to see if they are boolean constants. If so, they are used as such, otherwise they are assumed to be variables and are dereferenced.

Changed in version 3.1: To prevent ambiguity, potential variable or keyword names can be specified in a *Quoted Argument* or a *Bracket Argument*. A quoted or bracketed variable or keyword will be interpreted as a string and not dereferenced or interpreted. See policy *CMP0054*.

There is no automatic evaluation for environment or cache *Variable References*. Their values must be referenced as **$ENV{<name>}** or **$CACHE{<name>}** wherever the above−documented condition syntax accepts **<variable|string>**.

See also

- *else()*
- *elseif()*
- *endif()*

include

Load and run CMake code from a file or module.

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

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

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

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

include_guard

Added in version 3.10.

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

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

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

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

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

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

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

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

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

list

Operations on *semicolon−separated lists*.

Synopsis

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

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

Modification
  list(APPEND <list> [<element>...])
  list(FILTER <list> {INCLUDE | EXCLUDE} REGEX <regex>)
  list(INSERT <list> <index> [<element>...])
  list(POP_BACK <list> [<out-var>...])
  list(POP_FRONT <list> [<out-var>...])
  list(PREPEND <list> [<element>...])
  list(REMOVE_ITEM <list> <value>...)
  list(REMOVE_AT <list> <index>...)
  list(REMOVE_DUPLICATES <list>)
  list(TRANSFORM <list> <ACTION> [...])

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

Introduction

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

NOTE:

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

Individual elements may not contain an unequal number of **[** and **]** characters, and may not end in a backslash (\). See*semicolon−separ ated lists* for details.

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

Reading

list(LENGTH <list> <output variable>)
Returns the list's length.

list(GET <list> <element index> [<element index> ...] <output variable>)
Returns the list of elements specified by indices from the list.

list(JOIN <list> <glue> <output variable>)
Added in version 3.12.

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

list(SUBLIST <list> <begin> <length> <output variable>)
Added in version 3.12.

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

Search

list(FIND <list> <value> <output variable>)
Returns the index of the element specified in the list or **−1** if it wasn't found.

Modification

list(APPEND <list> [<element> ...])
Appends elements to the list. If no variable named **<list>** exists in the current scope its value is treated as empty and the elements are appended to that empty list.

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

Added in version 3.6.

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

For more information on regular expressions look under *string(REGEX)*.

list(INSERT <list> <element_index> <element> [<element> ...])
Inserts elements to the list to the specified index. It is an error to specify an out−of−range index. Valid indexes are *0* to *N* where *N* is the length of the list, inclusive. An empty list has length 0. If no variable named **<list>** exists in the current scope its value is treated as empty and the elements

are inserted in that empty list.

list(POP_BACK <list> [<out−var>...])
Added in version 3.15.

If no variable name is given, removes exactly one element. Otherwise, with *N* variable names provided, assign the last *N* elements' values to the given variables and then remove the last *N* values from **<list>**.

list(POP_FRONT <list> [<out−var>...])
Added in version 3.15.

If no variable name is given, removes exactly one element. Otherwise, with *N* variable names provided, assign the first *N* elements' values to the given variables and then remove the first *N* values from **<list>**.

list(PREPEND <list> [<element> ...])
Added in version 3.15.

Insert elements to the 0th position in the list. If no variable named **<list>** exists in the current scope its value is treated as empty and the elements are prepended to that empty list.

list(REMOVE_ITEM <list> <value> [<value> ...])
Removes all instances of the given items from the list.

list(REMOVE_AT <list> <index> [<index> ...])
Removes items at given indices from the list.

list(REMOVE_DUPLICATES <list>)
Removes duplicated items in the list. The relative order of items is preserved, but if duplicates are encountered, only the first instance is preserved.

list(TRANSFORM <list> <ACTION> [<SELECTOR>] [OUTPUT_VARIABLE <output variable>])
Added in version 3.12.

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

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

<ACTION> specifies the action to apply to the elements of the list. The actions have exactly the same semantics as sub−commands of the *string()* command. **<ACTION>** must be one of the following:

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

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

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

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

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

list(TRANSFORM <list> STRIP ...)

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

list(TRANSFORM <list> GENEX_STRIP ...)

REPLACE
Match the regular expression as many times as possible and substitute the replacement expression for the match for each element of the list (same semantic as *string(REGEX REPLACE)*).

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

<SELECTOR> determines which elements of the list will be transformed. Only one type of selector can be specified at a time. When given, **<SELECTOR>** must be one of the following:

AT Specify a list of indexes.

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

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

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

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

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

Ordering

list(REVERSE <list>)
Reverses the contents of the list in−place.

list(SORT <list> [COMPARE <compare>] [CASE <case>] [ORDER <order>])
Sorts the list in−place alphabetically.

Added in version 3.13: Added the **COMPARE**, **CASE**, and **ORDER** options.

Added in version 3.18: Added the **COMPARE NATURAL** option.

Use the **COMPARE** keyword to select the comparison method for sorting. The **<compare>** option should be one of:

STRING
Sorts a list of strings alphabetically. This is the default behavior if the **COMPARE** option is not given.

FILE_BASENAME
Sorts a list of pathnames of files by their basenames.

NATURAL
Sorts a list of strings using natural order (see **strverscmp(3)** manual), i.e. such that contiguous digits are compared as whole numbers. For example: the following list *10.0 1.1 2.1 8.0 2.0 3.1* will be sorted as *1.1 2.0 2.1 3.1 8.0 10.0* if the **NATURAL**

comparison is selected where it will be sorted as *1.1 10.0 2.0 2.1 3.1 8.0* with the **STRING** comparison.

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

SENSITIVE
List items are sorted in a case−sensitive manner. This is the default behavior if the **CASE** option is not given.

INSENSITIVE
List items are sorted case insensitively. The order of items which differ only by upper/lowercase is not specified.

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

ASCENDING
Sorts the list in ascending order. This is the default behavior when the **ORDER** option is not given.

DESCENDING
Sorts the list in descending order.

load_cache
Load in the values from another project's **CMakeCache.txt** cache file. This is useful for projects that depend on another project built in a separate directory tree.

This command has two signatures. The recommended signature is:

load_cache(<build−dir> READ_WITH_PREFIX <prefix> <entry>...)
Loads the cache file from the specified **<build−dir>** build directory and retrieves the listed cache entries. The retrieved values are stored in local variables, with their names prefixed by the provided **<prefix>**. This only reads the cache values; it does not create or modify entries in the local project's cache.

READ_WITH_PREFIX <prefix>
For each cache **<entry>**, a local variable is created using the specified **<prefix>** followed by the entry name.

This signature can be also used in *cmake −P* script mode.

The following signature of this command is strongly discouraged, but it is provided for backward compatibility:

load_cache(<build−dir> [EXCLUDE <entry>...] [INCLUDE_INTERNALS <entry>...])
This form loads the cache file from the specified **<build−dir>** build directory and imports all its non−internal cache entries into the local project's cache as internal cache variables. By default, only non−internal entries are imported, unless the **INCLUDE_INTERNALS** option is used.

The options are:

EXCLUDE <entry>...
This option can be used to exclude a given list of non−internal cache entries when importing values.

INCLUDE_INTERNALS <entry>...
This option can be used to provide a list of internal cache entries to include in addition to the non−internal cache entries.

This signature can be used only in CMake projects. Script mode is not supported.

NOTE:
Instead of loading the outside project's cache file and manually accessing variables, a more robust and convenient approach is to use the *export()* command in the outside project, when available. This allows the project to provide its targets, configuration, or features in a structured and maintainable way, making integration simpler and less error−prone.

Examples

Reading specific cache variables from another project and storing them as local variables:

```
load_cache(
  path/to/other-project/build-dir
  READ_WITH_PREFIX prefix_
  OTHER_PROJECT_CACHE_VAR_1
  OTHER_PROJECT_CACHE_VAR_2
)

message(STATUS "${prefix_OTHER_PROJECT_CACHE_VAR_1")
message(STATUS "${prefix_OTHER_PROJECT_CACHE_VAR_2")
# Outputs:
# -- some-value...
# -- another-value...
```

Reading all non−internal cache entries from another project and storing them as internal cache variables using the obsolete signature:

```
load_cache(path/to/other-project/build-dir)

message(STATUS "${OTHER_PROJECT_CACHE_VAR_1")
message(STATUS "${OTHER_PROJECT_CACHE_VAR_2")
# Outputs:
# -- some-value...
# -- another-value...
```

Excluding specific non−internal cache entries and including internal ones using the obsolete signature:

```
load_cache(
  path/to/other-project/build-dir
  EXCLUDE OTHER_PROJECT_CACHE_VAR_2
  INCLUDE_INTERNALS OTHER_PROJECT_INTERNAL_CACHE_VAR
)

message(STATUS "${OTHER_PROJECT_CACHE_VAR_1")
message(STATUS "${OTHER_PROJECT_CACHE_VAR_2")
message(STATUS "${OTHER_PROJECT_INTERNAL_CACHE_VAR}")
# Outputs:
# -- some-value...
# --
# -- some-internal-value...
```

macro

Start recording a macro for later invocation as a command

```
macro(<name> [<arg1> ...])
  <commands>
```

```
endmacro()
```

Defines a macro named **<name>** that takes arguments named **<arg1>**, ... Commands listed after macro, but before the matching *endmacro()*, are not executed until the macro is invoked.

Per legacy, the *endmacro()* command admits an optional **<name>** argument. If used, it must be a verbatim repeat of the argument of the opening **macro** command.

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

See the *Macro vs Function* section below for differences between CMake macros and *functions*.

Invocation

The macro invocation is case−insensitive. A macro defined as

```
macro(foo)
  <commands>
endmacro()
```

can be invoked through any of

```
foo()
Foo()
FOO()
cmake_language(CALL foo)
```

and so on. However, it is strongly recommended to stay with the case chosen in the macro definition. Typically macros use all−lowercase names.

Added in version 3.18: The *cmake_language(CALL ...)* command can also be used to invoke the macro.

Arguments

When a macro is invoked, the commands recorded in the macro are first modified by replacing formal parameters (**${arg1}**, ...) with the arguments passed, and then invoked as normal commands.

In addition to referencing the formal parameters you can reference the values **${ARGC}** which will be set to the number of arguments passed into the macro as well as **${ARGV0}**, **${ARGV1}**, **${ARGV2}**, ... which will have the actual values of the arguments passed in. This facilitates creating macros with optional arguments.

Furthermore, **${ARGV}** holds the list of all arguments given to the macro and **${ARGN}** holds the list of arguments past the last expected argument. Referencing to**${ARGV#}** ar guments beyond **${ARGC}** have undefined behavior. Checking that **${ARGC}** is greater than **#** is the only way to ensure that **${ARGV#}** was passed to the function as an extra argument.

Macro vs Function

The **macro** command is very similar to the *function()* command. Nonetheless, there are a few important differences.

In a function, **ARGN**, **ARGC**, **ARGV** and **ARGV0**, **ARGV1**, ... are true variables in the usual CMake sense. In a macro, they are not, they are string replacements much like the C preprocessor would do with a macro. This has a number of consequences, as explained in the *Argument Caveats* section below.

Another difference between macros and functions is the control flow. A function is executed by transferring control from the calling statement to the function body. A macro is executed as if the macro body

were pasted in place of the calling statement. This has the consequence that a *return()* in a macro body does not just terminate execution of the macro; rather, control is returned from the scope of the macro call. To avoid confusion, it is recommended to avoid *return()* in macros altogether.

Unlike a function, the *CMAKE_CURRENT_FUNCTION*, *CMAKE_CURRENT_FUNCTION_LIST_DIR*, *CMAKE_CURRENT_FUNCTION_LIST_FILE*, *CMAKE_CURRENT_FUNCTION_LIST_LINE* variables are not set for a macro.

Argument Caveats

Since **ARGN**, **ARGC**, **ARGV**, **ARGV0** etc. are not variables, you will NOT be able to use commands like

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

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

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

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

```
macro(bar)
  foreach(arg IN LISTS ARGN)
    <commands>
  endforeach()
endmacro()

function(foo)
  bar(x y z)
endfunction()

foo(a b c)
```

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

See Also

- *cmake_parse_arguments()*
- *endmacro()*

mark_as_advanced

Mark cmake cached variables as advanced.

```
mark_as_advanced([CLEAR|FORCE] <var1> ...)
```

Sets the advanced/non−advanced state of the named cached variables.

An advanced variable will not be displayed in any of the cmake GUIs unless the **show advanced** option is on. In script mode, the advanced/non−advanced state has no effect.

If the keyword **CLEAR** is given then advanced variables are changed back to unadvanced. If the keyword

FORCE is given then the variables are made advanced. If neither**FORCE** nor **CLEAR** is specified, ne w values will be marked as advanced, but if a variable already has an advanced/non−advanced state, it will not be changed.

Changed in version 3.17: Variables passed to this command which are not already in the cache are ignored. See policy *CMP0102*.

math

Evaluate a mathematical expression.

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

Evaluates a mathematical <**expression**> and sets <**variable**> to the resulting value. The result of the expression must be representable as a 64−bit signed integer. Floating point inputs are invalid e.g. **1.1 * 10**. Non−integer results e.g. **3 / 2** are truncated.

The mathematical expression must be given as a string (i.e. enclosed in double quotation marks). An example is "**5 * (10 + 13)**". Supported operators are**+**, **−**, *****, /, **%**, |, **&**, **^**, **~**, <<, >>, and **(...)**; the y have the same meaning as in C code.

Added in version 3.13: Hexadecimal numbers are recognized when prefixed with **0x**, as in C code.

Added in version 3.13: The result is formatted according to the option **OUTPUT_FORMAT**, where <**format**> is one of

HEXADECIMAL
Hexadecimal notation as in C code, i. e. starting with "0x".

DECIMAL
Decimal notation. Which is also used if no **OUTPUT_FORMAT** option is specified.

For example

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

message

Log a message.

Synopsis

```
General messages
  message([<mode>] "message text" ...)

Reporting checks
  message(<checkState> "message text" ...)

Configure Log
  message(CONFIGURE_LOG <text>...)
```

General messages

```
message([<mode>] "message text" ...)
```

Record the specified message text in the log. If more than one message string is given, they are concatenated into a single message with no separator between the strings.

The optional **<mode>** keyword determines the type of message, which influences the way the message is handled:

FATAL_ERROR
CMake Error, stop processing and generation.

The *cmake(1)* executable will return a non−zero *exit code*.

SEND_ERROR
CMake Error, continue processing, but skip generation.

WARNING
CMake Warning, continue processing.

AUTHOR_WARNING
CMake Warning (dev), continue processing.

DEPRECATION
CMake Deprecation Error or Warning if variable *CMAKE_ERROR_DEPRECATED* or *CMAKE_WARN_DEPRECATED* is enabled, respectively, else no message.

(none) or NOTICE
Important message printed to stderr to attract user's attention.

STATUS
The main interesting messages that project users might be interested in. Ideally these should be concise, no more than a single line, but still informative.

VERBOSE
Detailed informational messages intended for project users. These messages should provide additional details that won't be of interest in most cases, but which may be useful to those building the project when they want deeper insight into what's happening.

DEBUG
Detailed informational messages intended for developers working on the project itself as opposed to users who just want to build it. These messages will not typically be of interest to other users building the project and will often be closely related to internal implementation details.

TRACE
Fine−grained messages with very low−level implementation details. Messages using this log level would normally only be temporary and would expect to be removed before releasing the project, packaging up the files, etc.

Added in version 3.15: Added the **NOTICE**, **VERBOSE**, **DEBUG**, and **TRACE** levels.

The CMake command−line tool displays **STATUS** to **TRACE** messages on stdout with the message preceded by two hyphens and a space. All other message types are sent to stderr and are not prefixed with hyphens. The*CMak e GUI* displays all messages in its log area. The *curses interface* shows **STATUS** to **TRACE** messages one at a time on a status line and other messages in an interactive pop−up box. The *−−log−level* command−line option to each of these tools can be used to control which messages will be shown.

Added in version 3.17: To make a log level persist between CMake runs, the *CMAKE_MESSAGE_LOG_LEVEL* variable can be set instead. Note that the command line option takes precedence over the cache variable.

Added in version 3.16: Messages of log levels **NOTICE** and below will have each line preceded by the content of the *CMAKE_MESSAGE_INDENT* variable (converted to a single string by concatenating its list

items). For **STATUS** to **TRACE** messages, this indenting content will be inserted after the hyphens.

Added in version 3.17: Messages of log levels **NOTICE** and below can also have each line preceded with context of the form **[some.context.example]**. The content between the square brackets is obtained by converting the *CMAKE_MESSAGE_CONTEXT* list variable to a dot−separated string. The message context will always appear before any indenting content but after any automatically added leading hyphens. By default, message context is not shown, it has to be explicitly enabled by giving the *cmake −−log−context* command−line option or by setting the *CMAKE_MESSAGE_CONTEXT_SHOW* variable to true. See the *CMAKE_MESSAGE_CONTEXT* documentation for usage examples.

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

Reporting checks

Added in version 3.17.

A common pattern in CMake output is a message indicating the start of some sort of check, followed by another message reporting the result of that check. For example:

```
message(STATUS "Looking for someheader.h")
#... do the checks, set checkSuccess with the result
if(checkSuccess)
  message(STATUS "Looking for someheader.h − found")
else()
  message(STATUS "Looking for someheader.h − not found")
endif()
```

This can be more robustly and conveniently expressed using the **CHECK_...** keyword form of the **message()** command:

```
message(<checkState> "message" ...)
```

where <**checkState**> must be one of the following:

CHECK_START
: Record a concise message about the check about to be performed.

CHECK_PASS
: Record a successful result for a check.

CHECK_FAIL
: Record an unsuccessful result for a check.

When recording a check result, the command repeats the message from the most recently started check for which no result has yet been reported, then some separator characters and then the message text provided after the **CHECK_PASS** or **CHECK_FAIL** keyword. Check messages are always reported at **STATUS** log level.

Checks may be nested and every **CHECK_START** should have exactly one matching **CHECK_PASS** or **CHECK_FAIL**. The*CMAKE_MESSA GE_INDENT* variable can also be used to add indenting to nested checks if desired. For example:

```
message(CHECK_START "Finding my things")
list(APPEND CMAKE_MESSAGE_INDENT "  ")
```

```
unset(missingComponents)

message(CHECK_START "Finding partA")
# ... do check, assume we find A
message(CHECK_PASS "found")

message(CHECK_START "Finding partB")
# ... do check, assume we don't find B
list(APPEND missingComponents B)
message(CHECK_FAIL "not found")

list(POP_BACK CMAKE_MESSAGE_INDENT)
if(missingComponents)
  message(CHECK_FAIL "missing components: ${missingComponents}")
else()
  message(CHECK_PASS "all components found")
endif()
```

Output from the above would appear something like the following:

```
-- Finding my things
--   Finding partA
--   Finding partA - found
--   Finding partB
--   Finding partB - not found
-- Finding my things - missing components: B
```

Configure Log

Added in version 3.26.

```
message(CONFIGURE_LOG <text>...)
```

Record a *configure-log message event* with the specified **<text>**. By convention, if the text contains more than one line, the first line should be a summary of the event.

This mode is intended to record the details of a system inspection check or other one-time operation guarded by a cache entry, but that is not performed using *try_compile()* or *try_run()*, which automatically log their details. Projects should avoid calling it every time CMake runs. For example:

```
if (NOT DEFINED MY_CHECK_RESULT)
  # Print check summary in configure output.
  message(CHECK_START "My Check")

  # ... perform system inspection, e.g., with execute_process ...

  # Cache the result so we do not run the check again.
  set(MY_CHECK_RESULT "${MY_CHECK_RESULT}" CACHE INTERNAL "My Check")

  # Record the check details in the cmake-configure-log.
  message(CONFIGURE_LOG
    "My Check Result: ${MY_CHECK_RESULT}\n"
    "${details}"
  )
```

```
  # Print check result in configure output.
  if(MY_CHECK_RESULT)
    message(CHECK_PASS "passed")
  else()
    message(CHECK_FAIL "failed")
  endif()
endif()
```

If no project is currently being configured, such as in *cmake −P* script mode, this command does nothing.

See Also

- *cmake_language(GET_MESSAGE_LOG_LEVEL)*

option

Provide a boolean option that the user can optionally select.

```
option(<variable> "<help_text>" [value])
```

If no initial **<value>** is provided, boolean **OFF** is the default value. If**<v ariable>** is already set as a normal or cache variable, then the command does nothing (see policy *CMP0077*).

For options that depend on the values of other options, see the module help for *CMakeDependentOption*.

In CMake project mode, a boolean cache variable is created with the option value. In CMake script mode, a boolean variable is set with the option value.

return

Return from a file, directory or function.

```
return([PROPAGATE <var-name>...])
```

When this command is encountered in an included file (via *include()* or *find_package()*), it causes processing of the current file to stop and control is returned to the including file. If it is encountered in a file which is not included by another file, e.g. a **CMakeLists.txt**, deferred calls scheduled by *cmake_language(DEFER)* are invoked and control is returned to the parent directory if there is one.

If **return()** is called in a function, control is returned to the caller of that function. Note that a *macro()*, unlike a *function()*, is expanded in place and therefore cannot handle **return()**.

Policy *CMP0140* controls the behavior regarding the arguments of the command. All arguments are ignored unless that policy is set to **NEW**.

PROPAGATE

Added in version 3.25.

This option sets or unsets the specified variables in the parent directory or function caller scope. This is equivalent to *set(PARENT_SCOPE)* or *unset(PARENT_SCOPE)* commands, except for the way it interacts with the *block()* command, as described below.

The **PROPAGATE** option can be very useful in conjunction with the *block()* command. A **return** will propagate the specified variables through any enclosing block scopes created by the *block()* commands. Inside a function, this ensures the variables are propagated to the function's caller, regardless of any blocks within the function. If not inside a function, it ensures the variables are propagated to the parent file or directory scope. For example:

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.25)
project(example)

set(var1 "top-value")

block(SCOPE_FOR VARIABLES)
  add_subdirectory(subDir)
  # var1 has the value "block-nested"
endblock()

# var1 has the value "top-value"
```

subDir/CMakeLists.txt

```
function(multi_scopes result_var1 result_var2)
  block(SCOPE_FOR VARIABLES)
    # This would only propagate out of the immediate block, not to
    # the caller of the function.
    #set(${result_var1} "new-value" PARENT_SCOPE)
    #unset(${result_var2} PARENT_SCOPE)

    # This propagates the variables through the enclosing block and
    # out to the caller of the function.
    set(${result_var1} "new-value")
    unset(${result_var2})
    return(PROPAGATE ${result_var1} ${result_var2})
  endblock()
endfunction()

set(var1 "some-value")
set(var2 "another-value")

multi_scopes(var1 var2)
# Now var1 will hold "new-value" and var2 will be unset

block(SCOPE_FOR VARIABLES)
  # This return() will set var1 in the directory scope that included us
  # via add_subdirectory(). The surrounding block() here does not limit
  # propagation to the current file, but the block() in the parent
  # directory scope does prevent propagation going any further.
  set(var1 "block-nested")
  return(PROPAGATE var1)
endblock()
```

See Also

- *block()*
- *function()*

separate_arguments

Parse command-line arguments into a semicolon-separated list.

```
separate_arguments(<variable> <mode> [PROGRAM [SEPARATE_ARGS]] <args>)
```

Parses a space-separated string <**args**> into a list of items, and stores this list in semicolon-separated standard form in <**variable**>.

This function is intended for parsing command−line arguments. The entire command line must be passed as one string in the argument **<args>**.

The exact parsing rules depend on the operating system. They are specified by the **<mode>** argument which must be one of the following keywords:

UNIX_COMMAND
Arguments are separated by unquoted whitespace. Both single−quote and double−quote pairs are respected. A backslash escapes the next literal character (\" is "); there are no special escapes (**\n** is just **n**).

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

NATIVE_COMMAND
Added in version 3.9.

Proceeds as in **WINDOWS_COMMAND** mode if the host system is Windows. Otherwise proceeds as in **UNIX_COMMAND** mode.

PROGRAM
Added in version 3.19.

The first item in **<args>** is assumed to be an executable and will be searched in the system search path or left as a full path. If not found, **<variable>** will be empty. Otherwise, **<variable>** is a list of 2 elements:

0. Absolute path of the program
1. Any command−line arguments present in **<args>** as a string

For example:

```
separate_arguments (out UNIX_COMMAND PROGRAM "cc −c main.c")
```

- First element of the list: **/path/to/cc**
- Second element of the list: **" −c main.c**"

SEPARATE_ARGS
When this sub−option of **PROGRAM** option is specified, command−line arguments will be split as well and stored in **<variable>**.

For example:

```
separate_arguments (out UNIX_COMMAND PROGRAM SEPARATE_ARGS "cc −c main.c")
```

The contents of **out** will be: **/path/to/cc;−c;main.c**

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

Convert the value of **<var>** to a semi−colon separated list. All spaces are replaced with ';'. This helps with generating command lines.

set

Set a normal, cache, or environment variable to a given value. See the*cmak e−language(7) variables* documentation for the scopes and interaction of normal variables and cache entries.

Signatures of this command that specify a **<value>...** placeholder expect zero or more arguments. Multiple arguments will be joined as a *semicolon−separated list* to form the actual variable value to be set.

Set Normal Variable

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

Set or unset **<variable>** in the current function or directory scope:

- If at least one **<value>...** is given, set the variable to that value.
- If no value is given, unset the variable. This is equivalent to *unset(<variable>)*.

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

The *block(PROPAGATE)* and *return(PROPAGATE)* commands can be used as an alternate method to the *set(PARENT_SCOPE)* and *unset(PARENT_SCOPE)* commands to update the parent scope.

NOTE:

When evaluating *Variable References* of the form **${VAR}**, CMake first searches for a normal variable with that name. If no such normal variable exists, CMake will then search for a cache entry with that name. Because of this,**unsetting a normal v ariable can expose a cache variable that was previously hidden**. To force a variable reference of the form **${VAR}** to return an empty string, use **set(<variable> "")**, which clears the normal variable but leaves it defined.

Set Cache Entry

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

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

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

BOOL Boolean **ON/OFF** value. *cmake−gui(1)* offers a checkbox.

FILEPATH
Path to a file on disk. *cmake−gui(1)* offers a file dialog.

PATH Path to a directory on disk. *cmake−gui(1)* offers a file dialog.

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

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

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

If the cache entry does not exist prior to the call or the **FORCE** option is given then the cache entry will be set to the given value.

NOTE:
The content of the cache variable will not be directly accessible if a normal variable of the same name already exists (see *rules of variable evaluation*). If policy *CMP0126* is set to **OLD**, any normal variable binding in the current scope will be removed.

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

Set Environment Variable

set(ENV{<variable>} [<value>])

Sets an *Environment Variable* to the given value. Subsequent calls of**$ENV{<v ariable>}** will return this new value.

This command affects only the current CMake process, not the process from which CMake was called, nor the system environment at large, nor the environment of subsequent build or test processes.

If no argument is given after **ENV{<variable>}** or if **<value>** is an empty string, then this command will clear any existing value of the environment variable.

Arguments after **<value>** are ignored. If extra arguments are found, then an author warning is issued.

See Also

- *unset()*

set_directory_properties

Set properties of the current directory and subdirectories.

```
set_directory_properties(PROPERTIES <prop1> <value1> [<prop2> <value2>] ...)
```

Sets properties of the current directory and its subdirectories in key−value pairs.

See also the *set_property(DIRECTORY)* command.

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

See Also

- *define_property()*
- *get_directory_property()*
- the more general *set_property()* command

set_property

Set a named property in a given scope.

```
set_property(<GLOBAL                      |
             DIRECTORY [<dir>]            |
             TARGET    [<target1> ...]    |
             SOURCE    [<src1> ...]
                       [DIRECTORY <dirs> ...]
                       [TARGET_DIRECTORY <targets> ...] |
             INSTALL   [<file1> ...]      |
```

```
 TEST      [<test1> ...]
           [DIRECTORY <dir>] |
 CACHE     [<entry1> ...]    >
[APPEND] [APPEND_STRING]
PROPERTY <name> [<value1> ...])
```

Sets one property on zero or more objects of a scope.

The first argument determines the scope in which the property is set. It must be one of the following:

GLOBAL
Scope is unique and does not accept a name.

DIRECTORY
Scope defaults to the current directory but other directories (already processed by CMake) may be named by full or relative path. Relative paths are treated as relative to the current source directory. See also the *set_directory_properties()* command.

Added in version 3.19: **<dir>** may reference a binary directory.

TARGET
Scope may name zero or more existing targets. See also the*set_tar get_properties()* command.

Alias Targets do not support setting target properties.

SOURCE
Scope may name zero or more source files. By default, source file properties are only visible to targets added in the same directory (**CMakeLists.txt**).

Added in version 3.18: Visibility can be set in other directory scopes using one or both of the following sub−options:

DIRECTORY <dirs>...
The source file property will be set in each of the **<dirs>** directories' scopes. CMake must already know about each of these directories, either by having added them through a call to *add_subdirectory()* or it being the top level source directory. Relative paths are treated as relative to the current source directory.

Added in version 3.19: **<dirs>** may reference a binary directory.

TARGET_DIRECTORY <targets>...
The source file property will be set in each of the directory scopes where any of the specified **<targets>** were created (the **<targets>** must therefore already exist).

See also the *set_source_files_properties()* command.

INSTALL
Added in version 3.1.

Scope may name zero or more installed file paths. These are made available to CPack to influence deployment.

Both the property key and value may use generator expressions. Specific properties may apply to installed files and/or directories.

Path components have to be separated by forward slashes, must be normalized and are case sensitive.

To reference the installation prefix itself with a relative path use **..**

Currently installed file properties are only defined for the WIX generator where the given paths are relative to the installation prefix.

TEST Scope is limited to the directory the command is called in. It may name zero or more existing tests. See also command *set_tests_properties()*.

Test property values may be specified using *generator expressions* for tests created by the *add_test(NAME)* signature.

Added in version 3.28: Visibility can be set in other directory scopes using the following sub−option:

DIRECTORY <dir>
The test property will be set in the **<dir>** directory's scope. CMake must already know about this directory, either by having added it through a call to *add_subdirectory()* or it being the top level source directory. Relative paths are treated as relative to the current source directory. **<dir>** may reference a binary directory.

CACHE
Scope must name zero or more existing cache entries.

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

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

NOTE:
The *GENERATED* source file property may be globally visible. See its documentation for details.

See Also
- *define_property()*
- *get_property()*
- The *cmake−properties(7)* manual for a list of properties in each scope.

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

```
site_name(variable)
```

On UNIX−like platforms, if the variable **HOSTNAME** is set, its value will be executed as a command expected to print out the host name, much like the **hostname** command−line tool.

string
String operations.

Synopsis

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

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

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

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

Generation
  string(ASCII <number>... <out-var>)
  string(HEX <string> <out-var>)
  string(CONFIGURE <string> <out-var> [...])
  string(MAKE_C_IDENTIFIER <string> <out-var>)
  string(RANDOM [<option>...] <out-var>)
  string(TIMESTAMP <out-var> [<format string>] [UTC])
  string(UUID <out-var> ...)

JSON
  string(JSON <out-var> [ERROR_VARIABLE <error-var>]
         {GET | TYPE | LENGTH | REMOVE}
         <json-string> <member|index> [<member|index> ...])
  string(JSON <out-var> [ERROR_VARIABLE <error-var>]
         MEMBER <json-string>
         [<member|index> ...] <index>)
  string(JSON <out-var> [ERROR_VARIABLE <error-var>]
         SET <json-string>
         <member|index> [<member|index> ...] <value>)
  string(JSON <out-var> [ERROR_VARIABLE <error-var>]
         EQUAL <json-string1> <json-string2>)
```

Search and Replace

Search and Replace With Plain Strings

string(FIND <string> <substring> <output_variable> [REVERSE])

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

The **string(FIND)** subcommand treats all strings as ASCII−only characters. The index stored in **<output_variable>** will also be counted in bytes, so strings containing multi−byte characters may lead to unexpected results.

string(REPLACE <match_string> <replace_string> <output_variable> <input> [<input>...])
Replace all occurrences of **<match_string>** in the **<input>** with **<replace_string>** and store the result in the **<output_variable>**.

Search and Replace With Regular Expressions

string(REGEX MATCH <regular_expression> <output_variable> <input> [<input>...])
Match the **<regular_expression>** once and store the match in the **<output_variable>**. All**<input>** arguments are concatenated before matching. Regular expressions are specified in the subsection just below.

string(REGEX MATCHALL <regular_expression> <output_variable> <input> [<input>...])
Match the **<regular_expression>** as many times as possible and store the matches in the **<output_variable>** as a list. All **<input>** arguments are concatenated before matching.

string(REGEX REPLACE <regular_expression> <replacement_expression> <output_variable> <input> [<input>...])
Match the **<regular_expression>** as many times as possible and substitute the **<replacement_expression>** for the match in the output. All **<input>** arguments are concatenated before matching.

The **<replacement_expression>** may refer to parenthesis−delimited subexpressions of the match using **\1**, **\2**, ..., **\9**. Note that two backslashes (**\\1**) are required in CMake code to get a backslash through argument parsing.

Regex Specification

The following characters have special meaning in regular expressions:

^ Matches at beginning of input

$ Matches at end of input

. Matches any single character

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

[ ] Matches any character(s) inside the brackets. To match a literal **]**, make it the first character, e.g., **[]ab]**.

[^] Matches any character(s) not inside the brackets. To not match a literal **]**, make it the first character, e.g., **[^]ab]**.

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

***** Matches preceding pattern zero or more times

+ Matches preceding pattern one or more times

? Matches preceding pattern zero or once only

| Matches a pattern on either side of the **|**

() Saves a matched subexpression, which can be referenced in the **REGEX REPLACE** operation.

Added in version 3.9: All regular expression−related commands, including e.g. *if(MA TCHES)*, save subgroup matches in the variables *CMAKE_MATCH_*<*n*> for **<n>** 0..9.

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

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

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

Manipulation

string(APPEND <string_variable> [<input>...])
Added in version 3.4.

Append all the **<input>** arguments to the string.

string(PREPEND <string_variable> [<input>...])
Added in version 3.10.

Prepend all the **<input>** arguments to the string.

string(CONCAT <output_variable> [<input>...])
Concatenate all the **<input>** arguments together and store the result in the named **<output_variable>**.

string(JOIN <glue> <output_variable> [<input>...])
Added in version 3.12.

Join all the **<input>** arguments together using the **<glue>** string and store the result in the named **<output_variable>**.

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

string(TOLOWER <string> <output_variable>)
Convert **<string>** to lower characters.

string(TOUPPER <string> <output_variable>)
Convert **<string>** to upper characters.

string(LENGTH <string> <output_variable>)
Store in an **<output_variable>** a given string's length in bytes. Note that this means if **<string>** contains multi−byte characters, the result stored in **<output_variable>** will *not* be the number of characters.

string(SUBSTRING <string> <begin> <length> <output_variable>)
Store in an **<output_variable>** a substring of a given **<string>**. If**<length>** is **−1** the remainder of the string starting at **<begin>** will be returned.

Changed in version 3.2: If **<string>** is shorter than **<length>** then the end of the string is used instead. Previous versions of CMake reported an error in this case.

Both **<begin>** and **<length>** are counted in bytes, so care must be exercised if **<string>** could contain multi−byte characters.

string(STRIP <string> <output_variable>)
Store in an **<output_variable>** a substring of a given **<string>** with leading and trailing spaces removed.

string(GENEX_STRIP <string> <output_variable>)
Added in version 3.1.

Strip any *generator expressions* from the input **<string>** and store the result in the **<output_variable>**.

string(REPEAT <string> <count> <output_variable>)
Added in version 3.15.

Produce the output string as the input **<string>** repeated **<count>** times.

Comparison

string(COMPARE LESS <string1> <string2> <output_variable>)

string(COMPARE GREATER <string1> <string2> <output_variable>)

string(COMPARE EQUAL <string1> <string2> <output_variable>)

string(COMPARE NOTEQUAL <string1> <string2> <output_variable>)

string(COMPARE LESS_EQUAL <string1> <string2> <output_variable>)

string(COMPARE GREATER_EQUAL <string1> <string2> <output_variable>)
Compare the strings and store true or false in the **<output_variable>**.

Added in version 3.7: Added the **LESS_EQUAL** and **GREATER_EQUAL** options.

Hashing

string(<HASH> <output_variable> <input>)
Compute a cryptographic hash of the **<input>** string. The supported **<HASH>** algorithm names are:

MD5 Message−Digest Algorithm 5, RFC 1321.

SHA1 US Secure Hash Algorithm 1, RFC 3174.

SHA224
US Secure Hash Algorithms, RFC 4634.

SHA256
US Secure Hash Algorithms, RFC 4634.

SHA384
US Secure Hash Algorithms, RFC 4634.

SHA512
US Secure Hash Algorithms, RFC 4634.

SHA3_224
Keccak SHA−3.

SHA3_256
Keccak SHA−3.

SHA3_384
Keccak SHA−3.

SHA3_512
Keccak SHA−3.

Added in version 3.8: Added the **SHA3_*** hash algorithms.

Generation

string(ASCII <number> [<number> ...] <output_variable>)
Convert all numbers into corresponding ASCII characters.

string(HEX <string> <output_variable>)
Added in version 3.18.

Convert each byte in the input **<string>** to its hexadecimal representation and store the concatenated hex digits in the **<output_variable>**. Letters in the output (**a** through **f**) are in lowercase.

string(CONFIGURE <string> <output_variable> [@ONLY] [ESCAPE_QUOTES])
Transform a **<string>** like *configure_file()* transforms a file.

string(MAKE_C_IDENTIFIER <string> <output_variable>)
Convert each non−alphanumeric character in the input **<string>** to an underscore and store the result in the **<output_variable>**. If the first character of the**<string>** is a digit, an underscore will also be prepended to the result.

string(RANDOM [LENGTH <length>] [ALPHABET <alphabet>] [RANDOM_SEED <seed>] <output_variable>)
Return a random string of given **<length>** consisting of characters from the given **<alphabet>**. Default length is 5 characters and default alphabet is all numbers and upper and lower case letters. If an integer **RANDOM_SEED** is given, its value will be used to seed the random number generator.

string(TIMESTAMP <output_variable> [<format_string>] [UTC])
Write a string representation of the current date and/or time to the **<output_variable>**.

If the command is unable to obtain a timestamp, the **<output_variable>** will be set to the empty string **""**.

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

The optional **<format_string>** may contain the following format specifiers:

%% Added in version 3.8.

A literal percent sign (%).

%d The day of the current month (01−31).

%H The hour on a 24−hour clock (00−23).

%I The hour on a 12−hour clock (01−12).

%j The day of the current year (001−366).

%m The month of the current year (01−12).

%b Added in version 3.7.

Abbreviated month name (e.g. Oct).

%B Added in version 3.10.

Full month name (e.g. October).

%M The minute of the current hour (00−59).

%s Added in version 3.6.

Seconds since midnight (UTC) 1−Jan−1970 (UNIX time).

%S The second of the current minute. 60 represents a leap second. (00−60)

%f Added in version 3.23.

The microsecond of the current second (000000−999999).

%U The week number of the current year (00−53).

%V Added in version 3.22.

The ISO 8601 week number of the current year (01−53).

%w The day of the current week. 0 is Sunday. (0−6)

%a Added in version 3.7.

Abbreviated weekday name (e.g. Fri).

%A Added in version 3.10.

Full weekday name (e.g. Friday).

%y The last two digits of the current year (00−99).

%Y The current year.

%z Added in version 3.26.

The offset of the time zone from UTC, in hours and minutes, with format **+hhmm** or **−hhmm**.

%Z Added in version 3.26.

The time zone name.

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

If no explicit **<format_string>** is given, it will default to:

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

Added in version 3.8: If the **SOURCE_DATE_EPOCH** environment variable is set, its value will be used instead of the current time. See - *https://reproducible-builds.org/specs/source-date-epoch/* for details.

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

Added in version 3.1.

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

JSON

Added in version 3.19.

Functionality for querying a JSON string.

NOTE:

In each of the following JSON-related subcommands, if the optional **ERROR_VARIABLE** argument is given, errors will be reported in **<error-variable>** and the **<out-var>** will be set to **<member|index>-[<member|index>...]-NOTFOUND** with the path elements up to the point where the error occurred, or just **NOTFOUND** if there is no relevant path. If an error occurs but the **ERROR_VARIABLE** option is not present, a fatal error message is generated. If no error occurs, the **<error-variable>** will be set to **NOTFOUND**.

In the following subcommands, the **<json-string>** argument should be written as a *Quoted Argument* to ensure the entire JSON string is passed as a single argument even if it contains semicolons.

string(JSON <out-var> [ERROR_VARIABLE <error-variable>] GET <json-string> <member|index> [<member|index> ...])

Get an element from **<json-string>** at the location given by the list of **<member|index>** arguments. Array and object elements will be returned as a JSON string. Boolean elements will be returned as **ON** or **OFF**. Null elements will be returned as an empty string. Number and string types will be returned as strings.

string(JSON <out-var> [ERROR_VARIABLE <error-variable>] TYPE <json-string> <member|index> [<member|index> ...])

Get the type of an element in **<json-string>** at the location given by the list of **<member|index>** arguments. The **<out-var>** will be set to one of **NULL**, **NUMBER**, **STRING**, **BOOLEAN**, **ARRAY**, or **OBJECT**.

string(JSON <out-var> [ERROR_VARIABLE <error-var>] MEMBER <json-string> [<member|index> ...] <index>)

Get the name of the **<index>**-th member in **<json-string>** at the location given by the list of **<member|index>** arguments. Requires an element of object type.

string(JSON <out-var> [ERROR_VARIABLE <error-variable>] LENGTH <json-string> [<member|index> ...])

Get the length of an element in **<json-string>** at the location given by the list of

<member|index> arguments. Requires an element of array or object type.

string(JSON <out−var> [ERROR_VARIABLE <error−variable>] REMOVE <json−string> <member|index> [<member|index> ...])
Remove an element from **<json−string>** at the location given by the list of **<member|index>** arguments. The JSON string without the removed element will be stored in **<out−var>**.

string(JSON <out−var> [ERROR_VARIABLE <error−variable>] SET <json−string> <member|index> [<member|index> ...] <value>)
Set an element in **<json−string>** at the location given by the list of **<member|index>** arguments to **<value>**. The contents of**<v alue>** should be valid JSON. If **<json−string>** is an array, **<value>** can be appended to the end of the array by using a number greater or equal to the array length as the **<member|index>** argument.

string(JSON <out−var> [ERROR_VARIABLE <error−var>] EQUAL <json−string1> <json−string2>)
Compare the two JSON objects given by **<json−string1>** and **<json−string2>** for equality. The contents of **<json−string1>** and **<json−string2>** should be valid JSON. The **<out−var>** will be set to a true value if the JSON objects are considered equal, or a false value otherwise.

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

Unset Normal Variable or Cache Entry

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

Removes a normal variable from the current scope, causing it to become undefined. If **CACHE** is present, then a cache variable is removed instead of a normal variable.

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

NOTE:
When evaluating *Variable References* of the form **${VAR}**, CMake first searches for a normal variable with that name. If no such normal variable exists, CMake will then search for a cache entry with that name. Because of this,**unsetting a normal v ariable can expose a cache variable that was previously hidden**. To force a variable reference of the form **${VAR}** to return an empty string, use **set(<variable> "")**, which clears the normal variable but leaves it defined.

Unset Environment Variable

```
unset(ENV{<variable>})
```

Removes **<variable>** from the currently available *Environment Variables*. Subsequent calls of **$ENV{<variable>}** will return the empty string.

This command affects only the current CMake process, not the process from which CMake was called, nor the system environment at large, nor the environment of subsequent build or test processes.

See Also
- *set()*

variable_watch
Watch the CMake variable for change.

```
variable_watch(<variable> [<command>])
```

If the specified **<variable>** changes and no **<command>** is given, a message will be printed to inform about the change.

If **<command>** is given, this command will be executed instead. The command will receive the following arguments: **COMMAND(<variable> <access> <value> <current_list_file> <stack>)**

<variable>
Name of the variable being accessed.

<access>
One of **READ_ACCESS**, **UNKNOWN_READ_ACCESS**, **MODIFIED_ACCESS**, **UNKNOWN_MODIFIED_ACCESS**, or **REMOVED_ACCESS**. The**UNKNO WN_** values are only used when the variable has never been set. Once set, they are never used again during the same CMake run, even if the variable is later unset.

<value>
The value of the variable. On a modification, this is the new (modified) value of the variable. On removal, the value is empty.

<current_list_file>
Full path to the file doing the access.

<stack>
List of absolute paths of all files currently on the stack of file inclusion, with the bottom−most file first and the currently processed file (that is, **current_list_file**) last.

Note that for some accesses such as *list(APPEND)*, the watcher is executed twice, first with a read access and then with a write one. Also note that an *if(DEFINED)* query on the variable does not register as an access and the watcher is not executed.

Only non−cache variables can be watched using this command. Access to cache variables is never watched. However, the existence of a cache variable **var** causes accesses to the non−cache variable **var** to not use the **UNKNOWN_** prefix, even if a non−cache variable **var** has never existed.

while
Evaluate a group of commands while a condition is true

```
while(<condition>)
  <commands>
endwhile()
```

All commands between while and the matching *endwhile()* are recorded without being invoked. Once the *endwhile()* is evaluated, the recorded list of commands is invoked as long as the **<condition>** is true.

The **<condition>** has the same syntax and is evaluated using the same logic as described at length for the *if()* command.

The commands *break()* and *continue()* provide means to escape from the normal control flow.

Per legacy, the *endwhile()* command admits an optional **<condition>** argument. If used, it must be a verbatim repeat of the argument of the opening **while** command.

See Also
- *break()*
- *continue()*
- *foreach()*
- *endwhile()*

PROJECT COMMANDS

These commands are available only in CMake projects.

add_compile_definitions
Added in version 3.12.

Add preprocessor definitions to the compilation of source files.

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

Adds preprocessor definitions to the compiler command line.

The preprocessor definitions are added to the *COMPILE_DEFINITIONS* directory property for the current **CMakeLists** file. They are also added to the *COMPILE_DEFINITIONS* target property for each target in the current **CMakeLists** file.

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

Added in version 3.26: Any leading **−D** on an item will be removed.

Arguments to **add_compile_definitions** may use generator expressions with the syntax **$<...>**. See the *cmake−generator−expressions(7)* manual for available expressions. See the*cmak e−buildsystem(7)* manual for more on defining buildsystem properties.

See Also
- The command *target_compile_definitions()* adds target−specific definitions.

add_compile_options
Add options to the compilation of source files.

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

Adds options to the *COMPILE_OPTIONS* directory property. These options are used when compiling targets from the current directory and below.

NOTE:
These options are not used when linking. See the *add_link_options()* command for that.

Arguments
Arguments to **add_compile_options** may use generator expressions with the syntax **$<...>**. See the *cmake−generator−expressions(7)* manual for available expressions. See the*cmak e−buildsystem(7)* manual for more on defining buildsystem properties.

Option De−duplication
The final set of options used for a target is constructed by accumulating options from the current target and the usage requirements of its dependencies. The set of options is de−duplicated to avoid repetition.

Added in version 3.12: While beneficial for individual options, the de−duplication step can break up option groups. For example, **−option A −option B** becomes **−option A B**. One may specify a group of options using shell−like quoting along with a **SHELL:** prefix. The **SHELL:** prefix is dropped, and the rest of the option string is parsed using the *separate_arguments()* **UNIX_COMMAND** mode. For example, **"SHELL:−option A" "SHELL:−option B"** becomes **−option A −option B**.

Example

Since different compilers support different options, a typical use of this command is in a compiler−specific conditional clause:

```
if (MSVC)
    # warning level 4
    add_compile_options(/W4)
else()
    # additional warnings
    add_compile_options(−Wall −Wextra −Wpedantic)
endif()
```

To set per−language options, use the *$<COMPILE_LANGUAGE>* or *$<COMPILE_LANGUAGE:languages>* generator expressions.

See Also

- This command can be used to add any options. However, for adding preprocessor definitions and include directories it is recommended to use the more specific commands *add_compile_definitions()* and *include_directories()*.
- The command *target_compile_options()* adds target−specific options.
- This command adds compile options for all languages. Use the *COMPILE_LANGUAGE* generator expression to specify per−language compile options.
- The source file property *COMPILE_OPTIONS* adds options to one source file.
- *add_link_options()* adds options for linking.
- *CMAKE_<LANG>_FLAGS* and *CMAKE_<LANG>_FLAGS_<CONFIG>* add language−wide flags passed to all invocations of the compiler. This includes invocations that drive compiling and those that drive linking.

add_custom_command

Add a custom build rule to the generated build system.

There are two main signatures for **add_custom_command**.

Generating Files

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

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

This defines a command to generate specified **OUTPUT** file(s). A target created in the same directory (**CMakeLists.txt** file) that specifies any output of the custom command as a source file is given a rule to generate the file using the command at build time.

Do not list the output in more than one independent target that may build in parallel or the instances of the rule may conflict. Instead, use the *add_custom_target()* command to drive the command and make the other targets depend on that one. See the *Example: Generating Files for Multiple Targets* below.

The options are:

APPEND

Append the **COMMAND** and **DEPENDS** option values to the custom command for the first output specified. There must have already been a previous call to this command with the same output.

If the previous call specified the output via a generator expression, the output specified by the current call must match in at least one configuration after evaluating generator expressions. In this case, the appended commands and dependencies apply to all configurations.

The **COMMENT**, **MAIN_DEPENDENCY**, and **WORKING_DIRECTORY** options are currently ignored when **APPEND** is given, but may be used in the future.

BYPRODUCTS

Added in version 3.2.

Specify the files the command is expected to produce but whose modification time may or may not be newer than the dependencies. If a byproduct name is a relative path it will be interpreted relative to the build tree directory corresponding to the current source directory. Each byproduct file will be marked with the *GENERATED* source file property automatically.

See policy CMP0058 for the motivation behind this feature.

Explicit specification of byproducts is supported by the *Ninja* generator to tell the **ninja** build tool how to regenerate byproducts when they are missing. It is also useful when other build rules (e.g. custom commands) depend on the byproducts. Ninja requires a build rule for any generated file on which another rule depends even if there are order−only dependencies to ensure the byproducts will be available before their dependents build.

The *Makefile Generators* will remove **BYPRODUCTS** and other *GENERATED* files during **make clean**.

This keyword cannot be used with **APPEND** (see policy *CMP0175*). All byproducts must be set in the first call to **add_custom_command(OUTPUT...)** for the output files.

Added in version 3.20: Arguments to **BYPRODUCTS** may use a restricted set of *generator expressions*. *Target−dependent expressions* are not permitted.

Changed in version 3.28: In targets using *File Sets*, custom command byproducts are now considered private unless they are listed in a non−private file set. See policy *CMP0154*.

COMMAND

Specify the command−line(s) to execute at build time. At least one **COMMAND** would normally be given, but certain patterns may omit it, such as adding commands in separate calls using

APPEND.

If more than one **COMMAND** is specified, they will be executed in order, but *not* necessarily composed into a stateful shell or batch script. To run a full script, use the *configure_file()* command or the *file(GENERATE)* command to create it, and then specify a **COMMAND** to launch it.

The optional **ARGS** argument is for backward compatibility and will be ignored.

If **COMMAND** specifies an executable target name (created by the *add_executable()* command), it will automatically be replaced by the location of the executable created at build time if either of the following is true:

- The target is not being cross−compiled (i.e. the *CMAKE_CROSSCOMPILING* variable is not set to true).
- Added in version 3.6: The target is being cross−compiled and an emulator is provided (i.e. its *CROSSCOMPILING_EMULATOR* target property is set). In this case, the contents of *CROSSCOMPILING_EMULATOR* will be prepended to the command before the location of the target executable.

If neither of the above conditions are met, it is assumed that the command name is a program to be found on the **PATH** at build time.

Arguments to **COMMAND** may use *generator expressions*. Use the*T ARGET_FILE* generator expression to refer to the location of a target later in the command line (i.e. as a command argument rather than as the command to execute).

Whenever one of the following target based generator expressions are used as a command to execute or is mentioned in a command argument, a target−level dependency will be added automatically so that the mentioned target will be built before any target using this custom command (see policy *CMP0112*).

- **TARGET_FILE**
- **TARGET_LINKER_FILE**
- **TARGET_SONAME_FILE**
- **TARGET_PDB_FILE**

This target−level dependency does NOT add a file−level dependency that would cause the custom command to re−run whenever the executable is recompiled. List target names with the **DEPENDS** option to add such file−level dependencies.

COMMENT

Display the given message before the commands are executed at build time. This will be ignored if **APPEND** is given, although a future version may use it.

Added in version 3.26: Arguments to **COMMENT** may use *generator expressions*.

DEPENDS

Specify files on which the command depends. Each argument is converted to a dependency as follows:

1. If the argument is the name of a target (created by the *add_custom_target()*, *add_executable()*, or *add_library()* command) a target−level dependency is created to make sure the target is built before any target using this custom command. Additionally, if the target is an executable or library, a file−level dependency is created to cause the custom command to re−run whenever the

target is recompiled.

2. If the argument is an absolute path, a file−level dependency is created on that path.
3. If the argument is the name of a source file that has been added to a target or on which a source file property has been set, a file−level dependency is created on that source file.
4. If the argument is a relative path and it exists in the current source directory, a file−level dependency is created on that file in the current source directory.
5. Otherwise, a file−level dependency is created on that path relative to the current binary directory.

If any dependency is an **OUTPUT** of another custom command in the same directory (**CMakeLists.txt** file), CMake automatically brings the other custom command into the target in which this command is built.

Added in version 3.16: A target−level dependency is added if any dependency is listed as **BYPRODUCTS** of a target or any of its build events in the same directory to ensure the byproducts will be available.

If **DEPENDS** is not specified, the command will run whenever the **OUTPUT** is missing; if the command does not actually create the **OUTPUT**, the rule will always run.

Added in version 3.1: Arguments to **DEPENDS** may use *generator expressions*.

COMMAND_EXPAND_LISTS
Added in version 3.8.

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

This keyword cannot be used with **APPEND** (see policy *CMP0175*). If the appended commands need this option to be set, it must be set on the first call to **add_custom_command(OUTPUT...)** for the output files.

CODEGEN
Added in version 3.31.

Adds the custom command to a global **codegen** target that can be used to execute the custom command while avoiding the majority of the build graph.

This option is supported only by *Ninja Generators* and *Makefile Generators*, and is ignored by other generators. Furthermore, this option is allowed only if policy *CMP0171* is set to **NEW**.

This keyword cannot be used with **APPEND** (see policy *CMP0175*). It can only be set on the first call to **add_custom_command(OUTPUT...)** for the output files.

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

supported only for Makefile generators and will be ignored by other generators.

NOTE:
This option cannot be specified at the same time as **DEPFILE** option.

JOB_POOL
Added in version 3.15.

Specify a *pool* for the *Ninja* generator. Incompatible with **USES_TERMINAL**, which implies the **console** pool. Using a pool that is not defined by *JOB_POOLS* causes an error by ninja at build time.

This keyword cannot be used with **APPEND** (see policy *CMP0175*). Job pools can only be specified in the first call to **add_custom_command(OUTPUT...)** for the output files.

JOB_SERVER_AWARE
Added in version 3.28.

Specify that the command is GNU Make job server aware.

For the *Unix Makefiles*, *MSYS Makefiles*, and *MinGW Makefiles* generators this will add the + prefix to the recipe line. See the *GNU Make Documentation* for more information.

This option is silently ignored by other generators.

This keyword cannot be used with **APPEND** (see policy *CMP0175*). Job server awareness can only be specified in the first call to **add_custom_command(OUTPUT...)** for the output files.

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

This option is currently ignored if **APPEND** is given, but a future version may use it.

OUTPUT
Specify the output files the command is expected to produce. Each output file will be marked with the *GENERATED* source file property automatically. If the output of the custom command is not actually created as a file on disk it should be marked with the *SYMBOLIC* source file property.

If an output file name is a relative path, its absolute path is determined by interpreting it relative to:

1. the build directory corresponding to the current source directory (*CMAKE_CURRENT_BINARY_DIR*), or
2. the current source directory (*CMAKE_CURRENT_SOURCE_DIR*).

The path in the build directory is preferred unless the path in the source tree is mentioned as an absolute source file path elsewhere in the current directory.

The output file path may not contain < or > characters.

Added in version 3.20: Arguments to **OUTPUT** may use a restricted set of *generator expressions*. *Target−dependent expressions* are not permitted.

Changed in version 3.28: In targets using *File Sets*, custom command outputs are now considered private unless they are listed in a non−private file set. See policy *CMP0154*.

Changed in version 3.30: The output file path may now use # characters, except when using the *Borland Makefiles* generator.

USES_TERMINAL
Added in version 3.2.

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

This keyword cannot be used with **APPEND** (see policy *CMP0175*). If the appended commands need access to the terminal, it must be set on the first call to **add_custom_command(OUTPUT...)** for the output files.

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

This keyword cannot be used with **APPEND** (see policy *CMP0175*). If the appended commands need to be treated as **VERBATIM**, it must be set on the first call to **add_custom_command(OUTPUT...)** for the output files.

WORKING_DIRECTORY
Execute the command with the given current working directory. If it is a relative path, it will be interpreted relative to the build tree directory corresponding to the current source directory.

This option is currently ignored if **APPEND** is given, but a future version may use it.

Added in version 3.13: Arguments to **WORKING_DIRECTORY** may use *generator expressions*.

DEPFILE
Added in version 3.7.

Specify a depfile which holds dependencies for the custom command. It is usually emitted by the custom command itself. This keyword may only be used if the generator supports it, as detailed below.

The expected format, compatible with what is generated by **gcc** with the option **−M**, is independent of the generator or platform.

The formal syntax, as specified using *BNF* notation with the regular extensions, is the following:

depfile ::= *rule**
rule ::= *targets* (':' (*separator dependencies*?)?)? *eol*
targets ::= *target* (*separator target*)* *separator**
target ::= *pathname*

dependencies ::= *dependency* (*separator dependency*)* *separator**
dependency ::= *pathname*
separator ::= (*space* | *line_continue*)+
line_continue ::= '\' *eol*
space ::= ' ' | '\t'
pathname ::= *character+*
character ::= *std_character* | *dollar* | *hash* | *whitespace*
std_character ::= <any character except '$', '#' or ' '>
dollar ::= '$$'
hash ::= '\#'
whitespace ::= '\ '
eol ::= '\r'? '\n'

NOTE:
As part of **pathname**, any slash and backslash is interpreted as a directory separator.

Added in version 3.7: The *Ninja* generator supports **DEPFILE** since the keyword was first added.

Added in version 3.17: Added the *Ninja Multi−Config* generator, which included support for the **DEPFILE** keyword.

Added in version 3.20: Added support for *Makefile Generators*.

NOTE:
DEPFILE cannot be specified at the same time as the **IMPLICIT_DEPENDS** option for *Makefile Generators*.

Added in version 3.21: Added support for *Visual Studio Generators* with VS 2012 and above, and for the *Xcode* generator. Support for*g enerator expressions* was also added.

Added in version 3.29: The *Ninja Generators* will now incorporate the dependencies into its "deps log" database if the file is not listed in**OUTPUTS** or **BYPR ODUCTS**.

Using **DEPFILE** with generators other than those listed above is an error.

If the **DEPFILE** argument is relative, it should be relative to *CMAKE_CURRENT_BINARY_DIR*, and any relative paths inside the **DEPFILE** should also be relative to *CMAKE_CURRENT_BINARY_DIR*. See policy *CMP0116*, which is always **NEW** for *Makefile Generators*, *Visual Studio Generators*, and the *Xcode* generator.

This keyword cannot be used with **APPEND** (see policy *CMP0175*). Depfiles can only be set on the first call to **add_custom_command(OUTPUT...)** for the output files.

DEPENDS_EXPLICIT_ONLY
Added in version 3.27.

Indicates that the command's **DEPENDS** argument represents all files required by the command and

implicit dependencies are not required.

Without this option, if any target uses the output of the custom command, CMake will consider that target's dependencies as implicit dependencies for the custom command in case this custom command requires files implicitly created by those targets.

This option can be enabled on all custom commands by setting *CMAKE_ADD_CUSTOM_COMMAND_DEPENDS_EXPLICIT_ONLY* to **ON**.

This keyword cannot be used with **APPEND** (see policy *CMP0175*). It can only be set on the first call to **add_custom_command(OUTPUT...)** for the output files.

Only the *Ninja Generators* actually use this information to remove unnecessary implicit dependencies.

See also the *OPTIMIZE_DEPENDENCIES* target property, which may provide another way for reducing the impact of target dependencies in some scenarios.

Examples: Generating Files

Custom commands may be used to generate source files. For example, the code:

```
add_custom_command(
  OUTPUT out.c
  COMMAND someTool -i ${CMAKE_CURRENT_SOURCE_DIR}/in.txt
                   -o out.c
  DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/in.txt
  VERBATIM)
add_library(myLib out.c)
```

adds a custom command to run **someTool** to generate **out.c** and then compile the generated source as part of a library. The generation rule will re−run whenever **in.txt** changes.

Added in version 3.20: One may use generator expressions to specify per−configuration outputs. For example, the code:

```
add_custom_command(
  OUTPUT "out-$<CONFIG>.c"
  COMMAND someTool -i ${CMAKE_CURRENT_SOURCE_DIR}/in.txt
                   -o "out-$<CONFIG>.c"
                   -c "$<CONFIG>"
  DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/in.txt
  VERBATIM)
add_library(myLib "out-$<CONFIG>.c")
```

adds a custom command to run **someTool** to generate **out−<config>.c**, where**<config>** is the b uild configuration, and then compile the generated source as part of a library.

Added in version 3.31: Use the **CODEGEN** option to add a custom command's outputs to the builtin **codegen** target. This is useful to make generated code available for
static analysis without building the entire project. For example:

```
add_executable(someTool someTool.c)

add_custom_command(
  OUTPUT out.c
```

```
  COMMAND someTool -o out.c
  CODEGEN)

add_library(myLib out.c)
```

A user may build the **codegen** target to generate **out.c**. **someTool** is built as dependency, but **myLib** is not built at all.

Example: Generating Files for Multiple Targets

If multiple independent targets need the same custom command output, it must be attached to a single custom target on which they all depend. Consider the following example:

```
add_custom_command(
  OUTPUT table.csv
  COMMAND makeTable -i ${CMAKE_CURRENT_SOURCE_DIR}/input.dat
                    -o table.csv
  DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/input.dat
  VERBATIM)
add_custom_target(generate_table_csv DEPENDS table.csv)

add_custom_command(
  OUTPUT foo.cxx
  COMMAND genFromTable -i table.csv -case foo -o foo.cxx
  DEPENDS table.csv           # file-level dependency
          generate_table_csv  # target-level dependency
  VERBATIM)
add_library(foo foo.cxx)

add_custom_command(
  OUTPUT bar.cxx
  COMMAND genFromTable -i table.csv -case bar -o bar.cxx
  DEPENDS table.csv           # file-level dependency
          generate_table_csv  # target-level dependency
  VERBATIM)
add_library(bar bar.cxx)
```

Output **foo.cxx** is needed only by target **foo** and output **bar.cxx** is needed only by target **bar**, but *both* targets need **table.csv**, transitively. Since**f oo** and **bar** are independent targets that may build concurrently, we prevent them from racing to generate **table.csv** by placing its custom command in a separate target, **generate_table_csv**. The custom commands generating**f oo.cxx** and **bar.cxx** each specify a target-level dependency on **generate_table_csv**, so the targets using them, **foo** and **bar**, will not build until after target **generate_table_csv** is built.

Build Events

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

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

```
                [VERBATIM]
                [COMMAND_EXPAND_LISTS]
                [USES_TERMINAL])
```

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

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

PRE_BUILD
: This option has unique behavior for the *Visual Studio Generators*. When using one of the Visual Studio generators, the command will run before any other rules are executed within the target. With all other generators, this option behaves the same as **PRE_LINK** instead. Because of this, it is recommended to avoid using **PRE_BUILD** except when it is known that a Visual Studio generator is being used.

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

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

Projects should always specify one of the above three keywords when using the **TARGET** form. See policy *CMP0175*.

All other keywords shown in the signature above have the same meaning as they do for the *add_custom_command(OUTPUT)* form of the command. At least one **COMMAND** must be given, see policy *CMP0175*.

NOTE:
: Because generator expressions can be used in custom commands, it is possible to define **COMMAND** lines or whole custom commands which evaluate to empty strings for certain configurations. For *Visual Studio Generators* these command lines or custom commands will be omitted for the specific configuration and no "empty−string−command" will be added.

  This allows adding individual build events for every configuration.

Added in version 3.21: Support for target−dependent generator expressions.

Added in version 3.29: The **<target>** may be an *ALIAS target*.

Examples: Build Events

A **POST_BUILD** event may be used to post−process a binary after linking. For example, the code:

```
add_executable(myExe myExe.c)
add_custom_command(
  TARGET myExe POST_BUILD
  COMMAND someHasher -i "$<TARGET_FILE:myExe>"
                     -o "$<TARGET_FILE:myExe>.hash"
  VERBATIM)
```

will run **someHasher** to produce a **.hash** file next to the executable after linking.

Added in version 3.20: One may use generator expressions to specify per−configuration byproducts. For example, the code:

```
add_library(myPlugin MODULE myPlugin.c)
add_custom_command(
  TARGET myPlugin POST_BUILD
  COMMAND someHasher -i "$<TARGET_FILE:myPlugin>"
                     --as-code "myPlugin-hash-$<CONFIG>.c"
  BYPRODUCTS "myPlugin-hash-$<CONFIG>.c"
  VERBATIM)
add_executable(myExe myExe.c "myPlugin-hash-$<CONFIG>.c")
```

will run **someHasher** after linking **myPlugin**, e.g. to produce a **.c** file containing code to check the hash of **myPlugin** that the **myExe** executable can use to verify it before loading.

Ninja Multi−Config

Added in version 3.20: **add_custom_command** supports the *Ninja Multi−Config* generator's cross−config capabilities. See the generator documentation for more information.

See Also

- *add_custom_target()*

add_custom_target

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

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

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

The options are:

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

BYPRODUCTS

Added in version 3.2.

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

See policy CMP0058 for the motivation behind this feature.

Explicit specification of byproducts is supported by the *Ninja* generator to tell the **ninja** build tool how to regenerate byproducts when they are missing. It is also useful when other build rules (e.g. custom commands) depend on the byproducts. Ninja requires a build rule for any generated file on which another rule depends even if there are order−only dependencies to ensure the byproducts will be available before their dependents build.

The *Makefile Generators* will remove **BYPRODUCTS** and other *GENERATED* files during **make clean**.

Added in version 3.20: Arguments to **BYPRODUCTS** may use a restricted set of *generator expressions*. *Target−dependent expressions* are not permitted.

Changed in version 3.28: In custom targets using *File Sets*, byproducts are now considered private unless they are listed in a non−private file set. See policy *CMP0154*.

COMMAND

Specify the command−line(s) to execute at build time. If more than one **COMMAND** is specified they will be executed in order, but *not* necessarily composed into a stateful shell or batch script. (To run a full script, use the *configure_file()* command or the *file(GENERATE)* command to create it, and then specify a **COMMAND** to launch it.)

If **COMMAND** specifies an executable target name (created by the *add_executable()* command), it will automatically be replaced by the location of the executable created at build time if either of the following is true:

- The target is not being cross−compiled (i.e. the *CMAKE_CROSSCOMPILING* variable is not set to true).
- Added in version 3.6: The target is being cross−compiled and an emulator is provided (i.e. its *CROSSCOMPILING_EMULATOR* target property is set). In this case, the contents of *CROSSCOMPILING_EMULATOR* will be prepended to the command before the location of the target executable.

If neither of the above conditions are met, it is assumed that the command name is a program to be found on the **PATH** at build time.

Arguments to **COMMAND** may use *generator expressions*. Use the*T ARGET_FILE* generator expression to refer to the location of a target later in the command line (i.e. as a command argument rather than as the command to execute).

Whenever one of the following target based generator expressions are used as a command to execute or is mentioned in a command argument, a target−level dependency will be added automatically so that the mentioned target will be built before this custom target (see policy *CMP0112*).

- **TARGET_FILE**
- **TARGET_LINKER_FILE**
- **TARGET_SONAME_FILE**
- **TARGET_PDB_FILE**

The command and arguments are optional and if not specified an empty target will be created.

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

Added in version 3.26: Arguments to **COMMENT** may use *generator expressions*.

DEPENDS
Reference files and outputs of custom commands created with *add_custom_command()* command calls in the same directory (**CMakeLists.txt** file). They will be brought up to date when the target is built.

Changed in version 3.16: A target−level dependency is added if any dependency is a byproduct of a target or any of its build events in the same directory to ensure the byproducts will be available before this target is built.

Use the *add_dependencies()* command to add dependencies on other targets.

COMMAND_EXPAND_LISTS
Added in version 3.8.

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

JOB_POOL
Added in version 3.15.

Specify a *pool* for the *Ninja* generator. Incompatible with **USES_TERMINAL**, which implies the **console** pool. Using a pool that is not defined by *JOB_POOLS* causes an error by ninja at build time.

JOB_SERVER_AWARE
Added in version 3.28.

Specify that the command is GNU Make job server aware.

For the *Unix Makefiles*, *MSYS Makefiles*, and *MinGW Makefiles* generators this will add the **+** prefix to the recipe line. See the *GNU Make Documentation* for more information.

This option is silently ignored by other generators.

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

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

USES_TERMINAL
Added in version 3.2.

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

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

Added in version 3.13: Arguments to **WORKING_DIRECTORY** may use *generator expressions*.

Ninja Multi−Config
Added in version 3.20: **add_custom_target** supports the *Ninja Multi−Config* generator's cross−config capabilities. See the generator documentation for more information.

See Also
- *add_custom_command()*

add_definitions
Add **−D** define flags to the compilation of source files.

```
add_definitions(−DFOO −DBAR ...)
```

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

NOTE:
This command has been superseded by alternatives:
- Use *add_compile_definitions()* to add preprocessor definitions.
- Use *include_directories()* to add include directories.
- Use *add_compile_options()* to add other options.

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

See Also
- The *cmake−buildsystem(7)* manual for more on defining buildsystem properties.

add_dependencies
Add a dependency between top−level targets.

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

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

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

Added in version 3.3: Allow adding dependencies to interface libraries.

Added in version 3.8: Dependencies will populate the *MANUALLY_ADDED_DEPENDENCIES* property of **<target>**.

Changed in version 3.9: The *Ninja Generators* use weaker ordering than other generators in order to improve available concurrency. They only guarantee that the dependencies' custom commands are finished before sources in **<target>** start compiling; this ensures generated sources are available.

See Also

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

add_executable

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

Normal Executables

add_executable(<name> <options>... <sources>...)

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

The options are:

WIN32

Set the *WIN32_EXECUTABLE* target property automatically. See documentation of that target property for details.

MACOSX_BUNDLE

Set the *MACOSX_BUNDLE* target property automatically. See documentation of that target property for details.

EXCLUDE_FROM_ALL

Set the *EXCLUDE_FROM_ALL* target property automatically. See documentation of that target property for details.

The **<name>** corresponds to the logical target name and must be globally unique within a project. The actual file name of the executable built is constructed based on conventions of the native platform (such as **<name>.exe** or just **<name>**).

Added in version 3.1: Source arguments to **add_executable** may use "generator expressions" with the syntax **$<...>**. See the*cmak e−generator−expressions(7)* manual for available expressions.

Added in version 3.11: The source files can be omitted if they are added later using *target_sources()*.

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

See the *cmake−buildsystem(7)* manual for more on defining buildsystem properties.

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

Imported Executables

add_executable(<name> IMPORTED [GLOBAL])

Add an *IMPORTED executable target* to reference an executable file located outside the project. The target name may be referenced like any target built within the project, except that by default it is visible only in the directory in which it is created, and below.

The options are:

GLOBAL

Make the target name globally visible.

No rules are generated to build imported targets, and the *IMPORTED* target property is **True**. Imported executables are useful for convenient reference from commands like *add_custom_command()*.

Details about the imported executable are specified by setting properties whose names begin in **IMPORTED_**. The most important such property is*IMPORTED_LOCA TION* (and its per−configuration version *IMPORTED_LOCATION_<CONFIG>*) which specifies the location of the main executable file on disk. See documentation of the**IMPOR TED_*** properties for more information.

Alias Executables

add_executable(<name> ALIAS <target>)

Creates an *Alias Target*, such that **<name>** can be used to refer to **<target>** in subsequent commands. The**<name>** does not appear in the generated b uildsystem as a make target. The**<tar - get>** may not be an **ALIAS**.

Added in version 3.11: An **ALIAS** can target a **GLOBAL** *Imported Target*

Added in version 3.18: An **ALIAS** can target a non−**GLOBAL** Imported Target. Such alias is scoped to the directory in which it is created and subdirectories. The *ALIAS_GLOBAL* target property can be used to check if the alias is global or not.

ALIAS targets can be used as targets to read properties from, executables for custom commands and custom targets. They can also be tested for existence with the regular *if(TARGET)* subcommand. The **<name>** may not be used to modify properties of **<target>**, that is, it may not be used as the operand of *set_property()*, *set_target_properties()*, *target_link_libraries()* etc. An **ALIAS** target may not be installed or exported.

See Also

- *add_library()*

add_library

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

Normal Libraries

add_library(<name> [<type>] [EXCLUDE_FROM_ALL] <sources>...)

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

The optional **<type>** specifies the type of library to be created:

STATIC

A *Static Library*: an archive of object files for use when linking other targets.

SHARED

A *Shared Library*: a dynamic library that may be linked by other targets and loaded at runtime.

MODULE
A *Module Library*: a plugin that may not be linked by other targets, but may be dynamically loaded at runtime using dlopen−like functionality.

If no **<type>** is given the default is **STATIC** or **SHARED** based on the value of the *BUILD_SHARED_LIBS* variable.

The options are:

EXCLUDE_FROM_ALL
Set the *EXCLUDE_FROM_ALL* target property automatically. See documentation of that target property for details.

The **<name>** corresponds to the logical target name and must be globally unique within a project. The actual file name of the library built is constructed based on conventions of the native platform (such as **lib<name>.a** or **<name>.lib**).

Added in version 3.1: Source arguments to **add_library** may use "generator expressions" with the syntax **$<...>**. See the*cmak e−generator−expressions(7)* manual for available expressions.

Added in version 3.11: The source files can be omitted if they are added later using *target_sources()*.

For **SHARED** and **MODULE** libraries the *POSITION_INDEPENDENT_CODE* target property is set to **ON** automatically. A**SHARED** library may be mark ed with the *FRAMEWORK* target property to create an macOS Framework.

Added in version 3.8: A **STATIC** library may be marked with the *FRAMEWORK* target property to create a static Framework.

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

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

See the *cmake−buildsystem(7)* manual for more on defining buildsystem properties.

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

Changed in version 3.30: On platforms that do not support shared libraries, **add_library** now fails on calls creating **SHARED** libraries instead of automatically converting them to **STATIC** libraries as before. See policy *CMP0164*.

Object Libraries

add_library(<name> OBJECT <sources>...)
Add an *Object Library* to compile source files without archiving or linking their object files into a library.

Other targets created by **add_library** or *add_executable()* may reference the objects using an expression of the form *$<TARGET_OBJECTS:objlib>* as a source, where **objlib** is the object library name. For example:

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

will include objlib's object files in a library and an executable along with those compiled from their own sources. Object libraries may contain only sources that compile, header files, and other files that would not affect linking of a normal library (e.g. **.txt**). They may contain custom commands generating such sources, but not **PRE_BUILD**, **PRE_LINK**, or **POST_BUILD** commands. Some native build systems (such as Xcode) may not like targets that have only object files, so consider adding at least one real source file to any target that references *$<TARGET_OBJECTS:objlib>*.

Added in version 3.12: Object libraries can be linked to with *target_link_libraries()*.

Interface Libraries

add_library(<name> INTERFACE)
Add an *Interface Library* target that may specify usage requirements for dependents but does not compile sources and does not produce a library artifact on disk.

An interface library with no source files is not included as a target in the generated buildsystem. However, it may have properties set on it and it may be installed and exported. Typically, **INTERFACE_*** properties are populated on an interface target using the commands:

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

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

Added in version 3.15: An interface library can have *PUBLIC_HEADER* and *PRIVATE_HEADER* properties. The headers specified by those properties can be installed using the*install(TARGETS)* command.

add_library(<name> INTERFACE [EXCLUDE_FROM_ALL] <sources>...)
Added in version 3.19.

Add an *Interface Library* target with source files (in addition to usage requirements and properties as documented by the *above signature*). Source files may be listed directly in the**add_library** call or added later by calls to *target_sources()* with the **PRIVATE** or **PUBLIC** keywords.

If an interface library has source files (i.e. the *SOURCES* target property is set), or header sets (i.e. the *HEADER_SETS* target property is set), it will appear in the generated buildsystem as a build

target much like a target defined by the *add_custom_target()* command. It does not compile any sources, but does contain build rules for custom commands created by the *add_custom_command()* command.

The options are:

EXCLUDE_FROM_ALL
Set the *EXCLUDE_FROM_ALL* target property automatically. See documentation of that target property for details.

NOTE:
In most command signatures where the **INTERFACE** keyword appears, the items listed after it only become part of that target's usage requirements and are not part of the target's own settings. However, in this signature of **add_library**, the **INTERFACE** keyword refers to the library type only. Sources listed after it in the**add_library** call are **PRIV ATE** to the interface library and do not appear in its *INTERFACE_SOURCES* target property.

Imported Libraries

add_library(<name> <type> IMPORTED [GLOBAL])
Add an *IMPORTED library target* called **<name>**. The target name may be referenced like any target built within the project, except that by default it is visible only in the directory in which it is created, and below.

The **<type>** must be one of:

STATIC, SHARED, MODULE, UNKNOWN
References a library file located outside the project. The *IMPORTED_LOCATION* target property (or its per−configuration variant *IMPORTED_LOCATION_<CONFIG>*) specifies the location of the main library file on disk:

- For a **SHARED** library on most non−Windows platforms, the main library file is the **.so** or **.dylib** file used by both linkers and dynamic loaders. If the referenced library file has a **SONAME** (or on macOS, has a **LC_ID_DYLIB** starting in **@rpath/**), the value of that field should be set in the *IMPORTED_SONAME* target property. If the referenced library file does not have a **SONAME**, but the platform supports it, then the *IMPORTED_NO_SONAME* target property should be set.
- For a **SHARED** library on Windows, the *IMPORTED_IMPLIB* target property (or its per−configuration variant *IMPORTED_IMPLIB_<CONFIG>*) specifies the location of the DLL import library file (**.lib** or **.dll.a**) on disk, and the **IMPORTED_LOCATION** is the location of the **.dll** runtime library (and is optional, but needed by the *TARGET_RUNTIME_DLLS* generator expression).

Additional usage requirements may be specified in **INTERFACE_*** properties.

An **UNKNOWN** library type is typically only used in the implementation of *Find Modules*. It allows the path to an imported library (often found using the *find_library()* command) to be used without having to know what type of library it is. This is especially useful on Windows where a static library and a DLL's import library both have the same file extension.

OBJECT
References a set of object files located outside the project. The *IMPORTED_OBJECTS* target property (or its per−configuration variant *IMPORTED_OBJECTS_<CONFIG>*) specifies the locations of object files on disk. Additional usage requirements may be specified in **INTERFACE_*** properties.

INTERFACE
Does not reference any library or object files on disk, but may specify usage requirements in **INTERFACE_*** properties.

The options are:

GLOBAL
Make the target name globally visible.

No rules are generated to build imported targets, and the *IMPORTED* target property is **True**. Imported libraries are useful for convenient reference from commands like *target_link_libraries()*.

Details about the imported library are specified by setting properties whose names begin in **IMPORTED_** and **INTERFACE_**. See documentation of such properties for more information.

Alias Libraries

add_library(<name> ALIAS <target>)
Creates an *Alias Target*, such that **<name>** can be used to refer to **<target>** in subsequent commands. The**<name>** does not appear in the generated b uildsystem as a make target. The**<target>** may not be an **ALIAS**.

Added in version 3.11: An **ALIAS** can target a **GLOBAL** *Imported Target*

Added in version 3.18: An **ALIAS** can target a non−**GLOBAL** Imported Target. Such alias is scoped to the directory in which it is created and below. The*ALIAS_GLOB AL* target property can be used to check if the alias is global or not.

ALIAS targets can be used as linkable targets and as targets to read properties from. They can also be tested for existence with the regular *if(TARGET)* subcommand. The **<name>** may not be used to modify properties of **<target>**, that is, it may not be used as the operand of *set_property()*, *set_target_properties()*, *target_link_libraries()* etc. An **ALIAS** target may not be installed or exported.

See Also

- *add_executable()*

add_link_options

Added in version 3.13.

Add options to the link step for executable, shared library or module library targets in the current directory and below that are added after this command is invoked.

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

This command can be used to add any link options, but alternative commands exist to add libraries (*target_link_libraries()* or *link_libraries()*). See documentation of the*dir ectory* and *target* **LINK_OPTIONS** properties.

NOTE:
This command cannot be used to add options for static library targets, since they do not use a linker. To add archiver or MSVC librarian flags, see the *STATIC_LIBRARY_OPTIONS* target property.

Arguments to **add_link_options** may use generator expressions with the syntax **$<...>**. See the *cmake−generator−expressions(7)* manual for available expressions. See the*cmak e−buildsystem(7)* manual for more on defining buildsystem properties.

Host And Device Specific Link Options

Added in version 3.18: When a device link step is involved, which is controlled by *CUDA_SEPARABLE_COMPILATION* and *CUDA_RESOLVE_DEVICE_SYMBOLS* properties and policy *CMP0105*, the raw options will be delivered to the host and device link steps (wrapped in **−Xcompiler** or equivalent for device link). Options wrapped with *$<DEVICE_LINK:...>* generator expression will be used only for the device link step. Options wrapped with *$<HOST_LINK:...>* generator expression will be used only for the host link step.

Option De−duplication

The final set of options used for a target is constructed by accumulating options from the current target and the usage requirements of its dependencies. The set of options is de−duplicated to avoid repetition.

Added in version 3.12: While beneficial for individual options, the de−duplication step can break up option groups. For example, **−option A −option B** becomes **−option A B**. One may specify a group of options using shell−like quoting along with a **SHELL:** prefix. The **SHELL:** prefix is dropped, and the rest of the option string is parsed using the *separate_arguments()* **UNIX_COMMAND** mode. For example, **"SHELL:−option A" "SHELL:−option B"** becomes **−option A −option B**.

Handling Compiler Driver Differences

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.

See Also

- *link_libraries()*
- *target_link_libraries()*
- *target_link_options()*
- *CMAKE_<LANG>_FLAGS* and *CMAKE_<LANG>_FLAGS_<CONFIG>* add language−wide flags passed to all invocations of the compiler. This includes invocations that drive compiling and those that drive linking.

add_subdirectory

Add a subdirectory to the build.

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

Adds a subdirectory to the build. The**sour ce_dir** specifies the directory in which the source **CMakeLists.txt** and code files are located. If it is a relative path, it will be evaluated with respect to the current directory (the typical usage), but it may also be an absolute path. The **binary_dir** specifies the directory in which to place the output files. If it is a relative path, it will be evaluated with respect to the current output

directory, but it may also be an absolute path. If **binary_dir** is not specified, the value of **source_dir**, before expanding any relative path, will be used (the typical usage). The **CMakeLists.txt** file in the specified source directory will be processed immediately by CMake before processing in the current input file continues beyond this command.

If the **EXCLUDE_FROM_ALL** argument is provided then the *EXCLUDE_FROM_ALL* property will be set on the added directory. This will exclude the directory from a default build. See the directory property *EXCLUDE_FROM_ALL* for full details.

Added in version 3.25: If the **SYSTEM** argument is provided, the *SYSTEM* directory property of the subdirectory will be set to true. This property is used to initialize the *SYSTEM* property of each non−imported target created in that subdirectory.

add_test

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

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

Adds a test called **<name>**. The test name may contain arbitrary characters, expressed as a *Quoted Argument* or *Bracket Argument* if necessary. See policy *CMP0110*.

CMake only generates tests if the *enable_testing()* command has been invoked. The*CTest* module invokes **enable_testing** automatically unless *BUILD_TESTING* is set to **OFF**.

Tests added with the **add_test(NAME)** signature support using *generator expressions* in test properties set by *set_property(TEST)* or *set_tests_properties()*. Test properties may only be set in the directory the test is created in.

add_test options are:

COMMAND

Specify the test command−line.

If **<command>** specifies an executable target created by *add_executable()*:

- It will automatically be replaced by the location of the executable created at build time.
- Added in version 3.3: The target's *CROSSCOMPILING_EMULATOR*, if set, will be used to run the command on the host:

  ```
  <emulator> <command>
  ```

  Changed in version 3.29: The emulator is used only when *cross−compiling*. See policy *CMP0158*.

- Added in version 3.29: The target's *TEST_LAUNCHER*, if set, will be used to launch the command:

  ```
  <launcher> <command>
  ```

  If the *CROSSCOMPILING_EMULATOR* is also set, both are used:

```
<launcher> <emulator> <command>
```

The command may be specified using *generator expressions*.

CONFIGURATIONS
Restrict execution of the test only to the named configurations.

WORKING_DIRECTORY
Set the test property *WORKING_DIRECTORY* in which to execute the test. If not specified, the test will be run in *CMAKE_CURRENT_BINARY_DIR*. The working directory may be specified using *generator expressions*.

COMMAND_EXPAND_LISTS
Added in version 3.16.

Lists in **COMMAND** arguments will be expanded, including those created with *generator expressions*.

If the test command exits with code **0** the test passes. Non−zero exit code is a "failed" test. The test property *WILL_FAIL* inverts this logic. Note that system−level test failures such as segmentation faults or heap errors will still fail the test even if **WILL_FAIL** is true. Output written to stdout or stderr is captured by *ctest(1)* and only affects the pass/fail status via the *PASS_REGULAR_EXPRESSION*, *FAIL_REGULAR_EXPRESSION*, or *SKIP_REGULAR_EXPRESSION* test properties.

Added in version 3.16: Added *SKIP_REGULAR_EXPRESSION* property.

Example usage:

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

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

The command syntax above is recommended over the older, less flexible form:

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

Add a test called **<name>** with the given command−line.

Unlike the above **NAME** signature, target names are not supported in the command−line. Furthermore, tests added with this signature do not support *generator expressions* in the command−line or test properties.

aux_source_directory
Find all source files in a directory.

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

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

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

build_command

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

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

Sets the given **<variable>** to a command−line string of the form:

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

where **<cmake>** is the location of the *cmake(1)* command−line tool, and **<config>**, **<parallel>** and **<target>** are the values provided to the **CONFIGURATION**, **PARALLEL_LEVEL** and **TARGET** options, if any. In CMake versions prior to 4.0, a trailing **−− −i** option was added for *Makefile Generators* if policy *CMP0061* was not set to **NEW**.

When invoked, this *cmake −−build* command line will launch the underlying build system tool.

Added in version 3.21: The **PARALLEL_LEVEL** argument can be used to set the *−−parallel* flag.

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

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

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

NOTE:

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

cmake_file_api

Added in version 3.27.

Enables interacting with the *CMake file API*.

cmake_file_api(QUERY ...)

The **QUERY** subcommand adds a file API query for the current CMake invocation.

```
cmake_file_api(
  QUERY
  API_VERSION <version>
  [CODEMODEL <versions>...]
  [CACHE <versions>...]
  [CMAKEFILES <versions>...]
  [TOOLCHAINS <versions>...]
)
```

The **API_VERSION** must always be given. Currently, the only supported value for <**version**> is 1. See*API v1* for details of the reply content and location.

Each of the optional keywords **CODEMODEL**, **CACHE**, **CMAKEFILES** and **TOOLCHAINS** correspond to one of the object kinds that can be requested by the project. The **configureLog** object kind cannot be set with this command, since it must be set before CMake starts reading the top level **CMakeLists.txt** file.

For each of the optional keywords, the <**versions**> list must contain one or more version values of the form **major** or **major.minor**, where **major** and **minor** are integers. Projects should list the versions they accept in their preferred order, as only the first supported value from the list will be selected. The command will ignore versions with a **major** version higher than any major version it supports for that object kind. It will raise an error if it encounters an invalid version number, or if none of the requested versions is supported.

For each type of object kind requested, a query equivalent to a shared, stateless query will be added internally. No query file will be created in the file system. The reply*will* be written to the file system at generation time.

It is not an error to add a query for the same thing more than once, whether from query files or from multiple calls to **cmake_file_api(QUERY)**. The final set of queries will be a merged combination of all queries specified on disk and queries submitted by the project.

Example

A project may want to use replies from the file API at build time to implement some form of verification task. Instead of relying on something outside of CMake to create a query file, the project can use **cmake_file_api(QUERY)** to request the required information for the current run. It can then create a custom command to run at build time, knowing that the requested information should always be available.

```
cmake_file_api(
  QUERY
  API_VERSION 1
  CODEMODEL 2.3
  TOOLCHAINS 1
)

add_custom_target(verify_project
  COMMAND ${CMAKE_COMMAND}
    -D BUILD_DIR=${CMAKE_BINARY_DIR}
    -D CONFIG=$<CONFIG>
    -P ${CMAKE_CURRENT_SOURCE_DIR}/verify_project.cmake
)
```

cmake_instrumentation

Added in version 4.0.

NOTE:

This command is only available when experimental support for instrumentation has been enabled by the **CMAKE_EXPERIMENTAL_INSTRUMENTATION** gate.

Enables interacting with the *CMake Instrumentation API*.

This allows for configuring instrumentation at the project−level.

```
cmake_instrumentation(
  API_VERSION <version>
  DATA_VERSION <version>
  [HOOKS <hooks>...]
  [QUERIES <queries>...]
  [CALLBACK <callback>]
)
```

The **API_VERSION** and **DATA_VERSION** must always be given. Currently, the only supported value for both fields is 1. See *API v1* for details of the **API_VERSION** and *Data v1* for details of the **DATA_VERSION**.

Each of the optional keywords **HOOKS**, **QUERIES**, and **CALLBACK** correspond to one of the parameters to the *v1 Query Files*. The**CALLB ACK** keyword can be provided multiple times to create multiple callbacks.

Whenever **cmake_instrumentation** is invoked, a query file is generated in **<build>/.cmake/instrumentation/v1/query/generated** to enable instrumentation with the provided arguments.

Example

The following example shows an invocation of the command and its equivalent JSON query file.

```
cmake_instrumentation(
  API_VERSION 1
  DATA_VERSION 1
  HOOKS postGenerate preCMakeBuild postCMakeBuild
  QUERIES staticSystemInformation dynamicSystemInformation
  CALLBACK ${CMAKE_COMMAND} −P /path/to/handle_data.cmake
  CALLBACK ${CMAKE_COMMAND} −P /path/to/handle_data_2.cmake
)

{
  "version": 1,
  "hooks": [
    "postGenerate", "preCMakeBuild", "postCMakeBuild"
  ],
  "queries": [
    "staticSystemInformation", "dynamicSystemInformation"
  ],
  "callbacks": [
    "/path/to/cmake −P /path/to/handle_data.cmake"
    "/path/to/cmake −P /path/to/handle_data_2.cmake"
  ]
```

```
}
```

create_test_sourcelist

Create a test driver program that links together many small tests into a single executable. This is useful when building static executables with large libraries to shrink the total required size.

create_test_sourcelist(<sourceListName> <driverName> <test>... <options>...)

Generate a test driver source file from a list of individual test sources and provide a combined list of sources that can be built as an executable.

The options are:

<sourceListName>

The name of a variable in which to store the list of source files needed to build the test driver. The list will contain the**<test>...** sources and the generated **<dri verName>** source.

Changed in version 3.29: The test driver source is listed by absolute path in the build tree. Previously it was listed only as **<driverName>**.

<driverName>

Name of the test driver source file to be generated into the build tree. The source file will contain a **main()** program entry point that dispatches to whatever test is named on the command line.

<test>...

Test source files to be added to the driver binary. Each test source file must have a function in it that is the same name as the file with the extension removed. For example, a **foo.cxx** test source might contain:

```
int foo(int argc, char** argv)
```

EXTRA_INCLUDE <header>

Specify a header file to **#include** in the generated test driver source.

FUNCTION <function>

Specify a function to be called with pointers to **argc** and **argv**. The function may be provided in the **EXTRA_INCLUDE** header:

```
void function(int* pargc, char*** pargv)
```

This can be used to add extra command line processing to each test.

Additionally, some CMake variables affect test driver generation:

CMAKE_TESTDRIVER_BEFORE_TESTMAIN

Code to be placed directly before calling each test's function.

CMAKE_TESTDRIVER_AFTER_TESTMAIN

Code to be placed directly after the call to each test's function.

define_property

Define and document custom properties.

```
define_property(<GLOBAL | DIRECTORY | TARGET | SOURCE |
                 TEST | VARIABLE | CACHED_VARIABLE>
                 PROPERTY <name> [INHERITED]
                 [BRIEF_DOCS <brief-doc> [docs...]]
                 [FULL_DOCS <full-doc> [docs...]]
```

```
[INITIALIZE_FROM_VARIABLE <variable>])
```

Defines one property in a scope for use with the *set_property()* and *get_property()* commands. It is mainly useful for defining the way a property is initialized or inherited. Historically, the command also associated documentation with a property, but that is no longer considered a primary use case.

The first argument determines the kind of scope in which the property should be used. It must be one of the following:

- **GLOBAL** − associated with the global namespace.
- **DIRECTORY** − associated with one directory.
- **TARGET** − associated with one target.
- **SOURCE** − associated with one source file.
- **TEST** − associated with a test named with *add_test()*.
- **VARIABLE** − documents a CMake language variable.
- **CACHED_VARIABLE** − documents a CMake cache variable.

Note that unlike *set_property()* and *get_property()* no actual scope needs to be given; only the kind of scope is important.

The required **PROPERTY** option is immediately followed by the name of the property being defined.

If the **INHERITED** option is given, then the *get_property()* command will chain up to the next higher scope when the requested property is not set in the scope given to the command.

- **DIRECTORY** scope chains to its parent directory's scope, continuing the walk up parent directories until a directory has the property set or there are no more parents. If still not found at the top level directory, it chains to the **GLOBAL** scope.
- **TARGET**, **SOURCE** and **TEST** properties chain to **DIRECTORY** scope, including further chaining up the directories, etc. as needed.

Note that this scope chaining behavior only applies to calls to *get_property()*, *get_directory_property()*, *get_target_property()*, *get_source_file_property()* and *get_test_property()*. There is no inheriting behavior when *setting* properties, so using **APPEND** or **APPEND_STRING** with the *set_property()* command will not consider inherited values when working out the contents to append to.

The **BRIEF_DOCS** and **FULL_DOCS** options are followed by strings to be associated with the property as its brief and full documentation. CMake does not use this documentation other than making it available to the project via corresponding options to the *get_property()* command.

Changed in version 3.23: The **BRIEF_DOCS** and **FULL_DOCS** options are optional.

Added in version 3.23: The **INITIALIZE_FROM_VARIABLE** option specifies a variable from which the property should be initialized. It can only be used with target properties. The **<variable>** name must end with the property name and must not begin with **CMAKE_** or **_CMAKE_**. The property name must contain at least one underscore. It is recommended that the property name have a prefix specific to the project.

Property Redefinition

Once a property is defined for a particular type of scope, it cannot be redefined. Attempts to redefine an existing property by calling *define_property()* with the same scope type and property name will be silently ignored. Defining the same property name for two different kinds of scope is valid.

get_property() can be used to determine whether a property is already defined for a particular kind of scope, and if so, to examine its definition. For example:

```
# Initial definition
define_property(TARGET PROPERTY MY_NEW_PROP
  BRIEF_DOCS "My new custom property"
)

# Later examination
get_property(my_new_prop_exists
  TARGET NONE
  PROPERTY MY_NEW_PROP
  DEFINED
)

if(my_new_prop_exists)
  get_property(my_new_prop_docs
    TARGET NONE
    PROPERTY MY_NEW_PROP
    BRIEF_DOCS
  )
  # ${my_new_prop_docs} is now set to "My new custom property"
endif()
```

See Also

- *get_property()*
- *set_property()*

enable_language

Enable languages (CXX/C/OBJC/OBJCXX/Fortran/etc)

```
enable_language(<lang>... [OPTIONAL])
```

Enables support for the named languages in CMake. This is the same as the*pr oject()* command but does not create any of the extra variables that are created by the *project()* command.

Supported languages are **C**, **CXX** (i.e. C++), **CSharp** (i.e. C#), **CUDA**, **OBJC** (i.e. Objective−C), **OBJCXX** (i.e. Objective−C++), **Fortran**, **HIP**, **ISPC**, **Swift**, **ASM**, **ASM_NASM**, **ASM_MARMASM**, **ASM_MASM**, and **ASM−ATT**.

Added in version 3.8: Added **CSharp** and **CUDA** support.

Added in version 3.15: Added **Swift** support.

Added in version 3.16: Added **OBJC** and **OBJCXX** support.

Added in version 3.18: Added **ISPC** support.

Added in version 3.21: Added **HIP** support.

Added in version 3.26: Added **ASM_MARMASM** support.

If enabling **ASM**, list it last so that CMake can check whether compilers for other languages like **C** work for assembly too.

The following restrictions apply to where **enable_language()** may be called:

- It must be called in file scope, not in a function call.
- It must not be called before the first call to *project()*. See policy *CMP0165*.
- It must be called in the highest directory common to all targets using the named language directly for compiling sources or indirectly through link dependencies. It is simplest to enable all needed languages in the top−level directory of a project.

The **OPTIONAL** keyword is a placeholder for future implementation and does not currently work. Instead you can use the *CheckLanguage* module to verify support before enabling.

enable_testing

Enable testing for current directory and below.

```
enable_testing()
```

Enables testing for this directory and below.

This command should be in the top−level source directory because *ctest(1)* expects to find a test file in the top−level build directory.

This command is automatically invoked when the *CTest* module is included, except if the *BUILD_TESTING* option is turned off.

See also the *add_test()* command.

export

Export targets or packages for outside projects to use them directly from the current project's build tree, without installation.

See the *install(EXPORT)* command to export targets from an install tree.

Synopsis

```
export(TARGETS <target>... [...])
export(EXPORT <export-name> [...])
export(PACKAGE <PackageName>)
export(SETUP <export-name> [...])
```

Exporting Targets

export(TARGETS <target>... [...])

```
export(TARGETS <target>... [NAMESPACE <namespace>]
       [APPEND] FILE <filename> [EXPORT_LINK_INTERFACE_LIBRARIES]
       [CXX_MODULES_DIRECTORY <directory>])
```

Creates a file **<filename>** that may be included by outside projects to import targets named by **<target>...** from the current project's build tree. This is useful during cross−compiling to build utility executables that can run on the host platform in one project and then import them into another project being compiled for the target platform.

The file created by this command is specific to the build tree and should never be installed. See the *install(EXPORT)* command to export targets from an install tree.

The options are:

NAMESPACE <namespace>
Prepend the **<namespace>** string to all target names written to the file.

APPEND
Append to the file instead of overwriting it. This can be used to incrementally export multiple targets to the same file.

EXPORT_LINK_INTERFACE_LIBRARIES
Include the contents of the properties named with the pattern **(IMPORTED_)?LINK_INTERFACE_LIBRARIES(_<CONFIG>)?** in the export, even when policy *CMP0022* is NEW. This is useful to support consumers using CMake versions older than 2.8.12.

CXX_MODULES_DIRECTORY <directory>
Added in version 3.28.

Export C++ module properties to files under the given directory. Each file will be named according to the target's export name (without any namespace). These files will automatically be included from the export file.

This signature requires all targets to be listed explicitly. If a library target is included in the export, but a target to which it links is not included, the behavior is unspecified. See the *export(EXPORT)* signature to automatically export the same targets from the build tree as *install(EXPORT)* would from an install tree.

NOTE:
Object Libraries under *Xcode* have special handling if multiple architectures are listed in *CMAKE_OSX_ARCHITECTURES*. In this case they will be exported as *Interface Libraries* with no object files available to clients. This is sufficient to satisfy transitive usage requirements of other targets that link to the object libraries in their implementation.

This command exports all *Build Configurations* from the build tree. See the *CMAKE_MAP_IMPORTED_CONFIG_<CONFIG>* variable to map configurations of dependent projects to the exported configurations.

Exporting Targets to Android.mk

```
export(TARGETS <target>... ANDROID_MK <filename>)
```

Added in version 3.7.

This signature exports cmake built targets to the android ndk build system by creating an **Android.mk** file that references the prebuilt targets. The Android NDK supports the use of prebuilt libraries, both static and shared. This allows cmake to build the libraries of a project and make them available to an ndk build system complete with transitive dependencies, include flags and defines required to use the libraries. The signature takes a list of targets and puts them in the **Android.mk** file specified by the **<filename>** given. This signature can only be used if policy *CMP0022* is NEW for all targets given. A error will be issued if that policy is set to OLD for one of the targets.

Exporting Targets matching install(EXPORT)

export(EXPORT <export−name> [...])

```
export(EXPORT <export−name> [NAMESPACE <namespace>] [FILE <filename>]
       [CXX_MODULES_DIRECTORY <directory>] [EXPORT_PACKAGE_DEPENDENCIES])
```

Creates a file **<filename>** that may be included by outside projects to import targets from the current project's build tree. This is the same as the *export(TARGETS)* signature, except that the targets are not explicitly listed. Instead, it exports the targets associated with the installation export **<export−name>**. Target

installations may be associated with the export **<export−name>** using the **EXPORT** option of the *install(TARGETS)* command.

EXPORT_PACKAGE_DEPENDENCIES

NOTE:
Experimental. Gated by **CMAKE_EXPERIMENTAL_EXPORT_PACKAGE_DEPENDENCIES**.

Specify that *find_dependency()* calls should be exported. See *install(EXPORT)* for details on how this works.

Exporting Packages

export(PACKAGE <PackageName>)

```
export(PACKAGE <PackageName>)
```

Store the current build directory in the CMake user package registry for package **<PackageName>**. The *find_package()* command may consider the directory while searching for package **<PackageName>**. This helps dependent projects find and use a package from the current project's build tree without help from the user. Note that the entry in the package registry that this command creates works only in conjunction with a package configuration file (**<PackageName>Config.cmake**) that works with the build tree. In some cases, for example for packaging and for system wide installations, it is not desirable to write the user package registry.

Changed in version 3.1: If the *CMAKE_EXPORT_NO_PACKAGE_REGISTRY* variable is enabled, the **export(PACKAGE)** command will do nothing.

Changed in version 3.15: By default the **export(PACKAGE)** command does nothing (see policy *CMP0090*) because populating the user package registry has effects outside the source and build trees. Set the *CMAKE_EXPORT_PACKAGE_REGISTRY* variable to add build directories to the CMake user package registry.

Configuring Exports

export(SETUP <export−name> [...])

```
export(SETUP <export−name>
       [PACKAGE_DEPENDENCY <dep>
        [ENABLED (<bool−true>|<bool−false>|AUTO)]
        [EXTRA_ARGS <args>...]
       ] [...]
       [TARGET <target>
        [XCFRAMEWORK_LOCATION <location>]
       ] [...]
       )
```

Added in version 3.29.

Configure the parameters of an export. The arguments are as follows:

PACKAGE_DEPENDENCY <dep>

NOTE:
Experimental. Gated by

CMAKE_EXPERIMENTAL_EXPORT_PACKAGE_DEPENDENCIES.

Specify a package dependency to configure. This changes how *find_dependency()* calls are written during *export(EXPORT)* and *install(EXPORT)*. **<dep>** is the name of a package to export. This argument accepts the following additional arguments:

ENABLED
Manually control whether or not the dependency is exported. This accepts the following values:

<bool−true>
Any value that CMake recognizes as "true". Always export the dependency, even if no exported targets depend on it. This can be used to manually add *find_dependency()* calls to the export.

<bool−false>
Any value that CMake recognizes as "false". Never export the dependency, even if an exported target depends on it.

AUTO Only export the dependency if an exported target depends on it.

EXTRA_ARGS <args>
Specify additional arguments to pass to *find_dependency()* after the **REQUIRED** argument.

TARGET <target>
Specify a target to configure in this export. This argument accepts the following additional arguments:

XCFRAMEWORK_LOCATION
Specify the location of an **.xcframework** which contains the library from this target. If specified, the generated code will check to see if the **.xcframework** exists, and if it does, it will use the **.xcframework** as its imported location instead of the installed library.

fltk_wrap_ui
Create FLTK user interfaces Wrappers.

```
fltk_wrap_ui(resultingLibraryName source1
             source2 ... sourceN)
```

Produce .h and .cxx files for all the .fl and .fld files listed. The resulting .h and .cxx files will be added to a variable named **resultingLibraryName_FLTK_UI_SRCS** which should be added to your library.

get_source_file_property
Get a property for a source file.

```
get_source_file_property(<variable> <file>
                         [DIRECTORY <dir> | TARGET_DIRECTORY <target>]
                         <property>)
```

Gets a property from a source file. The value of the property is stored in the specified **<variable>**. If the **<file>** is not a source file, or the source property is not found, **<variable>** will be set to **NOTFOUND**. If the source property was defined to be an **INHERITED** property (see *define_property()*), the search will include the relevant parent scopes, as described for the *define_property()* command.

By default, the source file's property will be read from the current source directory's scope.

Added in version 3.18: Directory scope can be overridden with one of the following sub−options:

DIRECTORY <dir>
The source file property will be read from the **<dir>** directory's scope. CMake must already know about that source directory, either by having added it through a call to *add_subdirectory()* or **<dir>** being the top level source directory. Relative paths are treated as relative to the current source directory.

TARGET_DIRECTORY <target>
The source file property will be read from the directory scope in which **<target>** was created (**<target>** must therefore already exist).

Use *set_source_files_properties()* to set property values. Source file properties usually control how the file is built. One property that is always there is *LOCATION*.

NOTE:
The *GENERATED* source file property may be globally visible. See its documentation for details.

See Also
- *define_property()*
- the more general *get_property()* command
- *set_source_files_properties()*

get_target_property
Get a property from a target.

```
get_target_property(<variable> <target> <property>)
```

Get a property from a target. The value of the property is stored in the specified **<variable>**. If the target property is not found, **<variable>** will be set to **<variable>−NOTFOUND**. If the target property was defined to be an **INHERITED** property (see *define_property()*), the search will include the relevant parent scopes, as described for the *define_property()* command.

Use *set_target_properties()* to set target property values. Properties are usually used to control how a target is built, but some query the target instead. This command can get properties for any target so far created. The targets do not need to be in the current **CMakeLists.txt** file.

See Also
- *define_property()*
- the more general *get_property()* command
- *set_target_properties()*
- *Properties on Targets* for the list of properties known to CMake

get_test_property
Get a property of the test.

```
get_test_property(<test> <property> [DIRECTORY <dir>] <variable>)
```

Get a property from the test. The value of the property is stored in the specified **<variable>**. If the**<test>** is not defined, or the test property is not found, **<variable>** will be set to **NOTFOUND**. If the test property was defined to be an **INHERITED** property (see *define_property()*), the search will include the relevant parent scopes, as described for the *define_property()* command.

For a list of standard properties you can type *cmake −−help−property−list*.

Added in version 3.28: Directory scope can be overridden with the following sub−option:

DIRECTORY <dir>

The test property will be read from the **<dir>** directory's scope. CMake must already know about that source directory, either by having added it through a call to *add_subdirectory()* or **<dir>** being the top level source directory. Relative paths are treated as relative to the current source directory. **<dir>** may reference a binary directory.

See Also

- *define_property()*
- the more general *get_property()* command

include_directories

Add include directories to the build.

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

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

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

By default the directories specified are appended onto the current list of directories. This default behavior can be changed by setting *CMAKE_INCLUDE_DIRECTORIES_BEFORE* to **ON**. By using**AFTER** or **BEFORE** explicitly, you can select between appending and prepending, independent of the default.

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

Arguments to **include_directories** may use generator expressions with the syntax **$<...>**. See the *cmake−generator−expressions(7)* manual for available expressions. See the*cmak e−buildsystem(7)* manual for more on defining buildsystem properties.

NOTE:

Prefer the *target_include_directories()* command to add include directories to individual targets and optionally propagate/export them to dependents.

See Also

- *target_include_directories()*

include_external_msproject

Include an external Microsoft project file in the solution file produced by *Visual Studio Generators*. Ignored on other generators.

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

Includes an external Microsoft project in the generated solution file. This will create a target named **[projectname]**. This can be used in the*add_dependencies()* command to mak e things depend on the external project.

TYPE, **GUID** and **PLATFORM** are optional parameters that allow one to specify the type of project, id (**GUID**) of the project and the name of the target platform. This is useful for projects requiring values other than the default (e.g. WIX projects).

Added in version 3.9: If the imported project has different configuration names than the current project, set the *MAP_IMPORTED_CONFIG_<CONFIG>* target property to specify the mapping.

include_regular_expression

Set the regular expression used for dependency checking.

```
include_regular_expression(regex_match [regex_complain])
```

Sets the regular expressions used in dependency checking. Only files matching **regex_match** will be traced as dependencies. Only files matching **regex_complain** will generate warnings if they cannot be found (standard header paths are not searched).

The default for **regex_match** is **"^.*$"** (match everything). The default for **regex_complain** is **"^$"** (match empty string only).

install

Specify rules to run at install time.

Synopsis

```
install(TARGETS <target>... [...])
install(IMPORTED_RUNTIME_ARTIFACTS <target>... [...])
install({FILES | PROGRAMS} <file>... [...])
install(DIRECTORY <dir>... [...])
install(SCRIPT <file> [...])
install(CODE <code> [...])
install(EXPORT <export-name> [...])
install(PACKAGE_INFO <package-name> [...])
install(RUNTIME_DEPENDENCY_SET <set-name> [...])
```

Introduction

This command generates installation rules for a project. Install rules specified by calls to the **install()** command within a source directory are executed in order during installation.

Changed in version 3.14: Install rules in subdirectories added by calls to the *add_subdirectory()* command are interleaved with those in the parent directory to run in the order declared (see policy *CMP0082*).

Changed in version 3.22: The environment variable *CMAKE_INSTALL_MODE* can override the default copying behavior of **install()**.

Changed in version 3.31: Projects can enable *INSTALL_PARALLEL* to enable a parallel installation. When using the parallel install, subdirectories added by calls to the *add_subdirectory()* command are installed independently and the order that install rules added in different subdirectories will run is not guaranteed.

There are multiple signatures for this command. Some of them define installation options for files and targets. Options common to multiple signatures are covered here but they are valid only for signatures that specify them. The common options are:

DESTINATION <dir>
Specify the directory on disk to which a file will be installed. **<dir>** should be a relati ve path. An absolute path is allowed, but not recommended.

When a relative path is given, it is interpreted relative to the value of the *CMAKE_INSTALL_PREFIX* variable. The prefix can be relocated at install time using the**DEST - DIR** mechanism explained in the *CMAKE_INSTALL_PREFIX* variable documentation.

As absolute paths do not work with the **cmake −−install** command's *−−prefix* option, or with the *cpack* installer generators, it is strongly recommended to use relative paths throughout for best support by package maintainers. In particular, there is no need to make paths absolute by prepending *CMAKE_INSTALL_PREFIX*; this prefix is used by default if the DESTINATION is a relative path.

If an absolute path (with a leading slash or drive letter) is given it is used verbatim.

Changed in version 3.31: **<dir>** will be normalized according to the same *normalization rules* as the *cmake_path()* command.

PERMISSIONS <permission>...
Specify permissions for installed files. Valid permissions are **OWNER_READ**, **OWNER_WRITE**, **OWNER_EXECUTE**, **GROUP_READ**, **GROUP_WRITE**, **GROUP_EXECUTE**, **WORLD_READ**, **WORLD_WRITE**, **WORLD_EXECUTE**, **SETUID**, and **SETGID**. Permissions that do not make sense on certain platforms are ignored on those platforms.

If this option is used multiple times in a single call, its list of permissions accumulates. If an *install(TARGETS)* call uses *<artifact−kind>* arguments, a separate list of permissions is accumulated for each kind of artifact.

CONFIGURATIONS <config>...
Specify a list of build configurations for which the install rule applies (Debug, Release, etc.).

If this option is used multiple times in a single call, its list of configurations accumulates. If an *install(TARGETS)* call uses *<artifact−kind>* arguments, a separate list of configurations is accumulated for each kind of artifact.

COMPONENT <component>
Specify an installation component name with which the install rule is associated, such as **Runtime** or **Development**. During component−specific installation only install rules associated with the given component name will be executed. During a full installation all components are installed unless marked with **EXCLUDE_FROM_ALL**. If**COMPONENT** is not pro vided a default component "Unspecified" is created. The default component name may be controlled with the *CMAKE_INSTALL_DEFAULT_COMPONENT_NAME* variable.

Installation components can be then used by the **cmake −−install** command's *−−component* option and the *CPackComponent* module. Global target **list_install_components** lists all available components.

EXCLUDE_FROM_ALL
Added in version 3.6.

Specify that the file is excluded from a full installation and only installed as part of a component−specific installation

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

Added in version 3.1: Command signatures that install files may print messages during installation. Use the *CMAKE_INSTALL_MESSAGE* variable to control which messages are printed.

Added in version 3.11: Many of the **install()** variants implicitly create the directories containing the installed files. If *CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS* is set, these directories will be created with the permissions specified. Otherwise, they will be created according to the uname rules on Unix−like platforms. Windows platforms are unaffected.

Signatures

install(TARGETS <target>... [...])

Install target *Output Artifacts* and associated files:

```
install(TARGETS <target>... [EXPORT <export-name>]
        [RUNTIME_DEPENDENCIES <arg>...|RUNTIME_DEPENDENCY_SET <set-name>]
        [<artifact-option>...]
        [<artifact-kind> <artifact-option>...]...
        [INCLUDES DESTINATION [<dir> ...]]
        )
```

where **<artifact−option>...** group may contain:

```
[DESTINATION <dir>]
[PERMISSIONS <permission>...]
[CONFIGURATIONS <config>...]
[COMPONENT <component>]
[NAMELINK_COMPONENT <component>]
[OPTIONAL] [EXCLUDE_FROM_ALL]
[NAMELINK_ONLY|NAMELINK_SKIP]
```

The first **<artifact−option>...** group applies to target *Output Artifacts* that do not have a dedicated group specified later in the same call.

Each **<artifact−kind>** **<artifact−option>...** group applies to *Output Artifacts* of the specified artifact kind:

ARCHIVE
Target artifacts of this kind include:

- *Static libraries* (except on macOS when marked as **FRAMEWORK**, see below);
- *DLL import libraries* (on all Windows−based systems including Cygwin; they have extension **.lib**, in contrast to the **.dll** libraries that go to **RUNTIME**);
- On AIX, the *linker import file* created for executables with *ENABLE_EXPORTS* enabled.
- On macOS, the *linker import file* created for shared libraries with *ENABLE_EXPORTS* enabled (except when marked as **FRAMEWORK**, see below).

LIBRARY
Target artifacts of this kind include:

- *Shared libraries*, except
  - DLLs (these go to **RUNTIME**, see below),
  - on macOS when marked as **FRAMEWORK** (see below).

RUNTIME
Target artifacts of this kind include:

- *Executables* (except on macOS when marked as **MACOSX_BUNDLE**, see **BUNDLE** below);
- DLLs (on all Windows−based systems including Cygwin; note that the accompanying import libraries are of kind **ARCHIVE**).

OBJECTS
Added in version 3.9.

Object files associated with *object libraries*.

FRAMEWORK
Both static and shared libraries marked with the **FRAMEWORK** property are treated as **FRAMEWORK** targets on macOS.

BUNDLE
Executables marked with the *MACOSX_BUNDLE* property are treated as **BUNDLE** targets on macOS.

PUBLIC_HEADER
Any *PUBLIC_HEADER* files associated with a library are installed in the destination specified by the **PUBLIC_HEADER** argument on non−Apple platforms. Rules defined by this argument are ignored for *FRAMEWORK* libraries on Apple platforms because the associated files are installed into the appropriate locations inside the framework folder. See *PUBLIC_HEADER* for details.

PRIVATE_HEADER
Similar to **PUBLIC_HEADER**, but for **PRIVATE_HEADER** files. See *PRIVATE_HEADER* for details.

RESOURCE
Similar to **PUBLIC_HEADER** and **PRIVATE_HEADER**, but for **RESOURCE** files. See *RESOURCE* for details.

FILE_SET <set−name>
Added in version 3.23.

File sets are defined by the *target_sources(FILE_SET)* command. If the file set **<set−name>** exists and is **PUBLIC** or **INTERFACE**, any files in the set are installed under the destination (see below). The directory structure relative to the file set's base directories is preserved. For example, a file added to the file set as **/blah/include/myproj/here.h** with a base directory **/blah/include** would be installed to **myproj/here.h** below the destination.

CXX_MODULES_BMI
Added in version 3.28.

Any module files from C++ modules from **PUBLIC** sources in a file set of type **CXX_MODULES** will be installed to the given **DESTINATION**. All modules are placed directly in the destination as no directory structure is derived from the names of

the modules. An empty **DESTINATION** may be used to suppress installing these files (for use in generic code).

For regular executables, static libraries and shared libraries, the **DESTINATION** argument is not required. For these target types, when **DESTINATION** is omitted, a default destination will be taken from the appropriate variable from *GNUInstallDirs*, or set to a built−in default value if that variable is not defined. The same is true for file sets, and the public and private headers associated with the installed targets through the *PUBLIC_HEADER* and *PRIVATE_HEADER* target properties. A destination must always be provided for module libraries, Apple bundles and frameworks. A destination can be omitted for interface and object libraries, but they are handled differently (see the discussion of this topic toward the end of this section).

For shared libraries on DLL platforms, if neither **RUNTIME** nor **ARCHIVE** destinations are specified, both the **RUNTIME** and **ARCHIVE** components are installed to their default destinations. If either a **RUNTIME** or **ARCHIVE** destination is specified, the component is installed to that destination, and the other component is not installed. If both **RUNTIME** and **ARCHIVE** destinations are specified, then both components are installed to their respective destinations.

The following table shows the target types with their associated variables and built−in defaults that apply when no destination is given:

Target Type	GNUInstallDirs Variable	Built−In Default
RUNTIME	**${CMAKE_INSTALL_BINDIR}**	**bin**
LIBRARY	**${CMAKE_INSTALL_LIBDIR}**	**lib**
ARCHIVE	**${CMAKE_INSTALL_LIBDIR}**	**lib**
PRIVATE_HEADER	**${CMAKE_INSTALL_INCLUDEDIR}**	**include**
PUBLIC_HEADER	**${CMAKE_INSTALL_INCLUDEDIR}**	**include**
FILE_SET (type **HEADERS**)	**${CMAKE_INSTALL_INCLUDEDIR}**	**include**

Projects wishing to follow the common practice of installing headers into a project−specific subdirectory may prefer using file sets with appropriate paths and base directories. Otherwise, they must provide a **DESTINATION** instead of being able to rely on the above (see next example below).

To make packages compliant with distribution filesystem layout policies, if projects must specify a **DESTINATION**, it is strongly recommended that they use a path that begins with the appropriate relative *GNUInstallDirs* variable. This allows package maintainers to control the install destination by setting the appropriate cache variables. The following example shows a static library being installed to the default destination provided by *GNUInstallDirs*, but with its headers installed to a project−specific subdirectory without using file sets:

```
add_library(mylib STATIC ...)
set_target_properties(mylib PROPERTIES PUBLIC_HEADER mylib.h)
include(GNUInstallDirs)
install(TARGETS mylib
        PUBLIC_HEADER
          DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}/myproj
)
```

In addition to the *common options* listed above, each target can accept the following additional arguments:

NAMELINK_COMPONENT
Added in version 3.12.

On some platforms a versioned shared library has a symbolic link such as:

```
lib<name>.so -> lib<name>.so.1
```

where **lib<name>.so.1** is the soname of the library and **lib<name>.so** is a "namelink" allowing linkers to find the library when given **−l<name>**. The **NAMELINK_COMPONENT** option is similar to the **COMPONENT** option, but it changes the installation component of a shared library namelink if one is generated. If not specified, this defaults to the value of **COMPONENT**. It is an error to use this parameter outside of a **LIBRARY** block.

Changed in version 3.27: This parameter is also usable for an **ARCHIVE** block to manage the linker import file created, on macOS, for shared libraries with *ENABLE_EXPORTS* enabled.

See the *Example: Install Targets with Per−Artifact Components* for an example using **NAMELINK_COMPONENT**.

This option is typically used for package managers that have separate runtime and development packages. For example, on Debian systems, the library is expected to be in the runtime package, and the headers and namelink are expected to be in the development package.

See the *VERSION* and *SOVERSION* target properties for details on creating versioned shared libraries.

NAMELINK_ONLY
This option causes the installation of only the namelink when a library target is installed. On platforms where versioned shared libraries do not have namelinks or when a library is not versioned, the **NAMELINK_ONLY** option installs nothing. It is an error to use this parameter outside of a **LIBRARY** block.

Changed in version 3.27: This parameter is also usable for an **ARCHIVE** block to manage the linker import file created, on macOS, for shared libraries with *ENABLE_EXPORTS* enabled.

When **NAMELINK_ONLY** is given, either **NAMELINK_COMPONENT** or **COMPONENT** may be used to specify the installation component of the namelink, but **COMPONENT** should generally be preferred.

NAMELINK_SKIP
Similar to **NAMELINK_ONLY**, but it has the opposite effect: it causes the installation of library files other than the namelink when a library target is installed. When neither **NAMELINK_ONLY** or **NAMELINK_SKIP** are given, both portions are installed. On platforms where versioned shared libraries do not have symlinks or when a library is not versioned, **NAMELINK_SKIP** installs the library. It is an error to use this parameter outside of a **LIBRARY** block.

Changed in version 3.27: This parameter is also usable for an **ARCHIVE** block to manage the linker import file created, on macOS, for shared libraries with *ENABLE_EXPORTS* enabled.

If **NAMELINK_SKIP** is specified, **NAMELINK_COMPONENT** has no effect. It is not recommended to use **NAMELINK_SKIP** in conjunction with **NAMELINK_COMPONENT**.

The *install(TARGETS)* command can also accept the following options at the top level:

EXPORT
This option associates the installed target files with an export called **<export−name>**. It must appear before any target options. To actually install the export file itself, call *install(EXPORT)*, documented below. See documentation of the*EXPORT_N AME* target property to change the name of the exported target.

If **EXPORT** is used and the targets include **PUBLIC** or **INTERFACE** file sets, all of them must be specified with **FILE_SET** arguments. All **PUBLIC** or **INTERFACE** file sets associated with a target are included in the export.

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

Unlike other **DESTINATION** arguments for the various **install()** subcommands, paths given after **INCLUDES DESTINATION** are used as given. They are not normalized, nor assumed to be normalized, although it is recommended that they are given in normalized form (see *Normalization*).

RUNTIME_DEPENDENCY_SET <set−name>
Added in version 3.21.

This option causes all runtime dependencies of installed executable, shared library, and module targets to be added to the specified runtime dependency set. This set can then be installed with an *install(RUNTIME_DEPENDENCY_SET)* command.

This keyword and the **RUNTIME_DEPENDENCIES** keyword are mutually exclusive.

RUNTIME_DEPENDENCIES <arg>...
Added in version 3.21.

This option causes all runtime dependencies of installed executable, shared library, and module targets to be installed along with the targets themselves. The **RUNTIME**, **LIBRARY**, **FRAMEWORK**, and generic arguments are used to determine the properties (**DESTINATION**, **COMPONENT**, etc.) of the installation of these dependencies.

RUNTIME_DEPENDENCIES is semantically equivalent to the following pair of calls:

```
install(TARGETS ... RUNTIME_DEPENDENCY_SET <set−name>)
install(RUNTIME_DEPENDENCY_SET <set−name> <arg>...)
```

where **<set−name>** will be a randomly generated set name. **<arg>...** may include any of

the following keywords supported by the *install(RUNTIME_DEPENDENCY_SET)* command:

- **DIRECTORIES**
- **PRE_INCLUDE_REGEXES**
- **PRE_EXCLUDE_REGEXES**
- **POST_INCLUDE_REGEXES**
- **POST_EXCLUDE_REGEXES**
- **POST_INCLUDE_FILES**
- **POST_EXCLUDE_FILES**

The **RUNTIME_DEPENDENCIES** and **RUNTIME_DEPENDENCY_SET** keywords are mutually exclusive.

Interface Libraries may be listed among the targets to install. They install no artifacts but will be included in an associated **EXPORT**. If*Object Libr aries* are listed but given no destination for their object files, they will be exported as *Interface Libraries*. This is sufficient to satisfy transitive usage requirements of other targets that link to the object libraries in their implementation.

Installing a target with the *EXCLUDE_FROM_ALL* target property set to **TRUE** has undefined behavior.

Added in version 3.3: An install destination given as a **DESTINATION** argument may use "generator expressions" with the syntax**$<...>**. See the*cmak e−generator−expressions(7)* manual for available expressions.

Added in version 3.13: *install(TARGETS)* can install targets that were created in other directories. When using such cross−directory install rules, running **make install** (or similar) from a subdirectory will not guarantee that targets from other directories are up−to−date. You can use *target_link_libraries()* or *add_dependencies()* to ensure that such out−of−directory targets are built before the subdirectory−specific install rules are run.

install(IMPORTED_RUNTIME_ARTIFACTS <target>... [...])

Added in version 3.21.

Install runtime artifacts of imported targets:

```
install(IMPORTED_RUNTIME_ARTIFACTS <target>...
        [RUNTIME_DEPENDENCY_SET <set−name>]
        [[LIBRARY|RUNTIME|FRAMEWORK|BUNDLE]
         [DESTINATION <dir>]
         [PERMISSIONS <permission>...]
         [CONFIGURATIONS <config>...]
         [COMPONENT <component>]
         [OPTIONAL] [EXCLUDE_FROM_ALL]
        ] [...]
        )
```

The **IMPORTED_RUNTIME_ARTIFACTS** form specifies rules for installing the runtime artifacts of imported targets. Projects may do this if they want to bundle outside executables or

modules inside their installation. The **LIBRARY**, **RUNTIME**, **FRAMEWORK**, and **BUNDLE** arguments have the same semantics that they do in the *TARGETS* mode. Only the runtime artifacts of imported targets are installed (except in the case of *FRAMEWORK* libraries, *MACOSX_BUNDLE* executables, and *BUNDLE* CFBundles.) For example, headers and import libraries associated with DLLs are not installed. In the case of *FRAMEWORK* libraries, *MACOSX_BUNDLE* executables, and *BUNDLE* CFBundles, the entire directory is installed.

The **RUNTIME_DEPENDENCY_SET** option causes the runtime artifacts of the imported executable, shared library, and module library **targets** to be added to the **<set−name>** runtime dependency set. This set can then be installed with an *install(RUNTIME_DEPENDENCY_SET)* command.

install(FILES <file>... [...])

install(PROGRAMS <program>... [...])

NOTE:
If installing header files, consider using file sets defined by *target_sources(FILE_SET)* instead. File sets associate headers with a target and they install as part of the target.

Install files or programs:

```
install(<FILES|PROGRAMS> <file>...
        TYPE <type> | DESTINATION <dir>
        [PERMISSIONS <permission>...]
        [CONFIGURATIONS <config>...]
        [COMPONENT <component>]
        [RENAME <name>] [OPTIONAL] [EXCLUDE_FROM_ALL])
```

The **FILES** form specifies rules for installing files for a project. File names given as relative paths are interpreted with respect to the current source directory. Files installed by this form are by default given permissions **OWNER_WRITE**, **OWNER_READ**, **GROUP_READ**, and **WORLD_READ** if no **PERMISSIONS** argument is given.

The **PROGRAMS** form is identical to the **FILES** form except that the default permissions for the installed file also include **OWNER_EXECUTE**, **GROUP_EXECUTE**, and **WORLD_EXECUTE**. This form is intended to install programs that are not targets, such as shell scripts. Use the **TARGETS** form to install targets built within the project.

The list of **files...** given to **FILES** or **PROGRAMS** may use "generator expressions" with the syntax **$<...>**. See the*cmak e−generator−expressions(7)* manual for available expressions. However, if any item begins in a generator expression it must evaluate to a full path.

The optional **RENAME <name>** argument is used to specify a name for the installed file that is different from the original file name. Renaming is allowed only when a single file is installed by the command.

Either a **TYPE** or a **DESTINATION** must be provided, but not both. A **TYPE** argument specifies the generic file type of the files being installed. A destination will then be set automatically by taking the corresponding variable from *GNUInstallDirs*, or by using a built−in default if that variable is not defined. See the table below for the supported file types and their corresponding variables and built−in defaults. Projects can provide a **DESTINATION** argument instead of a file type if they wish to explicitly define the install destination.

TYPE Argument	GNUInstallDirs Variable	Built−In Default
BIN	**${CMAKE_INSTALL_BINDIR}**	**bin**
SBIN	**${CMAKE_INSTALL_SBINDIR}**	**sbin**
LIB	**${CMAKE_INSTALL_LIBDIR}**	**lib**
INCLUDE	**${CMAKE_INSTALL_INCLUDEDIR}**	**include**
SYSCONF	**${CMAKE_INSTALL_SYSCONFDIR}**	**etc**
SHAREDSTATE	**${CMAKE_INSTALL_SHARESTATEDIR}**	**com**
LOCALSTATE	**${CMAKE_INSTALL_LOCALSTATEDIR}**	**var**
RUNSTATE	**${CMAKE_INSTALL_RUNSTATEDIR}**	**<LOCALSTATE dir>/run**
DATA	**${CMAKE_INSTALL_DATADIR}**	**<DATAROOT dir>**
INFO	**${CMAKE_INSTALL_INFODIR}**	**<DATAROOT dir>/info**
LOCALE	**${CMAKE_INSTALL_LOCALEDIR}**	**<DATAROOT dir>/locale**
MAN	**${CMAKE_INSTALL_MANDIR}**	**<DATAROOT dir>/man**
DOC	**${CMAKE_INSTALL_DOCDIR}**	**<DATAROOT dir>/doc**
LIBEXEC	**${CMAKE_INSTALL_LIBEXECDIR}**	**libexec**

Projects wishing to follow the common practice of installing headers into a project−specific subdirectory will need to provide a destination rather than rely on the above. Using file sets for headers instead of **install(FILES)** would be even better (see *target_sources(FILE_SET)*).

Note that some of the types' built−in defaults use the **DATAROOT** directory as a prefix. The **DATAROOT** prefix is calculated similarly to the types, with **CMAKE_INSTALL_DATAROOTDIR** as the variable and **share** as the built−in default. You cannot use **DATAROOT** as a **TYPE** parameter; please use **DATA** instead.

To make packages compliant with distribution filesystem layout policies, if projects must specify a **DESTINATION**, it is strongly recommended that they use a path that begins with the appropriate relative *GNUInstallDirs* variable. This allows package maintainers to control the install destination by setting the appropriate cache variables. The following example shows how to follow this advice while installing an image to a project−specific documentation subdirectory:

```
include(GNUInstallDirs)
install(FILES logo.png
        DESTINATION ${CMAKE_INSTALL_DOCDIR}/myproj
)
```

Added in version 3.4: An install destination given as a **DESTINATION** argument may use "generator expressions" with the syntax**$<...>**. See the*cmak e−generator−expressions(7)* manual for available expressions.

Added in version 3.20: An install rename given as a **RENAME** argument may use "generator expressions" with the syntax**$<...>**. See the*cmak e−generator−expressions(7)* manual for available expressions.

Added in version 3.31: The **TYPE** argument now supports type **LIBEXEC**.

install(DIRECTORY <dir>... [...])

NOTE:
To install a directory sub−tree of headers, consider using file sets defined by *target_sources(FILE_SET)* instead. File sets not only preserve directory structure, they also associate headers with a target and install as part of the target.

Install the contents of one or more directories:

```
install(DIRECTORY dirs...
        TYPE <type> | DESTINATION <dir>
        [FILE_PERMISSIONS <permission>...]
        [DIRECTORY_PERMISSIONS <permission>...]
        [USE_SOURCE_PERMISSIONS] [OPTIONAL] [MESSAGE_NEVER]
        [CONFIGURATIONS <config>...]
        [COMPONENT <component>] [EXCLUDE_FROM_ALL]
        [FILES_MATCHING]
        [[PATTERN <pattern> | REGEX <regex>]
         [EXCLUDE] [PERMISSIONS <permission>...]] [...])
```

The **DIRECTORY** form installs contents of one or more directories to a given destination. The directory structure is copied verbatim to the destination. The last component of each directory name is appended to the destination directory but a trailing slash may be used to avoid this because it leaves the last component empty. Directory names given as relative paths are interpreted with respect to the current source directory. If no input directory names are given the destination directory will be created but nothing will be installed into it. The **FILE_PERMISSIONS** and **DIRECTORY_PERMISSIONS** options specify permissions given to files and directories in the destination. If**USE_SOURCE_PERMISSIONS** is specified and **FILE_PERMISSIONS** is not, file permissions will be copied from the source directory structure. If no permissions are specified files will be given the default permissions specified in the **FILES** form of the command, and the directories will be given the default permissions specified in the **PROGRAMS** form of the command.

Added in version 3.1: The **MESSAGE_NEVER** option disables file installation status output.

Installation of directories may be controlled with fine granularity using the **PATTERN** or **REGEX** options. These "match" options specify a globbing pattern or regular expression to match directories or files encountered within input directories. They may be used to apply certain options (see below) to a subset of the files and directories encountered. The full path to each input file or directory (with forward slashes) is matched against the expression. A**P ATTERN** will match only complete file names: the portion of the full path matching the pattern must occur at the end of the file name and be preceded by a slash. A **REGEX** will match any portion of the full path but it may

use / and **$** to simulate the **PATTERN** behavior. By default all files and directories are installed whether or not they are matched. The **FILES_MATCHING** option may be given before the first match option to disable installation of files (but not directories) not matched by any expression. For example, the code

```
install(DIRECTORY src/ DESTINATION doc/myproj
        FILES_MATCHING PATTERN "*.png")
```

will extract and install images from a source tree.

Some options may follow a **PATTERN** or **REGEX** expression as described under *string(REGEX)* and are applied only to files or directories matching them. The **EXCLUDE** option will skip the matched file or directory. The**PERMISSIONS** option o verrides the permissions setting for the matched file or directory. For example the code

```
install(DIRECTORY icons scripts/ DESTINATION share/myproj
        PATTERN "CVS" EXCLUDE
        PATTERN "scripts/*"
        PERMISSIONS OWNER_EXECUTE OWNER_WRITE OWNER_READ
                    GROUP_EXECUTE GROUP_READ)
```

will install the **icons** directory to **share/myproj/icons** and the **scripts** directory to **share/myproj**. The icons will get default file permissions, the scripts will be given specific permissions, and any **CVS** directories will be excluded.

Either a **TYPE** or a **DESTINATION** must be provided, but not both. A **TYPE** argument specifies the generic file type of the files within the listed directories being installed. A destination will then be set automatically by taking the corresponding variable from *GNUInstallDirs*, or by using a built−in default if that variable is not defined. See the table below for the supported file types and their corresponding variables and built−in defaults. Projects can provide a **DESTINATION** argument instead of a file type if they wish to explicitly define the install destination.

TYPE Argument	GNUInstallDirs Variable	Built−In Default
BIN	**${CMAKE_INSTALL_BINDIR}**	**bin**
SBIN	**${CMAKE_INSTALL_SBINDIR}**	**sbin**
LIB	**${CMAKE_INSTALL_LIBDIR}**	**lib**
INCLUDE	**${CMAKE_INSTALL_INCLUDEDIR}**	**include**
SYSCONF	**${CMAKE_INSTALL_SYSCONFDIR}**	**etc**
SHAREDSTATE	**${CMAKE_INSTALL_SHARESTATEDIR}**	**com**
LOCALSTATE	**${CMAKE_INSTALL_LOCALSTATEDIR}**	**var**
RUNSTATE	**${CMAKE_INSTALL_RUNSTATEDIR}**	**<LOCALSTATE dir>/run**
DATA	**${CMAKE_INSTALL_DATADIR}**	**<DATAROOT dir>**
INFO	**${CMAKE_INSTALL_INFODIR}**	**<DATAROOT dir>/info**
LOCALE	**${CMAKE_INSTALL_LOCALEDIR}**	**<DATAROOT dir>/locale**
MAN	**${CMAKE_INSTALL_MANDIR}**	**<DATAROOT dir>/man**
DOC	**${CMAKE_INSTALL_DOCDIR}**	**<DATAROOT dir>/doc**
LIBEXEC	**${CMAKE_INSTALL_LIBEXECDIR}**	**libexec**

Note that some of the types' built−in defaults use the **DATAROOT** directory as a prefix. The **DATAROOT** prefix is calculated similarly to the types, with **CMAKE_INSTALL_DATAROOTDIR** as the variable and **share** as the built−in default. You cannot use **DATAROOT** as a **TYPE** parameter; please use **DATA** instead.

To make packages compliant with distribution filesystem layout policies, if projects must specify a **DESTINATION**, it is strongly recommended that they use a path that begins with the appropriate relative *GNUInstallDirs* variable. This allows package maintainers to control the install destination by setting the appropriate cache variables.

Added in version 3.4: An install destination given as a **DESTINATION** argument may use "generator expressions" with the syntax**$<...>**. See the*cmak e−generator−expressions(7)* manual for available expressions.

Added in version 3.5: The list of **dirs...** given to **DIRECTORY** may use "generator expressions" too.

Added in version 3.31: The **TYPE** argument now supports type **LIBEXEC**.

install(SCRIPT <file> [...])

install(CODE <code> [...])

Invoke CMake scripts or code during installation:

```
install([[SCRIPT <file>] [CODE <code>]]
        [ALL_COMPONENTS | COMPONENT <component>]
        [EXCLUDE_FROM_ALL] [...])
```

The **SCRIPT** form will invoke the given CMake script files during installation. If the script file name is a relative path it will be interpreted with respect to the current source directory. The **CODE** form will invoke the given CMake code during installation. Code is specified as a single argument inside a double−quoted string. For example, the code

```
install(CODE "message(\"Sample install message.\")")
```

will print a message during installation.

Added in version 3.21: When the **ALL_COMPONENTS** option is given, the custom installation script code will be executed for every component of a component−specific installation. This option is mutually exclusive with the **COMPONENT** option.

Added in version 3.14: **<file>** or **<code>** may use "generator expressions" with the syntax**$<...>** (in the case of **<file>**, this refers to their use in the file name, not the file's contents). See the *cmake−generator−expressions(7)* manual for available expressions.

install(EXPORT <export−name> [...])

Install a CMake file exporting targets for dependent projects:

```
install(EXPORT <export−name> DESTINATION <dir>
        [NAMESPACE <namespace>] [FILE <name>.cmake]
        [PERMISSIONS <permission>...]
        [CONFIGURATIONS <config>...]
        [CXX_MODULES_DIRECTORY <directory>]
        [EXPORT_LINK_INTERFACE_LIBRARIES]
        [COMPONENT <component>]
        [EXCLUDE_FROM_ALL]
        [EXPORT_PACKAGE_DEPENDENCIES])
install(EXPORT_ANDROID_MK <export−name> DESTINATION <dir> [...])
```

The **EXPORT** form generates and installs a CMake file containing code to import targets from the installation tree into another project. Target installations are associated with the export **<export−name>** using the **EXPORT** option of the *install(TARGETS)* signature documented above. The **NAMESPACE** option will prepend **<namespace>** to the target names as they are written to the import file. By default the generated file will be called **<export−name>.cmake** but the **FILE** option may be used to specify a different name. The value given to the **FILE** option must be a file name with the **.cmake** extension.

If a **CONFIGURATIONS** option is given then the file will only be installed when one of the named configurations is installed. Additionally, the generated import file will reference only the matching target configurations. See the *CMAKE_MAP_IMPORTED_CONFIG_<CONFIG>* variable to map configurations of dependent projects to the installed configurations. The **EXPORT_LINK_INTERFACE_LIBRARIES** keyword, if present, causes the contents of the properties matching **(IMPORTED_)?LINK_INTERFACE_LIBRARIES(_<CONFIG>)?** to be

exported, when policy *CMP0022* is **NEW**.

NOTE:
The installed **<export−name>.cmake** file may come with additional per−configuration **<export−name>−*.cmake** files to be loaded by globbing. Do not use an export name that is the same as the package name in combination with installing a **<package−name>−config.cmake** file or the latter may be incorrectly matched by the glob and loaded.

When a **COMPONENT** option is given, the listed **<component>** implicitly depends on all components mentioned in the export set. The exported **<name>.cmake** file will require each of the exported components to be present in order for dependent projects to build properly. For example, a project may define components **Runtime** and **Development**, with shared libraries going into the **Runtime** component and static libraries and headers going into the **Development** component. The export set would also typically be part of the **Development** component, but it would export targets from both the **Runtime** and **Development** components. Therefore, the **Runtime** component would need to be installed if the **Development** component was installed, but not vice versa. If the **Development** component was installed without the **Runtime** component, dependent projects that try to link against it would have build errors. Package managers, such as APT and RPM, typically handle this by listing the **Runtime** component as a dependency of the **Development** component in the package metadata, ensuring that the library is always installed if the headers and CMake export file are present.

Added in version 3.7: In addition to cmake language files, the **EXPORT_ANDROID_MK** mode may be used to specify an export to the android ndk build system. This mode accepts the same options as the normal export mode. The Android NDK supports the use of prebuilt libraries, both static and shared. This allows cmake to build the libraries of a project and make them available to an ndk build system complete with transitive dependencies, include flags and defines required to use the libraries.

CXX_MODULES_DIRECTORY
Added in version 3.28.

Specify a subdirectory to store C++ module information for targets in the export set. This directory will be populated with files which add the necessary target property information to the relevant targets. Note that without this information, none of the C++ modules which are part of the targets in the export set will support being imported in consuming targets.

EXPORT_PACKAGE_DEPENDENCIES

NOTE:
Experimental. Gated by **CMAKE_EXPERIMENTAL_EXPORT_PACKAGE_DEPENDENCIES**.

Specify that *find_dependency()* calls should be exported. If this argument is specified, CMake examines all targets in the export set and gathers their **INTERFACE** link targets. If any such targets either were found with *find_package()* or have the *EXPORT_FIND_PACKAGE_NAME* property set, and such package dependency was not disabled by passing **ENABLED OFF** to *export(SETUP)*, then a *find_dependency()* call is written with the target's corresponding package name, a **REQUIRED** argument, and any additional arguments specified by the **EXTRA_ARGS** argument of *export(SETUP)*. Any package dependencies that were manually specified by passing **ENABLED ON** to *export(SETUP)* are also added, even if the exported targets don't depend on any targets from them.

The *find_dependency()* calls are written in the following order:

1. Any package dependencies that were listed in *export(SETUP)* are written in the order they were first specified, regardless of whether or not they contain **INTERFACE** dependencies of the exported targets.
2. Any package dependencies that contain **INTERFACE** link dependencies of the exported targets and that were never specified in *export(SETUP)* are written in the order they were first found.

The **EXPORT** form is useful to help outside projects use targets built and installed by the current project. For example, the code

```
install(TARGETS myexe EXPORT myproj DESTINATION bin)
install(EXPORT myproj NAMESPACE mp_ DESTINATION lib/myproj)
install(EXPORT_ANDROID_MK myproj DESTINATION share/ndk-modules)
```

will install the executable **myexe** to **<prefix>/bin** and code to import it in the file **<prefix>/lib/myproj/myproj.cmake** and **<prefix>/share/ndk-modules/Android.mk**. An outside project may load this file with the include command and reference the **myexe** executable from the installation tree using the imported target name **mp_myexe** as if the target were built in its own tree.

install(PACKAGE_INFO <package-name> [...])

Added in version 3.31.

NOTE:

Experimental. Gated by **CMAKE_EXPERIMENTAL_EXPORT_PACKAGE_INFO**.

Installs a *Common Package Specification* file exporting targets for dependent projects:

```
install(PACKAGE_INFO <package-name> EXPORT <export-name>
        [APPENDIX <appendix-name>]
        [DESTINATION <dir>]
        [LOWER_CASE_FILE]
        [VERSION <version>
         [COMPAT_VERSION <version>]
         [VERSION_SCHEMA <string>]]
        [DEFAULT_TARGETS <target>...]
        [DEFAULT_CONFIGURATIONS <config>...]
        [PERMISSIONS <permission>...]
        [CONFIGURATIONS <config>...]
        [COMPONENT <component>]
        [EXCLUDE_FROM_ALL])
```

The **PACKAGE_INFO** form generates and installs a Common Package Specification file which describes installed targets such that they can be consumed by another project. Target installations are associated with the export **<export-name>** using the **EXPORT** option of the *install(TARGETS)* signature documented above. Unlike *install(EXPORT)*, this information is not expressed in CMake code, and can be consumed by tools other than CMake. When imported into another CMake project, the imported targets will be prefixed with **<package-name>::**. By default, the generated file will be called **<package-name>[-<appendix-name>].cps**. If **LOWER_CASE_FILE** is given, the package name as it appears on disk (in both the file name and install destination) will be first converted to lower case.

If **DESTINATION** is not specified, a platform−specific default is used.

If **APPENDIX** is specified, rather than generating a top level package specification, the specified targets will be exported as an appendix to the named package. Appendices may be used to separate less commonly used targets (along with their external dependencies) from the rest of a package. This enables consumers to ignore transitive dependencies for targets that they don't use, and also allows a single logical "package" to be composed of artifacts produced by multiple build trees.

Appendices are not permitted to change basic package metadata; therefore, none of **VERSION**, **COMPAT_VERSION**, **VERSION_SCHEMA**, **DEFAULT_TARGETS** or **DEFAULT_CONFIGURATIONS** may be specified in combination with **APPENDIX**. Additionally, it is strongly recommended that use of **LOWER_CASE_FILE** should be consistent between the main package and any appendices.

install(RUNTIME_DEPENDENCY_SET <set−name> [...])

Added in version 3.21.

Installs a runtime dependency set:

```
install(RUNTIME_DEPENDENCY_SET <set-name>
        [[LIBRARY|RUNTIME|FRAMEWORK]
         [DESTINATION <dir>]
         [PERMISSIONS <permission>...]
         [CONFIGURATIONS <config>...]
         [COMPONENT <component>]
         [NAMELINK_COMPONENT <component>]
         [OPTIONAL] [EXCLUDE_FROM_ALL]
        ] [...]
        [PRE_INCLUDE_REGEXES <regex>...]
        [PRE_EXCLUDE_REGEXES <regex>...]
        [POST_INCLUDE_REGEXES <regex>...]
        [POST_EXCLUDE_REGEXES <regex>...]
        [POST_INCLUDE_FILES <file>...]
        [POST_EXCLUDE_FILES <file>...]
        [DIRECTORIES <dir>...]
        )
```

Installs a runtime dependency set previously created by one or more *install(TARGETS)* or *install(IMPORTED_RUNTIME_ARTIFACTS)* commands. The dependencies of targets belonging to a runtime dependency set are installed in the **RUNTIME** destination and component on DLL platforms, and in the **LIBRARY** destination and component on non−DLL platforms. macOS frameworks are installed in the **FRAMEWORK** destination and component. Targets built within the build tree will never be installed as runtime dependencies, nor will their own dependencies, unless the targets themselves are installed with *install(TARGETS)*.

The generated install script calls *file(GET_RUNTIME_DEPENDENCIES)* on the build−tree files to calculate the runtime dependencies. The build−tree executable files are passed as the **EXECUTABLES** argument, the build−tree shared libraries as the **LIBRARIES** argument, and the build−tree modules as the **MODULES** argument. On macOS, if one of the executables is a *MACOSX_BUNDLE*, that executable is passed as the **BUNDLE_EXECUTABLE** argument. At most one such bundle executable may be in the runtime dependency set on macOS. The *MACOSX_BUNDLE* property has no effect on other platforms. Note that *file(GET_RUNTIME_DEPENDENCIES)* only supports collecting the runtime dependencies for Windows, Linux and macOS platforms, so **install(RUNTIME_DEPENDENCY_SET)** has the

same limitation.

The following sub−arguments are forwarded through as the corresponding arguments to *file(GET_RUNTIME_DEPENDENCIES)* (for those that provide a non−empty list of directories, regular expressions or files). They all support *generator expressions*.

- **DIRECTORIES <dir>...**
- **PRE_INCLUDE_REGEXES <regex>...**
- **PRE_EXCLUDE_REGEXES <regex>...**
- **POST_INCLUDE_REGEXES <regex>...**
- **POST_EXCLUDE_REGEXES <regex>...**
- **POST_INCLUDE_FILES <file>...**
- **POST_EXCLUDE_FILES <file>...**

NOTE:

This command supersedes the *install_targets()* command and the *PRE_INSTALL_SCRIPT* and *POST_INSTALL_SCRIPT* target properties. It also replaces the **FILES** forms of the *install_files()* and *install_programs()* commands. The processing order of these install rules relative to those generated by *install_targets()*, *install_files()*, and *install_programs()* commands is not defined.

Examples

Example: Install Targets with Per−Artifact Components

Consider a project that defines targets with different artifact kinds:

```
add_executable(myExe myExe.c)
add_library(myStaticLib STATIC myStaticLib.c)
target_sources(myStaticLib PUBLIC FILE_SET HEADERS FILES myStaticLib.h)
add_library(mySharedLib SHARED mySharedLib.c)
target_sources(mySharedLib PUBLIC FILE_SET HEADERS FILES mySharedLib.h)
set_property(TARGET mySharedLib PROPERTY SOVERSION 1)
```

We may call *install(TARGETS)* with *<artifact−kind>* arguments to specify different options for each kind of artifact:

```
install(TARGETS
          myExe
          mySharedLib
          myStaticLib
        RUNTIME           # Following options apply to runtime artifacts.
          COMPONENT Runtime
        LIBRARY           # Following options apply to library artifacts.
          COMPONENT Runtime
          NAMELINK_COMPONENT Development
        ARCHIVE           # Following options apply to archive artifacts.
          COMPONENT Development
          DESTINATION lib/static
        FILE_SET HEADERS  # Following options apply to file set HEADERS.
          COMPONENT Development
        )
```

This will:

- Install **myExe** to **<prefix>/bin**, the default RUNTIME artifact destination, as part of the **Runtime** component.
- On non−DLL platforms:
  - Install **libmySharedLib.so.1** to **<prefix>/lib**, the default LIBRARY artifact destination, as part of the **Runtime** component.
  - Install the **libmySharedLib.so** "namelink" (symbolic link) to **<prefix>/lib**, the default LIBRARY artifact destination, as part of the **Development** component.
- On DLL platforms:
  - Install **mySharedLib.dll** to **<prefix>/bin**, the default RUNTIME artifact destination, as part of the **Runtime** component.
  - Install **mySharedLib.lib** to **<prefix>/lib/static**, the specified ARCHIVE artifact destination, as part of the **Development** component.
- Install **myStaticLib** to **<prefix>/lib/static**, the specified ARCHIVE artifact destination, as part of the **Development** component.
- Install **mySharedLib.h** and **myStaticLib.h** to **<prefix>/include**, the default destination for a file set of type HEADERS, as part of the **Development** component.

Example: Install Targets to Per−Config Destinations

Each *install(TARGETS)* call installs a given target *output artifact* to at most one **DESTINATION**, but the install rule itself may be filtered by the **CONFIGURATIONS** option. In order to install to a different destination for each configuration, one call per configuration is needed. For example, the code:

```
install(TARGETS myExe
        CONFIGURATIONS Debug
        RUNTIME
          DESTINATION Debug/bin
        )
install(TARGETS myExe
        CONFIGURATIONS Release
        RUNTIME
          DESTINATION Release/bin
        )
```

will install **myExe** to **<prefix>/Debug/bin** in the Debug configuration, and to **<prefix>/Release/bin** in the Release configuration.

Generated Installation Script

NOTE:

Use of this feature is not recommended. Please consider using the *cmake −−install* instead.

The **install()** command generates a file, **cmake_install.cmake**, inside the build directory, which is used internally by the generated install target and by CPack. You can also invoke this script manually with *cmake −P*. This script accepts several variables:

COMPONENT

Set this variable to install only a single CPack component as opposed to all of them. For example, if you only want to install the **Development** component, run **cmake −DCOMPONENT=Development −P cmake_install.cmake**.

BUILD_TYPE

Set this variable to change the build type if you are using a multi−config generator. For example, to install with the **Debug** configuration, run **cmake −DBUILD_TYPE=Debug −P cmake_install.cmake**.

DESTDIR
This is an environment variable rather than a CMake variable. It allows you to change the installation prefix on UNIX systems. See *DESTDIR* for details.

link_directories
Add directories in which the linker will look for libraries.

```
link_directories([AFTER|BEFORE] directory1 [directory2 ...])
```

Adds the paths in which the linker should search for libraries. Relative paths given to this command are interpreted as relative to the current source directory, see *CMP0015*.

The command will apply only to targets created after it is called.

Added in version 3.13: The directories are added to the *LINK_DIRECTORIES* directory property for the current **CMakeLists.txt** file, converting relative paths to absolute as needed. See the *cmake−buildsystem(7)* manual for more on defining buildsystem properties.

Added in version 3.13: By default the directories specified are appended onto the current list of directories. This default behavior can be changed by setting *CMAKE_LINK_DIRECTORIES_BEFORE* to **ON**. By using **AFTER** or **BEFORE** explicitly, you can select between appending and prepending, independent of the default.

Added in version 3.13: Arguments to **link_directories** may use "generator expressions" with the syntax "$<...>". See the*cmak e−generator−expressions(7)* manual for available expressions.

NOTE:
This command is rarely necessary and should be avoided where there are other choices. Prefer to pass full absolute paths to libraries where possible, since this ensures the correct library will always be linked. The*find_libr ary()* command provides the full path, which can generally be used directly in calls to *target_link_libraries()*. Situations where a library search path may be needed include:

- Project generators like Xcode where the user can switch target architecture at build time, but a full path to a library cannot be used because it only provides one architecture (i.e. it is not a universal binary).
- Libraries may themselves have other private library dependencies that expect to be found via **RPATH** mechanisms, but some linkers are not able to fully decode those paths (e.g. due to the presence of things like **$ORIGIN**).

If a library search path must be provided, prefer to localize the effect where possible by using the *target_link_directories()* command rather than **link_directories()**. The target−specific command can also control how the search directories propagate to other dependent targets.

See Also
- *target_link_directories()*
- *target_link_libraries()*

link_libraries
Link libraries to all targets added later.

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

Specify libraries or flags to use when linking any targets created later in the current directory or below by commands such as *add_executable()* or *add_library()*. See the*tar get_link_libraries()* command for meaning of arguments.

NOTE:
The *target_link_libraries()* command should be preferred whenever possible. Library dependencies are chained automatically, so directory−wide specification of link libraries is rarely needed.

project
Set the name of the project.

Synopsis

```
project(<PROJECT-NAME> [<language-name>...])
project(<PROJECT-NAME>
        [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]
        [DESCRIPTION <project-description-string>]
        [HOMEPAGE_URL <url-string>]
        [LANGUAGES <language-name>...])
```

Sets the name of the project, and stores it in the variable *PROJECT_NAME*. When called from the top−level **CMakeLists.txt** also stores the project name in the variable *CMAKE_PROJECT_NAME*.

Also sets the variables:

*PROJECT_SOURCE_DIR***,** *<PROJECT−NAME>_SOURCE_DIR*
Absolute path to the source directory for the project.

*PROJECT_BINARY_DIR***,** *<PROJECT−NAME>_BINARY_DIR*
Absolute path to the binary directory for the project.

*PROJECT_IS_TOP_LEVEL***,** *<PROJECT−NAME>_IS_TOP_LEVEL*
Added in version 3.21.

Boolean value indicating whether the project is top−level.

Further variables are set by the optional arguments described in *Options* further below. Where an option is not given, its corresponding variable is set to the empty string.

Note that variables of the form **<name>_SOURCE_DIR** and **<name>_BINARY_DIR** may also be set by other commands before **project()** is called (see the *FetchContent_MakeAvailable()* command for one example). Projects should not rely on**<PR OJECT−NAME>_SOURCE_DIR** or **<PROJECT−NAME>_BINARY_DIR** holding a particular value outside of the scope of the call to **project()** or one of its child scopes.

Changed in version 3.30: **<PROJECT−NAME>_SOURCE_DIR**, **<PROJECT−NAME>_BINARY_DIR**, and **<PROJECT−NAME>_IS_TOP_LEVEL**, if already set as normal variables when **project(<PROJECT−NAME> ...)** is called, are updated by the call. Cache entries by the same names are always set as before. See release notes for 3.30.3, 3.30.4, and 3.30.5 for details.

Changed in version 3.31: **<PROJECT−NAME>_SOURCE_DIR**, **<PROJECT−NAME>_BINARY_DIR**, and **<PROJECT−NAME>_IS_TOP_LEVEL** are always set as normal variables by **project(<PROJECT−NAME> ...)**. See policy *CMP0180*. Cache entries by the same names are always set as before.

Options

The options are:

VERSION <version>

Optional; may not be used unless policy *CMP0048* is set to **NEW**.

Takes a **<version>** argument composed of non−negative integer components, i.e. **<major>[.<minor>[.<patch>[.<tweak>]]]**, and sets the variables

- *PROJECT_VERSION*, *<PROJECT−NAME>_VERSION*
- *PROJECT_VERSION_MAJOR*, *<PROJECT−NAME>_VERSION_MAJOR*
- *PROJECT_VERSION_MINOR*, *<PROJECT−NAME>_VERSION_MINOR*
- *PROJECT_VERSION_PATCH*, *<PROJECT−NAME>_VERSION_PATCH*
- *PROJECT_VERSION_TWEAK*, *<PROJECT−NAME>_VERSION_TWEAK*.

Added in version 3.12: When the **project()** command is called from the top−level **CMakeLists.txt**, then the version is also stored in the variable *CMAKE_PROJECT_VERSION*.

DESCRIPTION <project−description−string>

Added in version 3.9.

Optional. Sets the variables

- *PROJECT_DESCRIPTION*, *<PROJECT−NAME>_DESCRIPTION*

to **<project−description−string>**. It is recommended that this description is a relatively short string, usually no more than a few words.

When the **project()** command is called from the top−level **CMakeLists.txt**, then the description is also stored in the variable *CMAKE_PROJECT_DESCRIPTION*.

Added in version 3.12: Added the **<PROJECT−NAME>_DESCRIPTION** variable.

HOMEPAGE_URL <url−string>

Added in version 3.12.

Optional. Sets the variables

- *PROJECT_HOMEPAGE_URL*, *<PROJECT−NAME>_HOMEPAGE_URL*

to **<url−string>**, which should be the canonical home URL for the project.

When the **project()** command is called from the top−level **CMakeLists.txt**, then the URL also is stored in the variable *CMAKE_PROJECT_HOMEPAGE_URL*.

LANGUAGES <language−name>...

Optional. Can also be specified without**LANGU AGES** keyword per the first, short signature.

Selects which programming languages are needed to build the project.

Supported languages are **C**, **CXX** (i.e. C++), **CSharp** (i.e. C#), **CUDA**, **OBJC** (i.e. Objective−C), **OBJCXX** (i.e. Objective−C++), **Fortran**, **HIP**, **ISPC**, **Swift**, **ASM**, **ASM_NASM**, **ASM_MARMASM**, **ASM_MASM**, and **ASM−ATT**.

Added in version 3.8: Added **CSharp** and **CUDA** support.

Added in version 3.15: Added **Swift** support.

Added in version 3.16: Added **OBJC** and **OBJCXX** support.

Added in version 3.18: Added **ISPC** support.

Added in version 3.21: Added **HIP** support.

Added in version 3.26: Added **ASM_MARMASM** support.

If enabling **ASM**, list it last so that CMake can check whether compilers for other languages like **C** work for assembly too.

By default **C** and **CXX** are enabled if no language options are given. Specify language**NONE**, or use the **LANGUAGES** keyword and list no languages, to skip enabling any languages.

The variables set through the **VERSION**, **DESCRIPTION** and **HOMEPAGE_URL** options are intended for use as default values in package metadata and documentation.

Code Injection

A number of variables can be defined by the user to specify files to include at different points during the execution of the **project()** command. The following outlines the steps performed during a **project()** call:

- Added in version 3.15: For every **project()** call regardless of the project name, include the file(s) and module(s) named by *CMAKE_PROJECT_INCLUDE_BEFORE*, if set.

- Added in version 3.17: If the **project()** command specifies **<PROJECT−NAME>** as its project name, include the file(s) and module(s) named by *CMAKE_PROJECT_<PROJECT−NAME>_INCLUDE_BEFORE*, if set.

- Set the various project−specific variables detailed in the *Synopsis* and *Options* sections above.
- For the very first **project()** call only:
  - If *CMAKE_TOOLCHAIN_FILE* is set, read it at least once. It may be read multiple times and it may also be read again when enabling languages later (see below).
  - Set the variables describing the host and target platforms. Language−specific variables might or might not be set at this point. On the first run, the only language−specific variables that might be defined are those a toolchain file may have set. On subsequent runs, language−specific variables cached from a previous run may be set.
  - Added in version 3.24: Include each file listed in *CMAKE_PROJECT_TOP_LEVEL_INCLUDES*, if set. The variable is ignored by CMake thereafter.

- Enable any languages specified in the call, or the default languages if none were provided. The toolchain file may be re−read when enabling a language for the first time.

- Added in version 3.15: For every **project()** call regardless of the project name, include the file(s) and module(s) named by *CMAKE_PROJECT_INCLUDE*, if set.

- If the **project()** command specifies **<PROJECT−NAME>** as its project name, include the file(s) and module(s) named by *CMAKE_PROJECT_<PROJECT−NAME>_INCLUDE*, if set.

Usage

The top−level **CMakeLists.txt** file for a project must contain a literal, direct call to the **project()** command; loading one through the *include()* command is not sufficient. If no such call exists, CMake will issue a warning and pretend there is a **project(Project)** at the top to enable the default languages (**C** and **CXX**).

NOTE:

Call the **project()** command near the top of the top−level **CMakeLists.txt**, but *after* calling *cmake_minimum_required()*. It is important to establish version and policy settings before invoking other commands whose behavior they may affect and for this reason the **project()** command will issue a warning if this order is not kept. See also policy *CMP0000*.

remove_definitions

Remove −D define flags added by *add_definitions()*.

```
remove_definitions(−DFOO −DBAR ...)
```

Removes flags (added by *add_definitions()*) from the compiler command line for sources in the current directory and below.

set_source_files_properties

Source files can have properties that affect how they are built.

```
set_source_files_properties(<files> ...
                            [DIRECTORY <dirs> ...]
                            [TARGET_DIRECTORY <targets> ...]
                            PROPERTIES <prop1> <value1>
                            [<prop2> <value2>] ...)
```

Sets properties associated with source files using a key/value paired list.

Added in version 3.18: By default, source file properties are only visible to targets added in the same directory (**CMakeLists.txt**). Visibility can be set in other directory scopes using one or both of the following options:

DIRECTORY <dirs>...

The source file properties will be set in each of the **<dirs>** directories' scopes. CMake must already know about each of these source directories, either by having added them through a call to *add_subdirectory()* or it being the top level source directory. Relative paths are treated as relative to the current source directory.

TARGET_DIRECTORY <targets>...

The source file properties will be set in each of the directory scopes where any of the specified **<targets>** were created (the **<targets>** must therefore already exist).

Use *get_source_file_property()* to get property values. See also the*set_pr operty(SOURCE)* command.

NOTE:

The *GENERATED* source file property may be globally visible. See its documentation for details.

See Also

- *define_property()*
- *get_source_file_property()*
- *Properties on Source Files* for the list of properties known to CMake

set_target_properties

Targets can have properties that affect how they are built.

```
set_target_properties(<targets> ...
                      PROPERTIES <prop1> <value1>
                      [<prop2> <value2>] ...)
```

Sets properties on targets. The syntax for the command is to list all the targets you want to change, and then provide the values you want to set next. You can use any prop value pair you want and extract it later with the *get_property()* or *get_target_property()* command.

Alias Targets do not support setting target properties.

See Also

- *define_property()*
- *get_target_property()*
- the more general *set_property()* command
- *Properties on Targets* for the list of properties known to CMake

set_tests_properties

Set a property of the tests.

```
set_tests_properties(<tests>...
                     [DIRECTORY <dir>]
                     PROPERTIES <prop1> <value1>
                     [<prop2> <value2>]...)
```

Sets a property for the tests. If the test is not found, CMake will report an error.

Test property values may be specified using *generator expressions* for tests created by the *add_test(NAME)* signature.

Added in version 3.28: Visibility can be set in other directory scopes using the following option:

DIRECTORY <dir>

The test properties will be set in the **<dir>** directory's scope. CMake must already know about this directory, either by having added it through a call to *add_subdirectory()* or it being the top level source directory. Relative paths are treated as relative to the current source directory. **<dir>** may reference a binary directory.

See Also

- *add_test()*
- *define_property()*
- the more general *set_property()* command
- *Properties on Tests* for the list of properties known to CMake

source_group

Define a grouping for source files in IDE project generation. There are two different signatures to create source groups.

```
source_group(<name> [FILES <src>...] [REGULAR_EXPRESSION <regex>])
source_group(TREE <root> [PREFIX <prefix>] [FILES <src>...])
```

Defines a group into which sources will be placed in project files. This is intended to set up file tabs in Visual Studio. The group is scoped in the directory where the command is called, and applies to sources in targets created in that directory.

The options are:

TREE Added in version 3.8.

CMake will automatically detect, from **<src>** files paths, source groups it needs to create, to keep structure of source groups analogically to the actual files and directories structure in the project. Paths of **<src>** files will be cut to be relative to **<root>**. The command fails if the paths within **src** do not start with **root**.

PREFIX
Added in version 3.8.

Source group and files located directly in **<root>** path, will be placed in **<prefix>** source groups.

FILES Any source file specified explicitly will be placed in group **<name>**. Relative paths are interpreted with respect to the current source directory.

REGULAR_EXPRESSION
Any source file whose name matches the regular expression will be placed in group **<name>**.

If a source file matches multiple groups, the *last* group that explicitly lists the file with **FILES** will be favored, if any. If no group explicitly lists the file, the *last* group whose regular expression matches the file will be favored.

The **<name>** of the group and **<prefix>** argument may contain forward slashes or backslashes to specify subgroups. Backslashes need to be escaped appropriately:

```
source_group(base/subdir ...)
source_group(outer\\inner ...)
source_group(TREE <root> PREFIX sources\\inc ...)
```

Added in version 3.18: Allow using forward slashes (/) to specify subgroups.

For backwards compatibility, the short−hand signature

```
source_group(<name> <regex>)
```

is equivalent to

```
source_group(<name> REGULAR_EXPRESSION <regex>)
```

target_compile_definitions
Add compile definitions to a target.

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

Specifies compile definitions to use when compiling a given **<target>**. The named**<tar get>** must have been created by a command such as *add_executable()* or *add_library()* and must not be an *ALIAS target*.

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

Added in version 3.11: Allow setting **INTERFACE** items on *IMPORTED targets*.

Arguments to **target_compile_definitions** may use generator expressions with the syntax **$<...>**. See the *cmake−generator−expressions(7)* manual for available expressions. See the*cmak e−buildsystem(7)* manual for more on defining buildsystem properties.

Any leading **−D** on an item will be removed. Empty items are ignored. For example, the following are all equivalent:

```
target_compile_definitions(foo PUBLIC FOO)
target_compile_definitions(foo PUBLIC −DFOO)  # −D removed
target_compile_definitions(foo PUBLIC "" FOO) # "" ignored
target_compile_definitions(foo PUBLIC −D FOO) # −D becomes "", then ignored
```

Definitions may optionally have values:

```
target_compile_definitions(foo PUBLIC FOO=1)
```

Note that many compilers treat **−DFOO** as equivalent to **−DFOO=1**, but other tools may not recognize this in all circumstances (e.g. IntelliSense).

See Also

- *add_compile_definitions()*
- *target_compile_features()*
- *target_compile_options()*
- *target_include_directories()*
- *target_link_libraries()*
- *target_link_directories()*
- *target_link_options()*
- *target_precompile_headers()*
- *target_sources()*

target_compile_features

Added in version 3.1.

Add expected compiler features to a target.

```
target_compile_features(<target> <PRIVATE|PUBLIC|INTERFACE> <feature> [...])
```

Specifies compiler features required when compiling a given target. If the feature is not listed in the *CMAKE_C_COMPILE_FEATURES*, *CMAKE_CUDA_COMPILE_FEATURES*, or

CMAKE_CXX_COMPILE_FEATURES variables, then an error will be reported by CMake. If the use of the feature requires an additional compiler flag, such as **−std=gnu++11**, the flag will be added automatically.

The **INTERFACE**, **PUBLIC** and **PRIVATE** keywords are required to specify the scope of the features. **PRIVATE** and **PUBLIC** items will populate the *COMPILE_FEATURES* property of **<target>**. **PUBLIC** and **INTERFACE** items will populate the *INTERFACE_COMPILE_FEATURES* property of **<target>**. Repeated calls for the same **<target>** append items.

Added in version 3.11: Allow setting **INTERFACE** items on *IMPORTED targets*.

The named **<target>** must have been created by a command such as *add_executable()* or *add_library()* and must not be an *ALIAS target*.

for more on defining buildsystem properties.

Arguments to **target_compile_features** may use generator expressions with the syntax **$<...>**. See the *cmake−generator−expressions(7)* manual for available expressions. See the*cmak e−compile−features(7)* manual for information on compile features and a list of supported compilers.

See Also

- *target_compile_definitions()*
- *target_compile_options()*
- *target_include_directories()*
- *target_link_libraries()*
- *target_link_directories()*
- *target_link_options()*
- *target_precompile_headers()*
- *target_sources()*

target_compile_options

Add compile options to a target.

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

Adds options to the *COMPILE_OPTIONS* or *INTERFACE_COMPILE_OPTIONS* target properties. These options are used when compiling the given **<target>**, which must have been created by a command such as *add_executable()* or *add_library()* and must not be an *ALIAS target*.

NOTE:

These options are not used when linking the target. See the*tar get_link_options()* command for that.

Arguments

If **BEFORE** is specified, the content will be prepended to the property instead of being appended. See policy *CMP0101* which affects whether **BEFORE** will be ignored in certain cases.

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

Added in version 3.11: Allow setting **INTERFACE** items on *IMPORTED targets*.

Arguments to **target_compile_options** may use generator expressions with the syntax **$<...>**. See the *cmake−generator−expressions(7)* manual for available expressions. See the*cmak e−buildsystem(7)* manual for more on defining buildsystem properties.

Option De−duplication

The final set of options used for a target is constructed by accumulating options from the current target and the usage requirements of its dependencies. The set of options is de−duplicated to avoid repetition.

Added in version 3.12: While beneficial for individual options, the de−duplication step can break up option groups. For example, **−option A −option B** becomes **−option A B**. One may specify a group of options using shell−like quoting along with a **SHELL:** prefix. The **SHELL:** prefix is dropped, and the rest of the option string is parsed using the *separate_arguments()* **UNIX_COMMAND** mode. For example, **"SHELL:−option A" "SHELL:−option B"** becomes **−option A −option B**.

See Also

- This command can be used to add any options. However, for adding preprocessor definitions and include directories it is recommended to use the more specific commands *target_compile_definitions()* and *target_include_directories()*.
- For directory−wide settings, there is the command *add_compile_options()*.
- For file−specific settings, there is the source file property *COMPILE_OPTIONS*.
- This command adds compile options for all languages in a target. Use the*COMPILE_LANGU AGE* generator expression to specify per−language compile options.
- *target_compile_features()*
- *target_link_libraries()*
- *target_link_directories()*
- *target_link_options()*
- *target_precompile_headers()*
- *target_sources()*
- *CMAKE_<LANG>_FLAGS* and *CMAKE_<LANG>_FLAGS_<CONFIG>* add language−wide flags passed to all invocations of the compiler. This includes invocations that drive compiling and those that drive linking.

target_include_directories

Add include directories to a target.

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

Specifies include directories to use when compiling a given target. The named**<tar get>** must have been created by a command such as *add_executable()* or *add_library()* and must not be an *ALIAS target*.

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

The **INTERFACE**, **PUBLIC** and **PRIVATE** keywords are required to specify the *scope* of the following arguments. **PRIVATE** and **PUBLIC** items will populate the *INCLUDE_DIRECTORIES* property of **<target>**. **PUBLIC** and **INTERFACE** items will populate the *INTERFACE_INCLUDE_DIRECTORIES*

property of **<target>**. The following arguments specify include directories.

Added in version 3.11: Allow setting **INTERFACE** items on *IMPORTED targets*.

Repeated calls for the same **<target>** append items in the order called.

If **SYSTEM** is specified, the compiler will be told the directories are meant as system include directories on some platforms. This may have effects such as suppressing warnings or skipping the contained headers in dependency calculations (see compiler documentation). Additionally, system include directories are searched after normal include directories regardless of the order specified.

If **SYSTEM** is used together with **PUBLIC** or **INTERFACE**, the *INTERFACE_SYSTEM_INCLUDE_DIRECTORIES* target property will be populated with the specified directories.

Arguments to **target_include_directories** may use generator expressions with the syntax **$<...>**. See the *cmake−generator−expressions(7)* manual for available expressions. See the*cmak e−buildsystem(7)* manual for more on defining buildsystem properties.

Specified include directories may be absolute paths or relative paths. A relative path will be interpreted as relative to the current source directory (i.e. *CMAKE_CURRENT_SOURCE_DIR*) and converted to an absolute path before storing it in the associated target property. If the path starts with a generator expression, it will always be assumed to be an absolute path (with one exception noted below) and will be used unmodified.

Include directories usage requirements commonly differ between the build−tree and the install−tree. The *BUILD_INTERFACE* and *INSTALL_INTERFACE* generator expressions can be used to describe separate usage requirements based on the usage location. Relative paths are allowed within the *INSTALL_INTERFACE* expression and are interpreted as relative to the installation prefix. Relative paths should not be used in *BUILD_INTERFACE* expressions because they will not be converted to absolute. For example:

```
target_include_directories(mylib PUBLIC
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include/mylib>
  $<INSTALL_INTERFACE:include/mylib>  # <prefix>/include/mylib
)
```

Creating Relocatable Packages

Note that it is not advisable to populate the *INSTALL_INTERFACE* of the *INTERFACE_INCLUDE_DIRECTORIES* of a target with absolute paths to the include directories of dependencies. That would hard−code into installed packages the include directory paths for dependencies **as found on the machine the package was made on**.

The *INSTALL_INTERFACE* of the *INTERFACE_INCLUDE_DIRECTORIES* is only suitable for specifying the required include directories for headers provided with the target itself, not those provided by the transitive dependencies listed in its *INTERFACE_LINK_LIBRARIES* target property. Those dependencies should themselves be targets that specify their own header locations in *INTERFACE_INCLUDE_DIRECTORIES*.

See the *Creating Relocatable Packages* section of the *cmake−packages(7)* manual for discussion of additional care that must be taken when specifying usage requirements while creating packages for redistribution.

See Also

- *include_directories()*
- *target_compile_definitions()*
- *target_compile_features()*
- *target_compile_options()*
- *target_link_libraries()*
- *target_link_directories()*
- *target_link_options()*
- *target_precompile_headers()*
- *target_sources()*

target_link_directories
Added in version 3.13.

Add link directories to a target.

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

Specifies the paths in which the linker should search for libraries when linking a given target. Each item can be an absolute or relative path, with the latter being interpreted as relative to the current source directory. These items will be added to the link command.

The named **<target>** must have been created by a command such as *add_executable()* or *add_library()* and must not be an *ALIAS target*.

The **INTERFACE**, **PUBLIC** and **PRIVATE** keywords are required to specify the *scope* of the items that follow them. **PRIVATE** and **PUBLIC** items will populate the *LINK_DIRECTORIES* property of **<target>**. **PUBLIC** and **INTERFACE** items will populate the *INTERFACE_LINK_DIRECTORIES* property of **<target>** (*IMPORTED targets* only support **INTERFACE** items). Each item specifies a link directory and will be converted to an absolute path if necessary before adding it to the relevant property. Repeated calls for the same **<target>** append items in the order called.

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

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

NOTE:
This command is rarely necessary and should be avoided where there are other choices. Prefer to pass full absolute paths to libraries where possible, since this ensures the correct library will always be linked. The*find_library()* command provides the full path, which can generally be used directly in calls to *target_link_libraries()*. Situations where a library search path may be needed include:

- Project generators like Xcode where the user can switch target architecture at build time, but a full path to a library cannot be used because it only provides one architecture (i.e. it is not a universal binary).
- Libraries may themselves have other private library dependencies that expect to be found via **RPATH** mechanisms, but some linkers are not able to fully decode those paths (e.g. due to the presence of things like **$ORIGIN**).

See Also

- *link_directories()*
- *target_compile_definitions()*
- *target_compile_features()*
- *target_compile_options()*
- *target_include_directories()*
- *target_link_libraries()*
- *target_link_options()*
- *target_precompile_headers()*
- *target_sources()*

target_link_libraries

Specify libraries or flags to use when linking a given target and/or its dependents. *Usa ge requirements* from linked library targets will be propagated. Usage requirements of a target's dependencies affect compilation of its own sources.

Overview

This command has several signatures as detailed in subsections below. All of them have the general form

```
target_link_libraries(<target> ... <item>... ...)
```

The named **<target>** must have been created by a command such as *add_executable()* or *add_library()* and must not be an *ALIAS target*. If policy *CMP0079* is not set to **NEW** then the target must have been created in the current directory. Repeated calls for the same**<tar get>** append items in the order called.

Added in version 3.13: The **<target>** doesn't have to be defined in the same directory as the **target_link_libraries** call.

Each **<item>** may be:

- **A library target name**: The generated link line will have the full path to the linkable library file associated with the target. The buildsystem will have a dependency to re−link **<target>** if the library file changes.

  The named target must be created by *add_library()* within the project or as an *IMPORTED library*. If it is created within the project an ordering dependency will automatically be added in the build system to make sure the named library target is up−to−date before the **<target>** links.

  If an imported library has the *IMPORTED_NO_SONAME* target property set, CMake may ask the linker to search for the library instead of using the full path (e.g. **/usr/lib/libfoo.so** becomes **−lfoo**).

  The full path to the target's artifact will be quoted/escaped for the shell automatically.

- **A full path to a library file**: The generated link line will normally preserve the full path to the file. The buildsystem will have a dependency to re−link **<target>** if the library file changes.

  There are some cases where CMake may ask the linker to search for the library (e.g. **/usr/lib/libfoo.so** becomes **−lfoo**), such as when a shared library is detected to have no **SONAME** field. In CMake versions prior to 4.0, see policy *CMP0060* for discussion of another case.

  If the library file is in a macOS framework, the **Headers** directory of the framework will also be processed as a *usage requirement*. This has the same effect as passing the framework directory as an include directory.

Added in version 3.28: The library file may point to a **.xcframework** folder on Apple platforms. If it does, the target will get the selected library's **Headers** directory as a usage requirement.

Added in version 3.8: On *Visual Studio Generators* for VS 2010 and above, library files ending in **.targets** will be treated as MSBuild targets files and imported into generated project files. This is not supported by other generators.

The full path to the library file will be quoted/escaped for the shell automatically.

- **A plain library name**: The generated link line will ask the linker to search for the library (e.g. **foo** becomes **−lfoo** or **foo.lib**).

  The library name/flag is treated as a command−line string fragment and will be used with no extra quoting or escaping.

- **A link flag**: Item names starting with **−**, but not **−l** or **−framework**, are treated as linker flags. Note that such flags will be treated like any other library link item for purposes of transitive dependencies, so they are generally safe to specify only as private link items that will not propagate to dependents.

  Link flags specified here are inserted into the link command in the same place as the link libraries. This might not be correct, depending on the linker. Use the *LINK_OPTIONS* target property or *target_link_options()* command to add link flags explicitly. The flags will then be placed at the toolchain−defined flag position in the link command.

  Added in version 3.13: *LINK_OPTIONS* target property and *target_link_options()* command. For earlier versions of CMake, use *LINK_FLAGS* property instead.

  The link flag is treated as a command−line string fragment and will be used with no extra quoting or escaping.

- **A generator expression**: A **$<...>** *generator expression* may evaluate to any of the above items or to a *semicolon−separated list* of them. If the **...** contains any **;** characters, e.g. after evaluation of a **${list}** variable, be sure to use an explicitly quoted argument **"$<...>"** so that this command receives it as a single **<item>**.

  Additionally, a generator expression may be used as a fragment of any of the above items, e.g. **foo$<1:_d>**.

- A **debug**, **optimized**, or **general** keyword immediately followed by another **<item>**. The item following such a keyword will be used only for the corresponding build configuration. The **debug** keyword corresponds to the **Debug** configuration (or to configurations named in the *DEBUG_CONFIGURATIONS* global property if it is set). The **optimized** keyword corresponds to all other configurations. The **general** keyword corresponds to all configurations, and is purely optional. Higher granularity may be achieved for per−configuration rules by creating and linking to *IMPORTED library targets*. These keywords are interpreted immediately by this command and therefore have no special meaning when produced by a generator expression.

Items containing **::**, such as **Foo::Bar**, are assumed to be *IMPORTED* or *ALIAS* library target names and will cause an error if no such target exists. See policy *CMP0028*.

See the *CMAKE_LINK_LIBRARIES_STRATEGY* variable and corresponding *LINK_LIBRARIES_STRATEGY* target property for details on how CMake orders direct link dependencies on linker command lines.

See the *cmake−buildsystem(7)* manual for more on defining buildsystem properties.

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.

Libraries for a Target and/or its Dependents

```
target_link_libraries(<target>
                      <PRIVATE|PUBLIC|INTERFACE> <item>...
                     [<PRIVATE|PUBLIC|INTERFACE> <item>...]...)
```

The **PUBLIC**, **PRIVATE** and **INTERFACE** *scope* keywords can be used to specify both the link dependencies and the link interface in one command.

Libraries and targets following **PUBLIC** are linked to, and are made part of the link interface. Libraries and targets following **PRIVATE** are linked to, but are not made part of the link interface. Libraries following **INTERFACE** are appended to the link interface and are not used for linking **<target>**.

Libraries for both a Target and its Dependents

```
target_link_libraries(<target> <item>...)
```

Library dependencies are transitive by default with this signature. When this target is linked into another target then the libraries linked to this target will appear on the link line for the other target too. This transitive "link interface" is stored in the*INTERFACE_LINK_LIBRARIES* target property and may be overridden by setting the property directly.

In CMake versions prior to 4.0, if *CMP0022* is not set to **NEW**, transitive linking is built in but may be overridden by the *LINK_INTERFACE_LIBRARIES* property. Calls to other signatures of this command may set the property making any libraries linked exclusively by this signature private.

Libraries for a Target and/or its Dependents (Legacy)

This signature is for compatibility only. Prefer the**PUBLIC** or **PRIVATE** keywords instead.

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

The **LINK_PUBLIC** and **LINK_PRIVATE** modes can be used to specify both the link dependencies and the link interface in one command.

Libraries and targets following **LINK_PUBLIC** are linked to, and are made part of the

INTERFACE_LINK_LIBRARIES.

In CMake versions prior to 4.0, if policy *CMP0022* is not **NEW**, they are also made part of the *LINK_INTERFACE_LIBRARIES*. Libraries and targets following **LINK_PRIVATE** are linked to, but are not made part of the *INTERFACE_LINK_LIBRARIES* (or *LINK_INTERFACE_LIBRARIES*).

Libraries for Dependents Only (Legacy)

This signature is for compatibility only. Prefer the**INTERF ACE** mode instead.

```
target_link_libraries(<target> LINK_INTERFACE_LIBRARIES <item>...)
```

The **LINK_INTERFACE_LIBRARIES** mode appends the libraries to the *INTERFACE_LINK_LIBRARIES* target property instead of using them for linking.

In CMake versions prior to 4.0, if policy *CMP0022* is not **NEW**, then this mode also appends libraries to the *LINK_INTERFACE_LIBRARIES* and its per−configuration equivalent.

Linking Object Libraries

Added in version 3.12.

Object Libraries may be used as the **<target>** (first) argument of **target_link_libraries** to specify dependencies of their sources on other libraries. For example, the code

```
add_library(A SHARED a.c)
target_compile_definitions(A PUBLIC A)

add_library(obj OBJECT obj.c)
target_compile_definitions(obj PUBLIC OBJ)
target_link_libraries(obj PUBLIC A)
```

compiles **obj.c** with **−DA −DOBJ** and establishes usage requirements for **obj** that propagate to its dependents.

Normal libraries and executables may link to *Object Libraries* to get their objects and usage requirements. Continuing the above example, the code

```
add_library(B SHARED b.c)
target_link_libraries(B PUBLIC obj)
```

compiles **b.c** with **−DA −DOBJ**, creates shared library **B** with object files from **b.c** and **obj.c**, and links **B** to **A**. Furthermore, the code

```
add_executable(main main.c)
target_link_libraries(main B)
```

compiles **main.c** with **−DA −DOBJ** and links executable **main** to **B** and **A**. The object library's usage requirements are propagated transitively through **B**, but its object files are not.

Object Libraries may "link" to other object libraries to get usage requirements, but since they do not have a link step nothing is done with their object files. Continuing from the above example, the code:

```
add_library(obj2 OBJECT obj2.c)
target_link_libraries(obj2 PUBLIC obj)

add_executable(main2 main2.c)
```

```
target_link_libraries(main2 obj2)
```

compiles **obj2.c** with **−DA −DOBJ**, creates executable **main2** with object files from **main2.c** and **obj2.c**, and links **main2** to **A**.

In other words, when *Object Libraries* appear in a target's *INTERFACE_LINK_LIBRARIES* property they will be treated as *Interface Libraries*, but when they appear in a target's *LINK_LIBRARIES* property their object files will be included in the link too.

Linking Object Libraries via $<TARGET_OBJECTS>

Added in version 3.21.

The object files associated with an object library may be referenced by the *$<TARGET_OBJECTS>* generator expression. Such object files are placed on the link line*befor e* all libraries, regardless of their relative order. Additionally, an ordering dependency will be added to the build system to make sure the object library is up−to−date before the dependent target links. For example, the code

```
add_library(obj3 OBJECT obj3.c)
target_compile_definitions(obj3 PUBLIC OBJ3)

add_executable(main3 main3.c)
target_link_libraries(main3 PRIVATE a3 $<TARGET_OBJECTS:obj3> b3)
```

links executable **main3** with object files from **main3.c** and **obj3.c** followed by the **a3** and **b3** libraries. **main3.c** is *not* compiled with usage requirements from **obj3**, such as **−DOBJ3**.

This approach can be used to achieve transitive inclusion of object files in link lines as usage requirements. Continuing the above example, the code

```
add_library(iface_obj3 INTERFACE)
target_link_libraries(iface_obj3 INTERFACE obj3 $<TARGET_OBJECTS:obj3>)
```

creates an interface library **iface_obj3** that forwards the **obj3** usage requirements and adds the **obj3** object files to dependents' link lines. The code

```
add_executable(use_obj3 use_obj3.c)
target_link_libraries(use_obj3 PRIVATE iface_obj3)
```

compiles **use_obj3.c** with **−DOBJ3** and links executable **use_obj3** with object files from **use_obj3.c** and **obj3.c**.

This also works transitively through a static library. Since a static library does not link, it does not consume the object files from object libraries referenced this way. Instead, the object files become transitive link dependencies of the static library. Continuing the above example, the code

```
add_library(static3 STATIC static3.c)
target_link_libraries(static3 PRIVATE iface_obj3)

add_executable(use_static3 use_static3.c)
target_link_libraries(use_static3 PRIVATE static3)
```

compiles **static3.c** with **−DOBJ3** and creates **libstatic3.a** using only its own object file. **use_static3.c** is compiled *without* **−DOBJ3** because the usage requirement is not transitive through the private dependency of **static3**. However, the link dependencies of **static3** are propagated, including the **iface_obj3** reference to

$<TARGET_OBJECTS:obj3>. The**use_static3** e xecutable is created with object files from **use_static3.c** and **obj3.c**, and linked to library **libstatic3.a**.

When using this approach, it is the project's responsibility to avoid linking multiple dependent binaries to **iface_obj3**, because they will all get the **obj3** object files on their link lines.

NOTE:

Referencing *$<TARGET_OBJECTS>* in **target_link_libraries** calls worked in versions of CMake prior to 3.21 for some cases, but was not fully supported:

- It did not place the object files before libraries on link lines.
- It did not add an ordering dependency on the object library.
- It did not work in Xcode with multiple architectures.

Cyclic Dependencies of Static Libraries

The library dependency graph is normally acyclic (a DAG), but in the case of mutually−dependent **STATIC** libraries CMake allows the graph to contain cycles (strongly connected components). When another target links to one of the libraries, CMake repeats the entire connected component. For example, the code

```
add_library(A STATIC a.c)
add_library(B STATIC b.c)
target_link_libraries(A B)
target_link_libraries(B A)
add_executable(main main.c)
target_link_libraries(main A)
```

links **main** to **A B A B**. While one repetition is usually sufficient, pathological object file and symbol arrangements can require more. One may handle such cases by using the *LINK_INTERFACE_MULTIPLICITY* target property or by manually repeating the component in the last **target_link_libraries** call. However, if two archives are really so interdependent they should probably be combined into a single archive, perhaps by using *Object Libraries*.

Creating Relocatable Packages

Note that it is not advisable to populate the *INTERFACE_LINK_LIBRARIES* of a target with absolute paths to dependencies. That would hard−code into installed packages the library file paths for dependencies **as found on the machine the package was made on**.

See the *Creating Relocatable Packages* section of the *cmake−packages(7)* manual for discussion of additional care that must be taken when specifying usage requirements while creating packages for redistribution.

See Also

- *target_compile_definitions()*
- *target_compile_features()*
- *target_compile_options()*
- *target_include_directories()*
- *target_link_directories()*
- *target_link_options()*
- *target_precompile_headers()*
- *target_sources()*

target_link_options

Added in version 3.13.

Add options to the link step for an executable, shared library or module library target.

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

The named **<target>** must have been created by a command such as *add_executable()* or *add_library()* and must not be an *ALIAS target*.

This command can be used to add any link options, but alternative commands exist to add libraries (-*target_link_libraries()* or *link_libraries()*). See documentation of the*dir ectory* and *target* **LINK_OPTIONS** properties.

NOTE:
This command cannot be used to add options for static library targets, since they do not use a linker. To add archiver or MSVC librarian flags, see the *STATIC_LIBRARY_OPTIONS* target property.

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

The **INTERFACE**, **PUBLIC** and **PRIVATE** keywords are required to specify the *scope* of the following arguments. **PRIVATE** and **PUBLIC** items will populate the *LINK_OPTIONS* property of **<target>**. **PUBLIC** and **INTERFACE** items will populate the *INTERFACE_LINK_OPTIONS* property of **<target>**. The following arguments specify link options. Repeated calls for the same **<target>** append items in the order called.

NOTE:
IMPORTED targets only support **INTERFACE** items.

Arguments to **target_link_options** may use generator expressions with the syntax **$<...>**. See the *cmake−generator−expressions(7)* manual for available expressions. See the*cmak e−buildsystem(7)* manual for more on defining buildsystem properties.

Host And Device Specific Link Options
Added in version 3.18: When a device link step is involved, which is controlled by *CUDA_SEPARABLE_COMPILATION* and *CUDA_RESOLVE_DEVICE_SYMBOLS* properties and policy *CMP0105*, the raw options will be delivered to the host and device link steps (wrapped in **−Xcompiler** or equivalent for device link). Options wrapped with *$<DEVICE_LINK:...>* generator expression will be used only for the device link step. Options wrapped with *$<HOST_LINK:...>* generator expression will be used only for the host link step.

Option De−duplication
The final set of options used for a target is constructed by accumulating options from the current target and the usage requirements of its dependencies. The set of options is de−duplicated to avoid repetition.

Added in version 3.12: While beneficial for individual options, the de−duplication step can break up option groups. For example, **−option A −option B** becomes **−option A B**. One may specify a group of options using shell−like quoting along with a **SHELL:** prefix. The **SHELL:** prefix is dropped, and the rest of the option string is parsed using the *separate_arguments()* **UNIX_COMMAND** mode. For example, **"SHELL:−option A" "SHELL:−option B"** becomes **−option A −option B**.

Handling Compiler Driver Differences
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.

See Also

- *target_compile_definitions()*
- *target_compile_features()*
- *target_compile_options()*
- *target_include_directories()*
- *target_link_libraries()*
- *target_link_directories()*
- *target_precompile_headers()*
- *target_sources()*
- *CMAKE_<LANG>_FLAGS* and *CMAKE_<LANG>_FLAGS_<CONFIG>* add language−wide flags passed to all invocations of the compiler. This includes invocations that drive compiling and those that drive linking.

target_precompile_headers

Added in version 3.16.

Add a list of header files to precompile.

Precompiling header files can speed up compilation by creating a partially processed version of some header files, and then using that version during compilations rather than repeatedly parsing the original headers.

Main Form

```
target_precompile_headers(<target>
  <INTERFACE|PUBLIC|PRIVATE> [header1...]
  [<INTERFACE|PUBLIC|PRIVATE> [header2...] ...])
```

The command adds header files to the*PRECOMPILE_HEADERS* and/or *INTERFACE_PRECOMPILE_HEADERS* target properties of **<target>**. The named**<tar get>** must have been created by a command such as *add_executable()* or *add_library()* and must not be an *ALIAS target*.

The **INTERFACE**, **PUBLIC** and **PRIVATE** keywords are required to specify the *scope* of the following arguments. **PRIVATE** and **PUBLIC** items will populate the *PRECOMPILE_HEADERS* property of **<target>**. **PUBLIC** and **INTERFACE** items will populate the *INTERFACE_PRECOMPILE_HEADERS* property of **<target>** (*IMPORTED targets* only support **INTERFACE** items). Repeated calls for the same **<target>** will append items in the order called.

Projects should generally avoid using **PUBLIC** or **INTERFACE** for targets that will be *exported*, or they should at least use the *$<BUILD_INTERFACE:...>* generator expression to prevent precompile headers from appearing in an installed exported target. Consumers of a target should typically be in control of what precompile headers they use, not have precompile headers forced on them by the targets being consumed (since precompile headers are not typically usage requirements). A notable exception to this is where an *interface library* is created to define a commonly used set of precompile headers in one place and then other targets link to that interface library privately. In this case, the interface library exists specifically to propagate the precompile headers to its consumers and the consumer is effectively still in control, since it decides whether to link to the interface library or not.

The list of header files is used to generate a header file named **cmake_pch.h|xx** which is used to generate the precompiled header file (**.pch**, **.gch**, **.pchi**) artifact. The**cmak e_pch.h|xx** header file will be force included (**−include** for GCC, **/FI** for MSVC) to all source files, so sources do not need to have **#include "pch.h"**.

Header file names specified with angle brackets (e.g. **<unordered_map>**) or explicit double quotes (escaped for the *cmake−language(7)*, e.g. **[["other_header.h"]]**) will be treated as is, and include directories must be available for the compiler to find them. Other header file names (e.g. **project_header.h**) are interpreted as being relative to the current source directory (e.g. *CMAKE_CURRENT_SOURCE_DIR*) and will be included by absolute path. For example:

```
target_precompile_headers(myTarget
  PUBLIC
    project_header.h
  PRIVATE
    [["other_header.h"]]
    <unordered_map>
)
```

for more on defining buildsystem properties.

Arguments to **target_precompile_headers** may use generator expressions with the syntax **$<...>**. See the *cmake−generator−expressions(7)* manual for available expressions. The*$<COMPILE_LANGU AGE:...>* generator expression is particularly useful for specifying a language−specific header to precompile for only one language (e.g. **CXX** and not **C**). In this case, header file names that are not explicitly in double quotes or angle brackets must be specified by absolute path. Also, when specifying angle brackets inside a generator expression, be sure to encode the closing > as *$<ANGLE−R>*. For example:

```
target_precompile_headers(mylib PRIVATE
  "$<$<COMPILE_LANGUAGE:CXX>:${CMAKE_CURRENT_SOURCE_DIR}/cxx_only.h>"
  "$<$<COMPILE_LANGUAGE:C>:<stddef.h$<ANGLE-R>>"
  "$<$<COMPILE_LANGUAGE:CXX>:<cstddef$<ANGLE-R>>"
)
```

Reusing Precompile Headers

The command also supports a second signature which can be used to specify that one target reuses a precompiled header file artifact from another target instead of generating its own:

```
target_precompile_headers(<target> REUSE_FROM <other_target>)
```

This form sets the *PRECOMPILE_HEADERS_REUSE_FROM* property to **<other_target>** and adds a dependency such that **<target>** will depend on **<other_target>**. CMake will halt with an error if the *PRECOMPILE_HEADERS* property of **<target>** is already set when the **REUSE_FROM** form is used.

NOTE:

The **REUSE_FROM** form requires the same set of compiler options, compiler flags and compiler

definitions for both **<target>** and **<other_target>**. Some compilers (e.g. GCC) may issue a warning if the precompiled header file cannot be used (**-Winvalid-pch**).

See Also

- To disable precompile headers for specific targets, see the *DISABLE_PRECOMPILE_HEADERS* target property.
- To prevent precompile headers from being used when compiling a specific source file, see the *SKIP_PRECOMPILE_HEADERS* source file property.
- *target_compile_definitions()*
- *target_compile_features()*
- *target_compile_options()*
- *target_include_directories()*
- *target_link_libraries()*
- *target_link_directories()*
- *target_link_options()*
- *target_sources()*

target_sources

Added in version 3.1.

Add sources to a target.

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

Specifies sources to use when building a target and/or its dependents. The named **<target>** must have been created by a command such as *add_executable()* or *add_library()* or *add_custom_target()* and must not be an *ALIAS target*. The**<items>** may use *g enerator expressions*.

Added in version 3.20: **<target>** can be a custom target.

The **INTERFACE**, **PUBLIC** and **PRIVATE** keywords are required to specify the *scope* of the source file paths (**<items>**) that follow them. **PRIVATE** and **PUBLIC** items will populate the *SOURCES* property of **<target>**, which are used when building the target itself. **PUBLIC** and **INTERFACE** items will populate the *INTERFACE_SOURCES* property of **<target>**, which are used when building dependents. A target created by *add_custom_target()* can only have **PRIVATE** scope.

Repeated calls for the same **<target>** append items in the order called.

Added in version 3.3: Allow exporting targets with *INTERFACE_SOURCES*.

Added in version 3.11: Allow setting **INTERFACE** items on *IMPORTED targets*.

Changed in version 3.13: Relative source file paths are interpreted as being relative to the current source directory (i.e. *CMAKE_CURRENT_SOURCE_DIR*). See policy *CMP0076*.

A path that begins with a generator expression is left unmodified. When a target's *SOURCE_DIR* property differs from *CMAKE_CURRENT_SOURCE_DIR*, use absolute paths in generator expressions to ensure the sources are correctly assigned to the target.

```
# WRONG: starts with generator expression, but relative path used
target_sources(MyTarget PRIVATE "$<$<CONFIG:Debug>:dbgsrc.cpp>")

# CORRECT: absolute path used inside the generator expression
target_sources(MyTarget PRIVATE "$<$<CONFIG:Debug>:${CMAKE_CURRENT_SOURCE_DIR}/db
```

See the *cmake−buildsystem(7)* manual for more on defining buildsystem properties.

File Sets

Added in version 3.23.

```
target_sources(<target>
  [<INTERFACE|PUBLIC|PRIVATE>
   [FILE_SET <set> [TYPE <type>] [BASE_DIRS <dirs>...] [FILES <files>...]]...
  ]...)
```

Adds a file set to a target, or adds files to an existing file set. Targets have zero or more named file sets. Each file set has a name, a type, a scope of **INTERFACE**, **PUBLIC**, or **PRIVATE**, one or more base directories, and files within those directories. The acceptable types include:

HEADERS
Sources intended to be used via a language's **#include** mechanism.

CXX_MODULES
Added in version 3.28.

Sources which contain C++ interface module or partition units (i.e., those using the **export** keyword). This file set type may not have an **INTERFACE** scope except on **IMPORTED** targets.

The optional default file sets are named after their type. The target may not be a custom target or *FRAMEWORK* target.

Files in a **PRIVATE** or **PUBLIC** file set are marked as source files for the purposes of IDE integration. Additionally, files in **HEADERS** file sets have their *HEADER_FILE_ONLY* property set to **TRUE**. Files in an **INTERFACE** or **PUBLIC** file set can be installed with the *install(TARGETS)* command, and exported with the *install(EXPORT)* and *export()* commands.

Each **target_sources(FILE_SET)** entry starts with **INTERFACE**, **PUBLIC**, or **PRIVATE** and accepts the following arguments:

FILE_SET <set>
The name of the file set to create or add to. It must contain only letters, numbers and underscores. Names starting with a capital letter are reserved for built−in file sets predefined by CMake. The only predefined set names are those matching the acceptable types. All other set names must not start with a capital letter or underscore.

TYPE <type>
Every file set is associated with a particular type of file. Only types specified above may be used and it is an error to specify anything else. As a special case, if the name of the file set is one of the types, the type does not need to be specified and the **TYPE <type>** arguments can be omitted. For all other file

set names, **TYPE** is required.

BASE_DIRS <dirs>...
An optional list of base directories of the file set. Any relative path is treated as relative to the current source directory (i.e. *CMAKE_CURRENT_SOURCE_DIR*). If no **BASE_DIRS** are specified when the file set is first created, the value of *CMAKE_CURRENT_SOURCE_DIR* is added. This argument supports *generator expressions*.

No two base directories for a file set may be sub−directories of each other. This requirement must be met across all base directories added to a file set, not just those within a single call to **target_sources()**.

FILES <files>...
An optional list of files to add to the file set. Each file must be in one of the base directories, or a subdirectory of one of the base directories. This argument supports *generator expressions*.

If relative paths are specified, they are considered relative to *CMAKE_CURRENT_SOURCE_DIR* at the time **target_sources()** is called. An exception to this is a path starting with **$<**. Such paths are treated as relative to the target's source directory after evaluation of generator expressions.

The following target properties are set by **target_sources(FILE_SET)**, but they should not generally be manipulated directly:

For file sets of type **HEADERS**:

- *HEADER_SETS*
- *INTERFACE_HEADER_SETS*
- *HEADER_SET*
- *HEADER_SET_<NAME>*
- *HEADER_DIRS*
- *HEADER_DIRS_<NAME>*

For file sets of type **CXX_MODULES**:

- *CXX_MODULE_SETS*
- *INTERFACE_CXX_MODULE_SETS*
- *CXX_MODULE_SET*
- *CXX_MODULE_SET_<NAME>*
- *CXX_MODULE_DIRS*
- *CXX_MODULE_DIRS_<NAME>*

Target properties related to include directories are also modified by **target_sources(FILE_SET)** as follows:

INCLUDE_DIRECTORIES
If the **TYPE** is **HEADERS**, and the scope of the file set is **PRIVATE** or **PUBLIC**, all of the **BASE_DIRS** of the file set are wrapped in *$<BUILD_INTERFACE>* and appended to this property.

INTERFACE_INCLUDE_DIRECTORIES
If the **TYPE** is **HEADERS**, and the scope of the file set is **INTERFACE** or **PUBLIC**, all of the **BASE_DIRS** of the file set are wrapped in *$<BUILD_INTERFACE>* and appended to this property.

See Also

- *add_executable()*
- *add_library()*
- *target_compile_definitions()*
- *target_compile_features()*
- *target_compile_options()*
- *target_include_directories()*
- *target_link_libraries()*
- *target_link_directories()*
- *target_link_options()*
- *target_precompile_headers()*

try_compile

Try building some code.

Try Compiling Whole Projects

```
try_compile(<compileResultVar> PROJECT <projectName>
            SOURCE_DIR <srcdir>
            [BINARY_DIR <bindir>]
            [TARGET <targetName>]
            [LOG_DESCRIPTION <text>]
            [NO_CACHE]
            [NO_LOG]
            [CMAKE_FLAGS <flags>...]
            [OUTPUT_VARIABLE <var>])
```

Added in version 3.25.

Try building a project. Build success returns **TRUE** and build failure returns **FALSE** in **<compileResultVar>**.

In this form, **<srcdir>** should contain a complete CMake project with a **CMakeLists.txt** file and all sources. The**<bindir>** and **<sr cdir>** will not be deleted after this command is run. Specify **<targetName>** to build a specific target instead of the **all** or **ALL_BUILD** target. See below for the meaning of other options.

Changed in version 3.24: CMake variables describing platform settings, and those listed by the *CMAKE_TRY_COMPILE_PLATFORM_VARIABLES* variable, are propagated into the project's build configuration. See policy *CMP0137*. Previously this was only done by the *source file* signature.

Added in version 3.26: This command records a *configure−log try_compile event* if the **NO_LOG** option is not specified.

Added in version 3.30: If the *PROPAGATE_TOP_LEVEL_INCLUDES_TO_TRY_COMPILE* global property is set to true, *CMAKE_PROJECT_TOP_LEVEL_INCLUDES* is propagated into the project's build configuration.

This command supports an alternate signature for CMake older than 3.25. The signature above is

recommended for clarity.

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

Try Compiling Source Files

```
try_compile(<compileResultVar>
            [SOURCES_TYPE <type>]
            <SOURCES <srcfile...>                  |
             SOURCE_FROM_CONTENT <name> <content> |
             SOURCE_FROM_VAR <name> <var>         |
             SOURCE_FROM_FILE <name> <path>       >...
            [LOG_DESCRIPTION <text>]
            [NO_CACHE]
            [NO_LOG]
            [CMAKE_FLAGS <flags>...]
            [COMPILE_DEFINITIONS <defs>...]
            [LINK_OPTIONS <options>...]
            [LINK_LIBRARIES <libs>...]
            [LINKER_LANGUAGE <lang>]
            [OUTPUT_VARIABLE <var>]
            [COPY_FILE <fileName> [COPY_FILE_ERROR <var>]]
            [<LANG>_STANDARD <std>]
            [<LANG>_STANDARD_REQUIRED <bool>]
            [<LANG>_EXTENSIONS <bool>]
            )
```

Added in version 3.25.

Try building an executable or static library from one or more source files. The binary type is determined by variable *CMAKE_TRY_COMPILE_TARGET_TYPE*. Build success returns boolean**true** and b uild failure returns boolean **false** in <**compileResultVar**> (cached unless **NO_CACHE** is specified).

In this form, one or more source files must be provided. Additionally, one of **SOURCES** and/or **SOURCE_FROM_*** must precede other keywords.

If *CMAKE_TRY_COMPILE_TARGET_TYPE* is unset or is set to **EXECUTABLE**, the sources must include a definition for **main** and CMake will create a **CMakeLists.txt** file to build the source(s) as an executable. If*CMAKE_TR Y_COMPILE_TARGET_TYPE* is set to **STATIC_LIBRARY**, a static library will be built instead and no definition for **main** is required. For an executable, the generated **CMakeLists.txt** file would contain something like the following:

```
add_definitions(<expanded COMPILE_DEFINITIONS from caller>)
include_directories(${INCLUDE_DIRECTORIES})
link_directories(${LINK_DIRECTORIES})
add_executable(cmTryCompileExec <srcfile>...)
target_link_options(cmTryCompileExec PRIVATE <LINK_OPTIONS from caller>)
target_link_libraries(cmTryCompileExec ${LINK_LIBRARIES})
```

CMake automatically generates, for each **try_compile** operation, a unique directory under **${CMAKE_BINARY_DIR}/CMakeFiles/CMakeScratch** with an unspecified name. These directories are cleaned automatically unless *--debug-trycompile* is passed to **cmake**. Such directories from previous runs are also

unconditionally cleaned at the beginning of any **cmake** execution.

This command supports an alternate signature for CMake older than 3.25. The signature above is recommended for clarity.

```
try_compile(<compileResultVar> <bindir> <srcfile|SOURCES srcfile...>
            [CMAKE_FLAGS <flags>...]
            [COMPILE_DEFINITIONS <defs>...]
            [LINK_OPTIONS <options>...]
            [LINK_LIBRARIES <libs>...]
            [OUTPUT_VARIABLE <var>]
            [COPY_FILE <fileName> [COPY_FILE_ERROR <var>]]
            [<LANG>_STANDARD <std>]
            [<LANG>_STANDARD_REQUIRED <bool>]
            [<LANG>_EXTENSIONS <bool>]
            )
```

In this version, **try_compile** will use **<bindir>/CMakeFiles/CMakeTmp** for its operation, and all such files will be cleaned automatically. For debugging, *−−debug−trycompile* can be passed to **cmake** to avoid this clean. However, multiple sequential **try_compile** operations, if given the same **<bindir>**, will reuse this single output directory, such that you can only debug one such **try_compile** call at a time. Use of the newer signature is recommended to simplify debugging of multiple **try_compile** operations.

Options

The options for the above signatures are:

CMAKE_FLAGS <flags>...

Specify flags of the form *−DVAR:TYPE=VALUE* to be passed to the *cmake(1)* command−line used to drive the test build. The above example shows how values for variables **COMPILE_DEFINITIONS**, **INCLUDE_DIRECTORIES**, **LINK_DIRECTORIES**, **LINK_LIBRARIES**, and **LINK_OPTIONS** are used. Compiler options can be passed in like **CMAKE_FLAGS −DCOMPILE_DEFINITIONS=−Werror**.

COMPILE_DEFINITIONS <defs>...

Specify **−Ddefinition** arguments to pass to *add_definitions()* in the generated test project.

COPY_FILE <fileName>

Copy the built executable or static library to the given **<fileName>**.

COPY_FILE_ERROR <var>

Use after **COPY_FILE** to capture into variable **<var>** any error message encountered while trying to copy the file.

LINK_LIBRARIES <libs>...

Specify libraries to be linked in the generated project. The list of libraries may refer to system libraries and to *Imported Targets* from the calling project.

If this option is specified, any **−DLINK_LIBRARIES=...** value given to the **CMAKE_FLAGS** option will be ignored.

Added in version 3.29: Alias targets to imported libraries are also supported.

LINK_OPTIONS <options>...

Added in version 3.14.

Specify link step options to pass to *target_link_options()* or to set the *STATIC_LIBRARY_OPTIONS* target property in the generated project, depending on the

CMAKE_TRY_COMPILE_TARGET_TYPE variable.

LINKER_LANGUAGE <lang>
Added in version 3.29.

Specify the *LINKER_LANGUAGE* target property of the generated project. When using multiple source files with different languages, set this to the language of the source file containing the program entry point, e.g., **main**.

LOG_DESCRIPTION <text>
Added in version 3.26.

Specify a non−empty text description of the purpose of the check. This is recorded in the *cmake−configure−log(7)* entry.

NO_CACHE
Added in version 3.25.

<compileResultVar> will be stored in a normal variable rather than a cache entry.

<compileResultVar> is normally cached so that a simple pattern can be used to avoid repeating the test on subsequent executions of CMake:

```
if(NOT DEFINED RESULTVAR)
  # ...(check-specific setup code)...
  try_compile(RESULTVAR ...)
  # ...(check-specific logging and cleanup code)...
endif()
```

If the guard variable and result variable are not the same (for example, if the test is part of a larger inspection), **NO_CACHE** may be useful to avoid leaking the intermediate result variable into the cache.

NO_LOG
Added in version 3.26.

Do not record a *cmake−configure−log(7)* entry for this call.

OUTPUT_VARIABLE <var>
Store the output from the build process in the given variable.

SOURCE_FROM_CONTENT <name> <content>
Added in version 3.25.

Write **<content>** to a file named **<name>** in the operation directory. This can be used to bypass the need to separately write a source file when the contents of the file are dynamically specified. The specified **<name>** is not allowed to contain path components.

SOURCE_FROM_CONTENT may be specified multiple times.

SOURCE_FROM_FILE <name> <path>
Added in version 3.25.

Copy **<path>** to a file named **<name>** in the operation directory. This can be used to consolidate files into the operation directory, which may be useful if a source which already exists (i.e. as a stand−alone file in a project's source repository) needs to refer to other file(s) created by **SOURCE_FROM_***. (Otherwise, **SOURCES** is usually more convenient.) The specified **<name>** is not allowed to contain path components.

SOURCE_FROM_VAR <name> <var>
Added in version 3.25.

Write the contents of **<var>** to a file named **<name>** in the operation directory. This is the same as **SOURCE_FROM_CONTENT**, but takes the contents from the specified CMake variable, rather than directly, which may be useful when passing arguments through a function which wraps **try_compile**. The specified **<name>** is not allowed to contain path components.

SOURCE_FROM_VAR may be specified multiple times.

SOURCES_TYPE <type>
Added in version 3.28.

Sources may be classified using the **SOURCES_TYPE** argument. Once specified, all subsequent sources specified will be treated as that type until another **SOURCES_TYPE** is given. Available types are:

NORMAL
Sources are not added to any **FILE_SET** in the generated project.

CXX_MODULE
Added in version 3.28.

Sources are added to a **FILE_SET** of type **CXX_MODULES** in the generated project.

The default type of sources is **NORMAL**.

<LANG>_STANDARD <std>
Added in version 3.8.

Specify the *C_STANDARD*, *CXX_STANDARD*, *OBJC_STANDARD*, *OBJCXX_STANDARD*, or *CUDA_STANDARD* target property of the generated project.

<LANG>_STANDARD_REQUIRED <bool>
Added in version 3.8.

Specify the *C_STANDARD_REQUIRED*, *CXX_STANDARD_REQUIRED*, *OBJC_STANDARD_REQUIRED*, *OBJCXX_STANDARD_REQUIRED*,or *CUDA_STANDARD_REQUIRED* target property of the generated project.

<LANG>_EXTENSIONS <bool>
Added in version 3.8.

Specify the *C_EXTENSIONS*, *CXX_EXTENSIONS*, *OBJC_EXTENSIONS*, *OBJCXX_EXTENSIONS*, or *CUDA_EXTENSIONS* target property of the generated project.

Other Behavior Settings

Added in version 3.4: If set, the following variables are passed in to the generated try_compile CMakeLists.txt to initialize compile target properties with default values:

- *CMAKE_CUDA_RUNTIME_LIBRARY*
- *CMAKE_ENABLE_EXPORTS*
- *CMAKE_EXE_LINKER_FLAGS*, unless using CMake versions prior to 4.0 without policy *CMP0056* set to **NEW**
- *CMAKE_LINK_SEARCH_START_STATIC*
- *CMAKE_LINK_SEARCH_END_STATIC*
- *CMAKE_MSVC_RUNTIME_LIBRARY*
- *CMAKE_POSITION_INDEPENDENT_CODE*
- *CMAKE_WATCOM_RUNTIME_LIBRARY*

Changed in version 3.14: If *CMP0083* is set to **NEW**, then in order to obtain correct behavior at link time, the **check_pie_supported()** command from the *CheckPIESupported* module must be called before using the **try_compile** command.

Some policies are set automatically in the generated test project as needed to honor the state of the calling project:

- *CMP0065* (in CMake versions prior to 4.0)
- *CMP0083*
- *CMP0091*
- *CMP0104*
- *CMP0123*
- *CMP0126*
- *CMP0128*
- *CMP0136*
- *CMP0141*
- *CMP0155*
- *CMP0157*
- *CMP0181*
- *CMP0184*

Added in version 4.0: The current setting of *CMP0181* policy is propagated through to the generated test project.

Set variable *CMAKE_TRY_COMPILE_CONFIGURATION* to choose a build configuration:

- For multi−config generators, this selects which configuration to build.
- For single−config generators, this sets *CMAKE_BUILD_TYPE* in the test project.

Added in version 3.6: Set the *CMAKE_TRY_COMPILE_TARGET_TYPE* variable to specify the type of target used for the source file signature.

Added in version 3.6: Set the *CMAKE_TRY_COMPILE_PLATFORM_VARIABLES* variable to specify variables that must be propagated into the test project. This variable is meant for use only in toolchain files and is only honored by the **try_compile()** command for the source files form, not when given a whole project.

Changed in version 3.8: If *CMP0067* is set to **NEW**, or any of the **<LANG>_STANDARD**, **<LANG>_STANDARD_REQUIRED**, or **<LANG>_EXTENSIONS** options are used, then the language standard variables are honored:

- *CMAKE_C_STANDARD*
- *CMAKE_C_STANDARD_REQUIRED*
- *CMAKE_C_EXTENSIONS*
- *CMAKE_CXX_STANDARD*
- *CMAKE_CXX_STANDARD_REQUIRED*
- *CMAKE_CXX_EXTENSIONS*
- *CMAKE_OBJC_STANDARD*
- *CMAKE_OBJC_STANDARD_REQUIRED*
- *CMAKE_OBJC_EXTENSIONS*
- *CMAKE_OBJCXX_STANDARD*
- *CMAKE_OBJCXX_STANDARD_REQUIRED*
- *CMAKE_OBJCXX_EXTENSIONS*
- *CMAKE_CUDA_STANDARD*
- *CMAKE_CUDA_STANDARD_REQUIRED*
- *CMAKE_CUDA_EXTENSIONS*

Their values are used to set the corresponding target properties in the generated project (unless overridden by an explicit option).

Changed in version 3.14: For the *Green Hills MULTI* generator, the GHS toolset and target system customization cache variables are also propagated into the test project.

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.

Added in version 3.25: If*CMP0141* is set to **NEW**, one can use *CMAKE_MSVC_DEBUG_INFORMATION_FORMAT* to specify the MSVC debug information format.

Added in version 3.30: If the *PROPAGATE_TOP_LEVEL_INCLUDES_TO_TRY_COMPILE* global property is set to true, *CMAKE_PROJECT_TOP_LEVEL_INCLUDES* is propagated into the test project's build configuration when using the *whole−project signature*.

Added in version 4.0: If *CMP0184* is set to **NEW**, one can use *CMAKE_MSVC_RUNTIME_CHECKS* to specify the enabled MSVC runtime checks.

See Also

- *try_run()*

try_run

Try compiling and then running some code.

Try Compiling and Running Source Files

```
try_run(<runResultVar> <compileResultVar>
        [SOURCES_TYPE <type>]
        <SOURCES <srcfile...>                  |
         SOURCE_FROM_CONTENT <name> <content> |
         SOURCE_FROM_VAR <name> <var>         |
         SOURCE_FROM_FILE <name> <path>       >...
        [LOG_DESCRIPTION <text>]
        [NO_CACHE]
        [NO_LOG]
        [CMAKE_FLAGS <flags>...]
        [COMPILE_DEFINITIONS <defs>...]
        [LINK_OPTIONS <options>...]
        [LINK_LIBRARIES <libs>...]
        [COMPILE_OUTPUT_VARIABLE <var>]
        [COPY_FILE <fileName> [COPY_FILE_ERROR <var>]]
        [<LANG>_STANDARD <std>]
        [<LANG>_STANDARD_REQUIRED <bool>]
        [<LANG>_EXTENSIONS <bool>]
        [RUN_OUTPUT_VARIABLE <var>]
        [RUN_OUTPUT_STDOUT_VARIABLE <var>]
        [RUN_OUTPUT_STDERR_VARIABLE <var>]
        [WORKING_DIRECTORY <var>]
        [ARGS <args>...]
        )
```

Added in version 3.25.

Try building an executable from one or more source files. Build success returns boolean **true** and build failure returns boolean **false** in **<compileResultVar>** (cached unless **NO_CACHE** is specified). If the build succeeds, this runs the executable and stores the exit code in **<runResultVar>** (cached unless **NO_CACHE** is specified). If the executable was built, but failed to run, then **<runResultVar>** will be set to **FAILED_TO_RUN**. See command*try_compile()* for documentation of options common to both commands, and for information on how the test project is constructed to build the source file.

One or more source files must be provided. Additionally, one of **SOURCES** and/or **SOURCE_FROM_*** must precede other keywords.

Added in version 3.26: This command records a *configure−log try_run event* if the **NO_LOG** option is not specified.

This command supports an alternate signature for CMake older than 3.25. The signature above is recommended for clarity.

```
try_run(<runResultVar> <compileResultVar>
        <bindir> <srcfile|SOURCES srcfile...>
        [CMAKE_FLAGS <flags>...]
        [COMPILE_DEFINITIONS <defs>...]
```

```
[LINK_OPTIONS <options>...]
[LINK_LIBRARIES <libs>...]
[LINKER_LANGUAGE <lang>]
[COMPILE_OUTPUT_VARIABLE <var>]
[COPY_FILE <fileName> [COPY_FILE_ERROR <var>]]
[<LANG>_STANDARD <std>]
[<LANG>_STANDARD_REQUIRED <bool>]
[<LANG>_EXTENSIONS <bool>]
[RUN_OUTPUT_VARIABLE <var>]
[OUTPUT_VARIABLE <var>]
[WORKING_DIRECTORY <var>]
[ARGS <args>...]
)
```

Options

The options specific to **try_run** are:

COMPILE_OUTPUT_VARIABLE <var>
Report the compile step build output in a given variable.

OUTPUT_VARIABLE <var>
Report the compile build output and the output from running the executable in the given variable. This option exists for legacy reasons and is only supported by the old **try_run** signature. Prefer **COMPILE_OUTPUT_VARIABLE** and **RUN_OUTPUT_VARIABLE** instead.

RUN_OUTPUT_VARIABLE <var>
Report the output from running the executable in a given variable.

RUN_OUTPUT_STDOUT_VARIABLE <var>
Added in version 3.25.

Report the output of stdout from running the executable in a given variable.

RUN_OUTPUT_STDERR_VARIABLE <var>
Added in version 3.25.

Report the output of stderr from running the executable in a given variable.

WORKING_DIRECTORY <var>
Added in version 3.20.

Run the executable in the given directory. If no **WORKING_DIRECTORY** is specified, the executable will run in **<bindir>** or the current build directory.

ARGS <args>...
Additional arguments to pass to the executable when running it.

Other Behavior Settings

Set variable *CMAKE_TRY_COMPILE_CONFIGURATION* to choose a build configuration:

- For multi−config generators, this selects which configuration to build.
- For single−config generators, this sets *CMAKE_BUILD_TYPE* in the test project.

Behavior when Cross Compiling

Added in version 3.3: Use **CMAKE_CROSSCOMPILING_EMULATOR** when running cross−compiled binaries.

When cross compiling, the executable compiled in the first step usually cannot be run on the build host. The **try_run** command checks the *CMAKE_CROSSCOMPILING* variable to detect whether CMake is in cross−compiling mode. If that is the case, it will still try to compile the executable, but it will not try to run the executable unless the *CMAKE_CROSSCOMPILING_EMULATOR* variable is set. Instead it will create cache variables which must be filled by the user or by presetting them in some CMake script file to the values the executable would have produced if it had been run on its actual target platform. These cache entries are:

<runResultVar>
Exit code if the executable were to be run on the target platform.

<runResultVar>__TRYRUN_OUTPUT
Output from stdout and stderr if the executable were to be run on the target platform. This is created only if the **RUN_OUTPUT_VARIABLE** or **OUTPUT_VARIABLE** option was used.

In order to make cross compiling your project easier, use **try_run** only if really required. If you use **try_run**, use the **RUN_OUTPUT_STDOUT_VARIABLE**, **RUN_OUTPUT_STDERR_VARIABLE**, **RUN_OUTPUT_VARIABLE** or **OUTPUT_VARIABLE** options only if really required. Using them will require that when cross−compiling, the cache variables will have to be set manually to the output of the executable. You can also "guard" the calls to **try_run** with an *if()* block checking the *CMAKE_CROSSCOMPILING* variable and provide an easy−to−preset alternative for this case.

CTEST COMMANDS

These commands are available only in CTest scripts.

ctest_build

Perform the *CTest Build Step* as a *Dashboard Client*.

```
ctest_build([BUILD <build-dir>] [APPEND]
            [CONFIGURATION <config>]
            [PARALLEL_LEVEL <parallel>]
            [FLAGS <flags>]
            [PROJECT_NAME <project-name>]
            [TARGET <target-name>]
            [NUMBER_ERRORS <num-err-var>]
            [NUMBER_WARNINGS <num-warn-var>]
            [RETURN_VALUE <result-var>]
            [CAPTURE_CMAKE_ERROR <result-var>]
            )
```

Build the project and store results in **Build.xml** for submission with the *ctest_submit()* command.

The *CTEST_BUILD_COMMAND* variable may be set to explicitly specify the build command line. Otherwise the build command line is computed automatically based on the options given.

The options are:

BUILD <build−dir>
Specify the top−level build directory. If not given, the *CTEST_BINARY_DIRECTORY* variable is used.

APPEND
Mark **Build.xml** for append to results previously submitted to a dashboard server since the last *ctest_start()* call. Append semantics are defined by the dashboard server in use. This does *not* cause results to be appended to a **.xml** file produced by a previous call to this command.

CONFIGURATION <config>
Specify the build configuration (e.g. **Debug**). If not specified the**CTEST_B UILD_CONFIGURATION** variable will be checked. Otherwise the−*C* <*cfg*> option gi ven to the *ctest(1)*

command will be used, if any.

PARALLEL_LEVEL <parallel>
Added in version 3.21.

Specify the parallel level of the underlying build system. If not specified, the *CMAKE_BUILD_PARALLEL_LEVEL* environment variable will be checked.

FLAGS <flags>
Pass additional arguments to the underlying build command. If not specified the **CTEST_BUILD_FLAGS** variable will be checked. This can, e.g., be used to trigger a parallel build using the **−j** option of **make**. See the *ProcessorCount* module for an example.

PROJECT_NAME <project−name>
Ignored since CMake 3.0.

Changed in version 3.14: This value is no longer required.

TARGET <target−name>
Specify the name of a target to build. If not specified the**CTEST_B UILD_TARGET** variable will be checked. Otherwise the default target will be built. This is the "all" target (called **ALL_BUILD** in *Visual Studio Generators*).

NUMBER_ERRORS <num−err−var>
Store the number of build errors detected in the given variable.

NUMBER_WARNINGS <num−warn−var>
Store the number of build warnings detected in the given variable.

RETURN_VALUE <result−var>
Store the return value of the native build tool in the given variable.

CAPTURE_CMAKE_ERROR <result−var>
Added in version 3.7.

Store in the **<result−var>** variable −1 if there are any errors running the command and prevent ctest from returning non−zero if an error occurs.

QUIET
Added in version 3.3.

Suppress any CTest−specific non−error output that would have been printed to the console otherwise. The summary of warnings / errors, as well as the output from the native build tool is unaffected by this option.

ctest_configure
Perform the *CTest Configure Step* as a *Dashboard Client*.

```
ctest_configure([BUILD <build−dir>] [SOURCE <source−dir>] [APPEND]
                [OPTIONS <options>] [RETURN_VALUE <result−var>] [QUIET]
                [CAPTURE_CMAKE_ERROR <result−var>])
```

Configure the project build tree and record results in **Configure.xml** for submission with the *ctest_submit()* command.

The options are:

BUILD <build−dir>
Specify the top−level build directory. If not given, the *CTEST_BINARY_DIRECTORY* variable is used.

SOURCE <source−dir>
Specify the source directory. If not given, the *CTEST_SOURCE_DIRECTORY* variable is used.

APPEND
Mark **Configure.xml** for append to results previously submitted to a dashboard server since the last *ctest_start()* call. Append semantics are defined by the dashboard server in use. This does *not* cause results to be appended to a **.xml** file produced by a previous call to this command.

OPTIONS <options>
Specify command−line arguments to pass to the configuration tool.

RETURN_VALUE <result−var>
Store in the **<result−var>** variable the return value of the native configuration tool.

CAPTURE_CMAKE_ERROR <result−var>
Added in version 3.7.

Store in the **<result−var>** variable −1 if there are any errors running the command and prevent ctest from returning non−zero if an error occurs.

QUIET
Added in version 3.3.

Suppress any CTest−specific non−error messages that would have otherwise been printed to the console. Output from the underlying configure command is not affected.

ctest_coverage
Perform the *CTest Coverage Step* as a *Dashboard Client*.

```
ctest_coverage([BUILD <build-dir>] [APPEND]
               [LABELS <label>...]
               [RETURN_VALUE <result-var>]
               [CAPTURE_CMAKE_ERROR <result-var>]
               [QUIET]
               )
```

Collect coverage tool results and stores them in **Coverage.xml** for submission with the *ctest_submit()* command.

The options are:

BUILD <build−dir>
Specify the top−level build directory. If not given, the *CTEST_BINARY_DIRECTORY* variable is used.

APPEND
Mark **Coverage.xml** for append to results previously submitted to a dashboard server since the last *ctest_start()* call. Append semantics are defined by the dashboard server in use. This does *not* cause results to be appended to a **.xml** file produced by a previous call to this command.

LABELS
Filter the coverage report to include only source files labeled with at least one of the labels specified.

RETURN_VALUE <result−var>
Store in the **<result−var>** variable **0** if coverage tools ran without error and non−zero otherwise.

CAPTURE_CMAKE_ERROR <result−var>
Added in version 3.7.

Store in the **<result−var>** variable −1 if there are any errors running the command and prevent ctest from returning non−zero if an error occurs.

QUIET
Added in version 3.3.

Suppress any CTest−specific non−error output that would have been printed to the console otherwise. The summary indicating how many lines of code were covered is unaffected by this option.

ctest_empty_binary_directory
empties the binary directory

```
ctest_empty_binary_directory(<directory>)
```

Removes a binary directory. This command will perform some checks prior to deleting the directory in an attempt to avoid malicious or accidental directory deletion.

ctest_memcheck
Perform the *CTest MemCheck Step* as a *Dashboard Client*.

```
ctest_memcheck([BUILD <build−dir>] [APPEND]
               [START <start−number>]
               [END <end−number>]
               [STRIDE <stride−number>]
               [EXCLUDE <exclude−regex>]
               [INCLUDE <include−regex>]
               [EXCLUDE_LABEL <label−exclude−regex>]
               [INCLUDE_LABEL <label−include−regex>]
               [EXCLUDE_FIXTURE <regex>]
               [EXCLUDE_FIXTURE_SETUP <regex>]
               [EXCLUDE_FIXTURE_CLEANUP <regex>]
               [PARALLEL_LEVEL <level>]
               [RESOURCE_SPEC_FILE <file>]
               [TEST_LOAD <threshold>]
               [SCHEDULE_RANDOM <ON|OFF>]
               [STOP_ON_FAILURE]
               [STOP_TIME <time−of−day>]
               [RETURN_VALUE <result−var>]
               [CAPTURE_CMAKE_ERROR <result−var>]
               [REPEAT <mode>:<n>]
               [OUTPUT_JUNIT <file>]
               [DEFECT_COUNT <defect−count−var>]
               [QUIET]
               )
```

Run tests with a dynamic analysis tool and store results in **MemCheck.xml** for submission with the *ctest_submit()* command.

Most options are the same as those for the *ctest_test()* command.

The options unique to this command are:

DEFECT_COUNT <defect−count−var>
Added in version 3.8.

Store in the **<defect−count−var>** the number of defects found.

ctest_read_custom_files
read CTestCustom files.

```
ctest_read_custom_files(<directory>...)
```

Read all the CTestCustom.ctest or CTestCustom.cmake files from the given directory.

By default, invoking *ctest(1)* without a script will read custom files from the binary directory.

ctest_run_script
runs a *ctest −S* script

```
ctest_run_script([NEW_PROCESS] script_file_name script_file_name1
            script_file_name2 ... [RETURN_VALUE var])
```

Runs a script or scripts much like if it was run from *ctest −S*. If**NEW_PR OCESS** is specified then each script will be run in a separate process. If **RETURN_VALUE** is specified the return value of the last script run will be put into **var**.

ctest_sleep
sleeps for some amount of time

```
ctest_sleep(<seconds>)
```

Sleep for given number of seconds.

```
ctest_sleep(<time1> <duration> <time2>)
```

Sleep for t=(time1 + duration − time2) seconds if t > 0.

ctest_start
Starts the testing for a given model

```
ctest_start(<model> [<source> [<binary>]] [GROUP <group>] [QUIET])

ctest_start([<model> [<source> [<binary>]]] [GROUP <group>] APPEND [QUIET])
```

Starts the testing for a given model. The command should be called after the binary directory is initialized.

The parameters are as follows:

<model>
Set the dashboard model. Must be one of **Experimental**, **Continuous**, or **Nightly**. This parameter is required unless **APPEND** is specified.

<source>
Set the source directory. If not specified, the value of *CTEST_SOURCE_DIRECTORY* is used instead.

<binary>
Set the binary directory. If not specified, the value of *CTEST_BINARY_DIRECTORY* is used instead.

GROUP <group>
If **GROUP** is used, the submissions will go to the specified group on the CDash server. If no **GROUP** is specified, the name of the model is used by default.

Changed in version 3.16: This replaces the deprecated option **TRACK**. Despite the name change its behavior is unchanged.

APPEND
If **APPEND** is used, the existing **TAG** is used rather than creating a new one based on the current time stamp. If you use **APPEND**, you can omit the **<model>** and **GROUP <group>** parameters, because they will be read from the generated **TAG** file. For example:

```
ctest_start(Experimental GROUP GroupExperimental)
```

Later, in another *ctest −S* script:

```
ctest_start(APPEND)
```

When the second script runs **ctest_start(APPEND)**, it will read the **Experimental** model and **GroupExperimental** group from the **TAG** file generated by the first **ctest_start()** command. Please note that if you call **ctest_start(APPEND)** and specify a different model or group than in the first **ctest_start()** command, a warning will be issued, and the new model and group will be used.

QUIET
Added in version 3.3.

If **QUIET** is used, CTest will suppress any non−error messages that it otherwise would have printed to the console.

The parameters for **ctest_start()** can be issued in any order, with the exception that **<model>**, **<source>**, and **<binary>** have to appear in that order with respect to each other. The following are all valid and equivalent:

```
ctest_start(Experimental path/to/source path/to/binary GROUP SomeGroup QUIET APPE

ctest_start(GROUP SomeGroup Experimental QUIET path/to/source APPEND path/to/bina

ctest_start(APPEND QUIET Experimental path/to/source GROUP SomeGroup path/to/bina
```

However, for the sake of readability, it is recommended that you order your parameters in the order listed at the top of this page.

If the *CTEST_CHECKOUT_COMMAND* variable (or the *CTEST_CVS_CHECKOUT* variable) is set, its content is treated as command−line. The command is invoked with the current working directory set to the parent of the source directory, even if the source directory already exists. This can be used to create the source tree from a version control repository.

ctest_submit
Perform the *CTest Submit Step* as a *Dashboard Client*.

```
ctest_submit([PARTS <part>...] [FILES <file>...]
             [SUBMIT_URL <url>]
             [BUILD_ID <result-var>]
             [HTTPHEADER <header>]
             [RETRY_COUNT <count>]
             [RETRY_DELAY <delay>]
             [RETURN_VALUE <result-var>]
             [CAPTURE_CMAKE_ERROR <result-var>]
             [QUIET]
             )
```

Submit results to a dashboard server. By default all available parts are submitted.

The options are:

PARTS <part>...
Specify a subset of parts to submit. Valid part names are:

- **Start** − nothing.
- **Update** − *ctest_update()* results, in **Update.xml**.
- **Configure** − *ctest_configure()* results, in **Configure.xml**.
- **Build** − *ctest_build()* results, in **Build.xml**.
- **Test** − *ctest_test()* results, in **Test.xml**.
- **Coverage** − *ctest_coverage()* results, in **Coverage.xml**.
- **MemCheck** − *ctest_memcheck()* results, in **DynamicAnalysis.xml** and **DynamicAnalysis−Test.xml**.
- **Notes** − Files listed by *CTEST_NOTES_FILES*, in **Notes.xml**.
- **ExtraFiles** − Files listed by *CTEST_EXTRA_SUBMIT_FILES*.
- **Upload** − Files prepared for upload by *ctest_upload()*, in **Upload.xml**.
- **Submit** − nothing.
- **Done** − Build is complete, in **Done.xml**.

FILES <file>...
Specify an explicit list of specific files to be submitted. Each individual file must exist at the time of the call.

SUBMIT_URL <url>
Added in version 3.14.

The **http** or **https** URL of the dashboard server to send the submission to. If not given, the *CTEST_SUBMIT_URL* variable is used.

BUILD_ID <result−var>
Added in version 3.15.

Store in the **<result−var>** variable the ID assigned to this build by CDash.

HTTPHEADER <HTTP−header>
Added in version 3.9.

Specify HTTP header to be included in the request to CDash during submission. For example,

CDash can be configured to only accept submissions from authenticated clients. In this case, you should provide a bearer token in your header:

```
ctest_submit(HTTPHEADER "Authorization: Bearer <auth-token>")
```

This suboption can be repeated several times for multiple headers.

RETRY_COUNT <count>
Specify how many times to retry a timed-out submission.

RETRY_DELAY <delay>
Specify how long (in seconds) to wait after a timed-out submission before attempting to re-submit.

RETURN_VALUE <result-var>
Store in the **<result-var>** variable **0** for success and non-zero on failure.

CAPTURE_CMAKE_ERROR <result-var>
Added in version 3.13.

Store in the **<result-var>** variable -1 if there are any errors running the command and prevent ctest from returning non-zero if an error occurs.

QUIET
Added in version 3.3.

Suppress all non-error messages that would have otherwise been printed to the console.

Submit to CDash Upload API
Added in version 3.2.

```
ctest_submit(CDASH_UPLOAD <file> [CDASH_UPLOAD_TYPE <type>]
             [SUBMIT_URL <url>]
             [BUILD_ID <result-var>]
             [HTTPHEADER <header>]
             [RETRY_COUNT <count>]
             [RETRY_DELAY <delay>]
             [RETURN_VALUE <result-var>]
             [QUIET])
```

This second signature is used to upload files to CDash via the CDash file upload API. The API first sends a request to upload to CDash along with a content hash of the file. If CDash does not already have the file, then it is uploaded. Along with the file, a CDash type string is specified to tell CDash which handler to use to process the data.

This signature interprets options in the same way as the first one.

Added in version 3.8: Added the **RETRY_COUNT**, **RETRY_DELAY**, **QUIET** options.

Added in version 3.9: Added the **HTTPHEADER** option.

Added in version 3.13: Added the **RETURN_VALUE** option.

Added in version 3.14: Added the **SUBMIT_URL** option.

Added in version 3.15: Added the **BUILD_ID** option.

ctest_test

Perform the *CTest Test Step* as a *Dashboard Client*.

```
ctest_test([BUILD <build-dir>] [APPEND]
           [START <start-number>]
           [END <end-number>]
           [STRIDE <stride-number>]
           [EXCLUDE <exclude-regex>]
           [INCLUDE <include-regex>]
           [EXCLUDE_LABEL <label-exclude-regex>]
           [INCLUDE_LABEL <label-include-regex>]
           [EXCLUDE_FROM_FILE <filename>]
           [INCLUDE_FROM_FILE <filename>]
           [EXCLUDE_FIXTURE <regex>]
           [EXCLUDE_FIXTURE_SETUP <regex>]
           [EXCLUDE_FIXTURE_CLEANUP <regex>]
           [PARALLEL_LEVEL [<level>]]
           [RESOURCE_SPEC_FILE <file>]
           [TEST_LOAD <threshold>]
           [SCHEDULE_RANDOM <ON|OFF>]
           [STOP_ON_FAILURE]
           [STOP_TIME <time-of-day>]
           [RETURN_VALUE <result-var>]
           [CAPTURE_CMAKE_ERROR <result-var>]
           [REPEAT <mode>:<n>]
           [OUTPUT_JUNIT <file>]
           [QUIET]
           )
```

Run tests in the project build tree and store results in **Test.xml** for submission with the *ctest_submit()* command.

The options are:

BUILD <build-dir>
Specify the top-level build directory. If not given, the *CTEST_BINARY_DIRECTORY* variable is used.

APPEND
Mark **Test.xml** for append to results previously submitted to a dashboard server since the last *ctest_start()* call. Append semantics are defined by the dashboard server in use. This does *not* cause results to be appended to a **.xml** file produced by a previous call to this command.

START <start-number>
Specify the beginning of a range of test numbers.

END <end-number>
Specify the end of a range of test numbers.

STRIDE <stride-number>
Specify the stride by which to step across a range of test numbers.

EXCLUDE <exclude−regex>
Specify a regular expression matching test names to exclude.

INCLUDE <include−regex>
Specify a regular expression matching test names to include. Tests not matching this expression are excluded.

EXCLUDE_LABEL <label−exclude−regex>
Specify a regular expression matching test labels to exclude.

INCLUDE_LABEL <label−include−regex>
Specify a regular expression matching test labels to include. Tests not matching this expression are excluded.

EXCLUDE_FROM_FILE <filename>
Added in version 3.29.

Do NOT run tests listed with their exact name in the given file.

INCLUDE_FROM_FILE <filename>
Added in version 3.29.

Only run the tests listed with their exact name in the given file.

EXCLUDE_FIXTURE <regex>
Added in version 3.7.

If a test in the set of tests to be executed requires a particular fixture, that fixture's setup and cleanup tests would normally be added to the test set automatically. This option prevents adding setup or cleanup tests for fixtures matching the **<regex>**. Note that all other fixture behavior is retained, including test dependencies and skipping tests that have fixture setup tests that fail.

EXCLUDE_FIXTURE_SETUP <regex>
Added in version 3.7.

Same as **EXCLUDE_FIXTURE** except only matching setup tests are excluded.

EXCLUDE_FIXTURE_CLEANUP <regex>
Added in version 3.7.

Same as **EXCLUDE_FIXTURE** except only matching cleanup tests are excluded.

PARALLEL_LEVEL [<level>]
Run tests in parallel, limited to a given level of parallelism.

Added in version 3.29: The **<level>** may be omitted, or **0**, to let ctest use a default level of parallelism, or unbounded parallelism, respectively, as documented by the *ctest −−parallel* option.

RESOURCE_SPEC_FILE <file>
Added in version 3.16.

Specify a *resource specification file*. See *Resource Allocation* for more information.

TEST_LOAD <threshold>
Added in version 3.4.

While running tests in parallel, try not to start tests when they may cause the CPU load to pass above a given threshold. If not specified the *CTEST_TEST_LOAD* variable will be checked, and then the *−−test−load* command−line argument to *ctest(1)*. See also the **TestLoad** setting in the *CTest Test Step*.

REPEAT <mode>:<n>
Added in version 3.17.

Run tests repeatedly based on the given **<mode>** up to **<n>** times. The modes are:

UNTIL_FAIL
Require each test to run **<n>** times without failing in order to pass. This is useful in finding sporadic failures in test cases.

UNTIL_PASS
Allow each test to run up to **<n>** times in order to pass. Repeats tests if they fail for any reason. This is useful in tolerating sporadic failures in test cases.

AFTER_TIMEOUT
Allow each test to run up to **<n>** times in order to pass. Repeats tests only if they timeout. This is useful in tolerating sporadic timeouts in test cases on busy machines.

SCHEDULE_RANDOM <ON|OFF>
Launch tests in a random order. This may be useful for detecting implicit test dependencies.

STOP_ON_FAILURE
Added in version 3.18.

Stop the execution of the tests once one has failed.

STOP_TIME <time−of−day>
Specify a time of day at which the tests should all stop running.

RETURN_VALUE <result−var>
Store in the **<result−var>** variable **0** if all tests passed. Store non−zero if anything went wrong.

CAPTURE_CMAKE_ERROR <result−var>
Added in version 3.7.

Store in the **<result−var>** variable −1 if there are any errors running the command and prevent ctest from returning non−zero if an error occurs.

OUTPUT_JUNIT <file>
Added in version 3.21.

Write test results to **<file>** in JUnit XML format. If **<file>** is a relative path, it will be placed in the build directory. If **<file>** already exists, it will be overwritten. Note that the resulting JUnit XML file is **not** uploaded to CDash because it would be redundant with CTest's **Test.xml** file.

QUIET
Added in version 3.3.

Suppress any CTest−specific non−error messages that would have otherwise been printed to the console. Output from the underlying test command is not affected. Summary info detailing the percentage of passing tests is also unaffected by the **QUIET** option.

See also the *CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE*, *CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE* and *CTEST_CUSTOM_TEST_OUTPUT_TRUNCATION* variables, along with their corresponding *ctest(1)* command line options *−−test−output−size−passed*, *−−test−output−size−failed*, and *−−test−output−truncation*.

Additional Test Measurements

CTest can parse the output of your tests for extra measurements to report to CDash.

When run as a *Dashboard Client*, CTest will include these custom measurements in the **Test.xml** file that gets uploaded to CDash.

Check the *CDash test measurement documentation* for more information on the types of test measurements that CDash recognizes.

The following example demonstrates how to output a variety of custom test measurements.

```
std::cout <<
  "<CTestMeasurement type=\"numeric/double\" name=\"score\">28.3</CTestMeasuremen
  << std::endl;

std::cout <<
  "<CTestMeasurement type=\"text/string\" name=\"color\">red</CTestMeasurement>"
  << std::endl;

std::cout <<
  "<CTestMeasurement type=\"text/link\" name=\"CMake URL\">https://cmake.org</CTe
  << std::endl;

std::cout <<
  "<CTestMeasurement type=\"text/preformatted\" name=\"Console Output\">" <<
  "line 1.\n" <<
  "  \033[31;1m line 2. Bold red, and indented!\033[0;0ml\n" <<
  "line 3. Not bold or indented...\n" <<
  "</CTestMeasurement>" << std::endl;
```

Image Measurements

The following example demonstrates how to upload test images to CDash.

```
std::cout <<
  "<CTestMeasurementFile type=\"image/jpg\" name=\"TestImage\">" <<
  "/dir/to/test_img.jpg</CTestMeasurementFile>" << std::endl;

std::cout <<
  "<CTestMeasurementFile type=\"image/gif\" name=\"ValidImage\">" <<
  "/dir/to/valid_img.gif</CTestMeasurementFile>" << std::endl;

std::cout <<
  "<CTestMeasurementFile type=\"image/png\" name=\"AlgoResult\">" <<
  "/dir/to/img.png</CTestMeasurementFile>"
  << std::endl;
```

Images will be displayed together in an interactive comparison mode on CDash if they are provided with two or more of the following names.

- **TestImage**
- **ValidImage**
- **BaselineImage**
- **DifferenceImage2**

By convention, **TestImage** is the image generated by your test, and **ValidImage** (or **BaselineImage**) is basis of comparison used to determine if the test passed or failed.

If another image name is used it will be displayed by CDash as a static image separate from the interactive comparison UI.

Attached Files
Added in version 3.21.

The following example demonstrates how to upload non−image files to CDash.

```
std::cout <<
  "<CTestMeasurementFile type=\"file\" name=\"TestInputData1\">" <<
  "/dir/to/data1.csv</CTestMeasurementFile>\n"                   <<
  "<CTestMeasurementFile type=\"file\" name=\"TestInputData2\">" <<
  "/dir/to/data2.csv</CTestMeasurementFile>"                     << std::endl;
```

If the name of the file to upload is known at configure time, you can use the *ATTACHED_FILES* or *ATTACHED_FILES_ON_FAIL* test properties instead.

Custom Details
Added in version 3.21.

The following example demonstrates how to specify a custom value for the **Test Details** field displayed on CDash.

```
std::cout <<
  "<CTestDetails>My Custom Details Value</CTestDetails>" << std::endl;
```

Additional Labels
Added in version 3.22.

The following example demonstrates how to add additional labels to a test at runtime.

```
std::cout <<
  "<CTestLabel>Custom Label 1</CTestLabel>\n" <<
  "<CTestLabel>Custom Label 2</CTestLabel>"   << std::endl;
```

Use the *LABELS* test property instead for labels that can be determined at configure time.

ctest_update
Perform the *CTest Update Step* as a *Dashboard Client*.

```
ctest_update([SOURCE <source−dir>]
             [RETURN_VALUE <result−var>]
```

```
  [CAPTURE_CMAKE_ERROR <result-var>]
  [QUIET])
```

Update the source tree from version control and record results in **Update.xml** for submission with the *ctest_submit()* command.

The options are:

SOURCE <source-dir>
Specify the source directory. If not given, the *CTEST_SOURCE_DIRECTORY* variable is used.

RETURN_VALUE <result-var>
Store in the **<result-var>** variable the number of files updated or **-1** on error.

CAPTURE_CMAKE_ERROR <result-var>
Added in version 3.13.

Store in the **<result-var>** variable -1 if there are any errors running the command and prevent ctest from returning non-zero if an error occurs.

QUIET
Added in version 3.3.

Tell CTest to suppress most non-error messages that it would have otherwise printed to the console. CTest will still report the new revision of the repository and any conflicting files that were found.

The update always follows the version control branch currently checked out in the source directory. See the *CTest Update Step* documentation for information about variables that change the behavior of **ctest_update()**.

ctest_upload
Upload files to a dashboard server as a *Dashboard Client*.

```
ctest_upload(FILES <file>... [QUIET] [CAPTURE_CMAKE_ERROR <result-var>])
```

The options are:

FILES <file>...
Specify a list of files to be sent along with the build results to the dashboard server.

QUIET
Added in version 3.3.

Suppress any CTest-specific non-error output that would have been printed to the console otherwise.

CAPTURE_CMAKE_ERROR <result-var>
Added in version 3.7.

Store in the **<result-var>** variable -1 if there are any errors running the command and prevent ctest from returning non-zero if an error occurs.

DEPRECATED COMMANDS

These commands are deprecated and are only made available to maintain backward compatibility. The documentation of each command states the CMake version in which it was deprecated. Do not use these

commands in new code.

build_name
Disallowed since version 3.0. See CMake Policy *CMP0036*.

Use **${CMAKE_SYSTEM}** and **${CMAKE_CXX_COMPILER}** instead.

```
build_name(variable)
```

Sets the specified variable to a string representing the platform and compiler settings. These values are now available through the *CMAKE_SYSTEM* and *CMAKE_CXX_COMPILER* variables.

exec_program
Changed in version 3.28: This command is available only if policy *CMP0153* is not set to **NEW**. Port projects to the *execute_process()* command.

Deprecated since version 3.0: Use the *execute_process()* command instead.

Run an executable program during the processing of the CMakeList.txt file.

```
exec_program(Executable [directory in which to run]
             [ARGS <arguments to executable>]
             [OUTPUT_VARIABLE <var>]
             [RETURN_VALUE <var>])
```

The executable is run in the optionally specified directory. The executable can include arguments if it is double quoted, but it is better to use the optional **ARGS** argument to specify arguments to the program. This is because cmake will then be able to escape spaces in the executable path. An optional argument **OUTPUT_VARIABLE** specifies a variable in which to store the output. To capture the return value of the execution, provide a **RETURN_VALUE**. If**OUTPUT_VARIABLE** is specified, then no output will go to the stdout/stderr of the console running cmake.

export_library_dependencies
Disallowed since version 3.0. See CMake Policy *CMP0033*.

Use *install(EXPORT)* or *export()* command.

This command generates an old−style library dependencies file. Projects requiring CMake 2.6 or later should not use the command. Use instead the *install(EXPORT)* command to help export targets from an installation tree and the *export()* command to export targets from a build tree.

The old−style library dependencies file does not take into account per−configuration names of libraries or the *LINK_INTERFACE_LIBRARIES* target property.

```
export_library_dependencies(<file> [APPEND])
```

Create a file named **<file>** that can be included into a CMake listfile with the INCLUDE command. The file will contain a number of SET commands that will set all the variables needed for library dependency information. This should be the last command in the top level CMakeLists.txt file of the project. If the **APPEND** option is specified, the SET commands will be appended to the given file instead of replacing it.

install_files
Deprecated since version 3.0: Use the *install(FILES)* command instead.

This command has been superseded by the *install()* command. It is provided for compatibility with older CMake code. The **FILES** form is directly replaced by the **FILES** form of the *install()* command. The regexp form can be expressed more clearly using the **GLOB** form of the *file()* command.

```
install_files(<dir> extension file file ...)
```

Create rules to install the listed files with the given extension into the given directory. Only files existing in the current source tree or its corresponding location in the binary tree may be listed. If a file specified already has an extension, that extension will be removed first. This is useful for providing lists of source files such as foo.cxx when you want the corresponding foo.h to be installed. A typical extension is **.h**.

```
install_files(<dir> regexp)
```

Any files in the current source directory that match the regular expression will be installed.

```
install_files(<dir> FILES file file ...)
```

Any files listed after the **FILES** keyword will be installed explicitly from the names given. Full paths are allowed in this form.

The directory **<dir>** is relative to the installation prefix, which is stored in the variable *CMAKE_INSTALL_PREFIX*.

install_programs

Deprecated since version 3.0: Use the *install(PROGRAMS)* command instead.

This command has been superseded by the *install()* command. It is provided for compatibility with older CMake code. The **FILES** form is directly replaced by the **PROGRAMS** form of the *install()* command. The regexp form can be expressed more clearly using the **GLOB** form of the *file()* command.

```
install_programs(<dir> file1 file2 [file3 ...])
install_programs(<dir> FILES file1 [file2 ...])
```

Create rules to install the listed programs into the given directory. Use the**FILES** ar gument to guarantee that the file list version of the command will be used even when there is only one argument.

```
install_programs(<dir> regexp)
```

In the second form any program in the current source directory that matches the regular expression will be installed.

This command is intended to install programs that are not built by cmake, such as shell scripts. See the **TARGETS** form of the *install()* command to create installation rules for targets built by cmake.

The directory **<dir>** is relative to the installation prefix, which is stored in the variable *CMAKE_INSTALL_PREFIX*.

install_targets

Deprecated since version 3.0: Use the *install(TARGETS)* command instead.

This command has been superseded by the *install()* command. It is provided for compatibility with older CMake code.

```
install_targets(<dir> [RUNTIME_DIRECTORY dir] target target)
```

Create rules to install the listed targets into the given directory. The directory**<dir>** is relati ve to the installation prefix, which is stored in the variable *CMAKE_INSTALL_PREFIX*. If**R UNTIME_DIRECTORY** is specified, then on systems with special runtime files (Windows DLL), the files will be copied to that directory.

load_command

Disallowed since version 3.0. See CMake Policy *CMP0031*.

Load a command into a running CMake.

```
load_command(COMMAND_NAME <loc1> [loc2 ...])
```

The given locations are searched for a library whose name is cmCOMMAND_NAME. If found, it is loaded as a module and the command is added to the set of available CMake commands. Usually, *try_compile()* is used before this command to compile the module. If the command is successfully loaded a variable named **CMAKE_LOADED_COMMAND_<COMMAND_NAME>** will be set to the full path of the module that was loaded. Otherwise the variable will not be set.

make_directory

Deprecated since version 3.0: Use the *file(MAKE_DIRECTORY)* command instead.

```
make_directory(directory)
```

Creates the specified directory. Full paths should be given. Any parent directories that do not exist will also be created. Use with care.

output_required_files

Disallowed since version 3.0. See CMake Policy *CMP0032*.

Approximate C preprocessor dependency scanning.

This command exists only because ancient CMake versions provided it. CMake handles preprocessor dependency scanning automatically using a more advanced scanner.

```
output_required_files(srcfile outputfile)
```

Outputs a list of all the source files that are required by the specified **srcfile**. This list is written into**outputfile**. This is similar to writing out the dependencies for**sr cfile** except that it jumps from **.h** files into **.cxx**, **.c** and **.cpp** files if possible.

qt_wrap_cpp

Deprecated since version 3.14: This command was originally added to support Qt 3 before the *add_custom_command()* command was sufficiently mature. The *FindQt4* module provides the **qt4_wrap_cpp()** macro, which should be used instead for Qt 4 projects. For projects using Qt 5 or later, use the equivalent macro provided by Qt itself (e.g. Qt 5 provides *qt5_wrap_cpp()*).

Manually create Qt Wrappers.

```
qt_wrap_cpp(resultingLibraryName DestName SourceLists ...)
```

Produces moc files for all the .h files listed in the SourceLists. The moc files will be added to the library using the **DestName** source list.

Consider updating the project to use the *AUTOMOC* target property instead for a more automated way of invoking the **moc** tool.

qt_wrap_ui
Deprecated since version 3.14: This command was originally added to support Qt 3 before the *add_custom_command()* command was sufficiently mature. The *FindQt4* module provides the **qt4_wrap_ui()** macro, which should be used instead for Qt 4 projects. For projects using Qt 5 or later, use the equivalent macro provided by Qt itself (e.g. Qt 5 provides **qt5_wrap_ui()**).

Manually create Qt user interfaces Wrappers.

```
qt_wrap_ui(resultingLibraryName HeadersDestName
           SourcesDestName SourceLists ...)
```

Produces .h and .cxx files for all the .ui files listed in the **SourceLists**. The .h files will be added to the library using the **HeadersDestNamesource** list. The .cxx files will be added to the library using the **SourcesDestNamesource** list.

Consider updating the project to use the *AUTOUIC* target property instead for a more automated way of invoking the **uic** tool.

remove
Deprecated since version 3.0: Use the *list(REMOVE_ITEM)* command instead.

```
remove(VAR VALUE VALUE ...)
```

Removes **VALUE** from the variable **VAR**. This is typically used to remove entries from a vector (e.g. semicolon separated list). **VALUE** is expanded.

subdir_depends
Disallowed since version 3.0. See CMake Policy *CMP0029*.

Does nothing.

```
subdir_depends(subdir dep1 dep2 ...)
```

Does not do anything. This command used to help projects order parallel builds correctly. This functionality is now automatic.

subdirs
Deprecated since version 3.0: Use the *add_subdirectory()* command instead.

Add a list of subdirectories to the build.

```
subdirs(dir1 dir2 ...[EXCLUDE_FROM_ALL exclude_dir1 exclude_dir2 ...]
        [PREORDER])
```

Add a list of subdirectories to the build. The*add_subdir ectory()* command should be used instead of **subdirs** although **subdirs** will still work. This will cause any CMakeLists.txt files in the sub directories to be processed by CMake. Any directories after the **PREORDER** flag are traversed first by makefile builds, the **PREORDER** flag has no effect on IDE projects. Any directories after the **EXCLUDE_FROM_ALL** marker will not be included in the top level makefile or project file. This is useful for having CMake create makefiles or projects for a set of examples in a project. You would want CMake to generate makefiles or project files for all the examples at the same time, but you would not want them to show up in the top level project or be built each time make is run from the top.

use_mangled_mesa
Disallowed since version 3.0. See CMake Policy *CMP0030*.

Copy mesa headers for use in combination with system GL.

```
use_mangled_mesa(PATH_TO_MESA OUTPUT_DIRECTORY)
```

The path to mesa includes, should contain **gl_mangle.h**. The mesa headers are copied to the specified output directory. This allows mangled mesa headers to override other GL headers by being added to the include directory path earlier.

utility_source
Disallowed since version 3.0. See CMake Policy *CMP0034*.

Specify the source tree of a third−party utility.

```
utility_source(cache_entry executable_name
               path_to_source [file1 file2 ...])
```

When a third−party utility's source is included in the distribution, this command specifies its location and name. The cache entry will not be set unless the**path_to_sour ce** and all listed files exist. It is assumed that the source tree of the utility will have been built before it is needed.

When cross compiling CMake will print a warning if a **utility_source()** command is executed, because in many cases it is used to build an executable which is executed later on. This doesn't work when cross compiling, since the executable can run only on their target platform. So in this case the cache entry has to be adjusted manually so it points to an executable which is runnable on the build host.

variable_requires
Disallowed since version 3.0. See CMake Policy *CMP0035*.

Use the *if()* command instead.

Assert satisfaction of an option's required variables.

```
variable_requires(TEST_VARIABLE RESULT_VARIABLE
                  REQUIRED_VARIABLE1
                  REQUIRED_VARIABLE2 ...)
```

The first argument (**TEST_VARIABLE**) is the name of the variable to be tested, if that variable is false nothing else is done. If **TEST_VARIABLE** is true, then the next argument (**RESULT_VARIABLE**) is a variable that is set to true if all the required variables are set. The rest of the arguments are variables that must be true or not set to **NOTFOUND** to avoid an error. If any are not true, an error is reported.

write_file
Deprecated since version 3.0: Use the *file(WRITE)* command instead.

```
write_file(filename "message to write"... [APPEND])
```

The first argument is the file name, the rest of the arguments are messages to write. If the argument **APPEND** is specified, then the message will be appended.

NOTE 1: *file(WRITE)* and *file(APPEND)* do exactly the same as this one but add some more functionality.

NOTE 2: When using **write_file** the produced file cannot be used as an input to CMake

(CONFIGURE_FILE, source file ...) because it will lead to an infinite loop. Use *configure_file()* if you want to generate input files to CMake.

COPYRIGHT