

NAME

cmake-modules – CMake Modules Reference

The modules listed here are part of the CMake distribution. Projects may provide further modules; their location(s) can be specified in the *CMAKE_MODULE_PATH* variable.

UTILITY MODULES

These modules are loaded using the *include()* command.

AndroidTestUtilities

Added in version 3.7.

Create a test that automatically loads specified data onto an Android device.

Introduction

Use this module to push data needed for testing an Android device behavior onto a connected Android device. The module will accept files and libraries as well as separate destinations for each. It will create a test that loads the files into a device object store and link to them from the specified destination. The files are only uploaded if they are not already in the object store.

For example:

```
include(AndroidTestUtilities)
android_add_test_data(
  example_setup_test
  FILES <files>...
  LIBS <libs>...
  DEVICE_TEST_DIR "/data/local/tests/example"
  DEVICE_OBJECT_STORE "/sdcard/.ExternalData/SHA"
)
```

At build time a test named "example_setup_test" will be created. Run this test on the command line with *ctest(1)* to load the data onto the Android device.

Module Functions**android_add_test_data**

```
android_add_test_data(<test-name>
  [FILES <files>...] [FILES_DEST <device-dir>]
  [LIBS <libs>...] [LIBS_DEST <device-dir>]
  [DEVICE_OBJECT_STORE <device-dir>]
  [DEVICE_TEST_DIR <device-dir>]
  [NO_LINK_REGEX <strings>...]
)
```

The **android_add_test_data** function is used to copy files and libraries needed to run project-specific tests. On the host operating system, this is done at build time. For on-device testing, the files are loaded onto the device by the manufactured test at run time.

This function accepts the following named parameters:

FILES <files>...

zero or more files needed for testing

LIBS <libs>...

zero or more libraries needed for testing

FILES_DEST <device-dir>

absolute path where the data files are expected to be

LIBS_DEST <device-dir>

absolute path where the libraries are expected to be

DEVICE_OBJECT_STORE <device-dir>

absolute path to the location where the data is stored on-device

DEVICE_TEST_DIR <device-dir>

absolute path to the root directory of the on-device test location

NO_LINK_REGEX <strings>...

list of regex strings matching the names of files that should be copied from the object store to the testing directory

BundleUtilities

Functions to help assemble a standalone bundle application.

A collection of CMake utility functions useful for dealing with **.app** bundles on the Mac and bundle-like directories on any OS.

NOTE:

Do not use these functions at configure time (from **CMakeLists.txt**). Instead, invoke them from an *install(CODE)* or *install(SCRIPT)*.

Functions

The following functions are provided by this module:

fixup_bundle

```
fixup_bundle(<app> <libs> <dirs> [IGNORE_ITEM <file>...])
```

Fix up <app> bundle in-place and make it standalone, such that it can be drag-n-drop copied to another machine and run on that machine as long as all of the system libraries are compatible.

If you pass plugins to **fixup_bundle** as the libs parameter, you should install them or copy them into the bundle before calling **fixup_bundle**. The <libs> parameter is a list of libraries that must be fixed up, but that cannot be determined by **otool** output analysis (i.e. **plugins**).

Gather all the keys for all the executables and libraries in a bundle, and then, for each key, copy each prerequisite into the bundle. Then fix each one up according to its own list of prerequisites.

Then clear all the keys and call **verify_app** on the final bundle to ensure that it is truly standalone.

Added in version 3.6: As an optional parameter (**IGNORE_ITEM**) a list of file names can be passed, which are then ignored (e.g. **IGNORE_ITEM "vcredist_x86.exe;vcredist_x64.exe"**).

copy_and_fixup_bundle

```
copy_and_fixup_bundle(<src> <dst> <libs> <dirs>)
```

Makes a copy of the bundle <src> at location <dst> and then fixes up the new copied bundle in-place at <dst>.

verify_app

```
verify_app(<app> [IGNORE_ITEM <file>...])
```

Verifies that an application **<app>** appears valid based on running analysis tools on it. Calls *message(FATAL_ERROR)* if the application is not verified.

Added in version 3.6: As an optional parameter (**IGNORE_ITEM**) a list of file names can be passed, which are then ignored (e.g. **IGNORE_ITEM "vcredist_x86.exe;vcredist_x64.exe"**)

get_bundle_main_executable

```
get_bundle_main_executable(<bundle> <result_var>)
```

The result will be the full path name of the bundle's main executable file or an **error:** prefixed string if it could not be determined.

get_dotapp_dir

```
get_dotapp_dir(<exe> <dotapp_dir_var>)
```

Returns the nearest parent dir whose name ends with **.app** given the full path to an executable. If there is no such parent dir, then simply return the dir containing the executable.

The returned directory may or may not exist.

get_bundle_and_executable

```
get_bundle_and_executable(<app> <bundle_var> <executable_var> <valid_var>)
```

Takes either a **.app** directory name or the name of an executable nested inside a **.app** directory and returns the path to the **.app** directory in **<bundle_var>** and the path to its main executable in **<executable_var>**.

get_bundle_all_executables

```
get_bundle_all_executables(<bundle> <exes_var>)
```

Scans **<bundle>** bundle recursively for all **<exes_var>** executable files and accumulates them into a variable.

get_item_key

```
get_item_key(<item> <key_var>)
```

Given **<item>** file name, generate **<key_var>** key that should be unique considering the set of libraries that need copying or fixing up to make a bundle standalone. This is essentially the file name including extension with **.** replaced by **_**

This key is used as a prefix for CMake variables so that we can associate a set of variables with a given item based on its key.

get_item_rpaths

```
get_item_rpaths(<item> <rpaths_var>)
```

Get RPATHS of the **<item>** file name and store them to the variable with provided name **<rpaths_var>**.

clear_bundle_keys

```
clear_bundle_keys(<keys_var>)
```

Loop over the `<keys_var>` list of keys, clearing all the variables associated with each key. After the loop, clear the list of keys itself.

Caller of `get_bundle_keys` should call `clear_bundle_keys` when done with list of keys.

`set_bundle_key_values`

```
set_bundle_key_values(<keys_var> <context> <item> <exepath> <dirs>
                    <copyflag> [<rpaths>])
```

Add `<keys_var>` key to the list (if necessary) for the given item. If added, also set all the variables associated with that key.

`get_bundle_keys`

```
get_bundle_keys(<app> <libs> <dirs> <keys_var> [IGNORE_ITEM <file>...])
```

Loop over all the executable and library files within `<app>` bundle (and given as extra `<libs>`) and accumulate a list of keys representing them. Set values associated with each key such that we can loop over all of them and copy prerequisite libs into the bundle and then do appropriate `install_name_tool` fixups.

Added in version 3.6: As an optional parameter (**IGNORE_ITEM**) a list of file names can be passed, which are then ignored (e.g. **IGNORE_ITEM "vcredist_x86.exe;vcredist_x64.exe"**)

`copy_resolved_item_into_bundle`

```
copy_resolved_item_into_bundle(<resolved_item> <resolved_embedded_item>)
```

Copy a resolved item into the bundle if necessary. Copy is not necessary, if the `<resolved_item>` is "the same as" the `<resolved_embedded_item>`.

`copy_resolved_framework_into_bundle`

```
copy_resolved_framework_into_bundle(<resolved_item> <resolved_embedded_item>)
```

Copy a resolved framework into the bundle if necessary. Copy is not necessary, if the `<resolved_item>` is "the same as" the `<resolved_embedded_item>`.

By default, **BU_COPY_FULL_FRAMEWORK_CONTENTS** is not set. If you want full frameworks embedded in your bundles, set **BU_COPY_FULL_FRAMEWORK_CONTENTS** to **ON** before calling `fixup_bundle`. By default, **COPY_RESOLVED_FRAMEWORK_INTO_BUNDLE** copies the framework dylib itself plus the framework **Resources** directory.

`fixup_bundle_item`

```
fixup_bundle_item(<resolved_embedded_item> <exepath> <dirs>)
```

Get the direct/non-system prerequisites of the `<resolved_embedded_item>`. For each prerequisite, change the way it is referenced to the value of the `_EMBEDDED_ITEM` keyed variable for that prerequisite. (Most likely changing to an `@executable_path` style reference.)

This function requires that the `<resolved_embedded_item>` be **inside** the bundle already. In other words, if you pass plugins to `fixup_bundle` as the `libs` parameter, you should install them or copy them into the bundle before calling `fixup_bundle`. The `libs` parameter is a list of libraries that

must be fixed up, but that cannot be determined by otool output analysis. (i.e., **plugins**)

Also, change the id of the item being fixed up to its own **_EMBEDDED_ITEM** value.

Accumulate changes in a local variable and make *one* call to **install_name_tool** at the end of the function with all the changes at once.

If the **BU_CHMOD_BUNDLE_ITEMS** variable is set then bundle items will be marked writable before **install_name_tool** tries to change them.

verify_bundle_prerequisites

```
verify_bundle_prerequisites(<bundle> <result_var> <info_var>
                           [IGNORE_ITEM <file>...])
```

Verifies that the sum of all prerequisites of all files inside the bundle are contained within the bundle or are **system** libraries, presumed to exist everywhere.

Added in version 3.6: As an optional parameter (**IGNORE_ITEM**) a list of file names can be passed, which are then ignored (e.g. **IGNORE_ITEM "vcredist_x86.exe;vcredist_x64.exe"**)

verify_bundle_symlinks

```
verify_bundle_symlinks(<bundle> <result_var> <info_var>)
```

Verifies that any symlinks found in the **<bundle>** bundle point to other files that are already also in the bundle... Anything that points to an external file causes this function to fail the verification.

CheckCCompilerFlag

Check once whether the C compiler supports a given flag.

check_c_compiler_flag

```
check_c_compiler_flag(<flag> <resultVar>)
```

Check once that the **<flag>** is accepted by the compiler without a diagnostic. The result is stored in the internal cache variable specified by **<resultVar>**, with boolean **true** for success and boolean **false** for failure.

true indicates only that the compiler did not issue a diagnostic message when given the flag. Whether the flag has any effect is beyond the scope of this module.

Internally, *try_compile()* is used to perform the check. If **CMAKE_TRY_COMPILE_TARGET_TYPE** is set to **EXECUTABLE** (default), the check compiles and links an executable program. If set to **STATIC_LIBRARY**, the check is compiled but not linked.

See also *check_compiler_flag()* for a more general command syntax.

The compile and link commands can be influenced by setting any of the following variables prior to calling **check_c_compiler_flag()**. Unknown flags in these variables can cause a false negative result.

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited—**--a ;-list** will not work. The contents of **CMAKE_<LANG>_FLAGS** and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A ;*-list* of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;*-list* of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the *INCLUDE_DIRECTORIES* directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;*-list* of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;*-list* of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;*-list* of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckCompilerFlag

Added in version 3.19.

Check once whether the **<lang>** compiler supports a given flag.

check_compiler_flag

```
check_compiler_flag(<lang> <flag> <resultVar>)
```

Check once that the **<flag>** is accepted by the **<lang>** compiler without a diagnostic. The result is stored in the internal cache variable specified by **<resultVar>**, with boolean **true** for success and boolean **false** for failure.

true indicates only that the compiler did not issue a diagnostic message when given the flag. Whether the flag has any effect is beyond the scope of this module.

Internally, *try_compile()* is used to perform the check. If *CMAKE_TRY_COMPILE_TARGET_TYPE* is set to **EXECUTABLE** (default), the check compiles and links an executable program. If set to **STATIC_LIBRARY**, the check is compiled but not linked.

The compile and link commands can be influenced by setting any of the following variables prior to calling **check_compiler_flag()**. Unknown flags in these variables can case a false negative result.

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;*-list* will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated

configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;-list of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the **INCLUDE_DIRECTORIES** directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckCSourceCompiles

Check once if C source code can be built.

check_c_source_compiles

```
check_c_source_compiles(<code> <resultVar>
                        [FAIL_REGEX <regex1> [<regex2>...]])
```

Check once that the source supplied in **<code>** can be built. The result is stored in the internal cache variable specified by **<resultVar>**, with boolean **true** for success and boolean **false** for failure.

If **FAIL_REGEX** is provided, then failure is determined by checking if anything in the compiler output matches any of the specified regular expressions.

Internally, *try_compile()* is used to compile the source. If **CMAKE_TRY_COMPILE_TARGET_TYPE** is set to **EXECUTABLE** (default), the source is compiled and linked as an executable program. If set to **STATIC_LIBRARY**, the source is compiled but not linked. In any case, all functions must be declared as usual.

See also *check_source_compiles()* for a more general command syntax.

See also *check_source_runs()* to run compiled source.

The compile and link commands can be influenced by setting any of the following variables prior to calling **check_c_source_compiles()**:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a *;-list* will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A *;-list* of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A *;-list* of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the *INCLUDE_DIRECTORIES* directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A *;-list* of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A *;-list* of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A *;-list* of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckCSourceRuns

Check once if given C source compiles and links into an executable and can subsequently be run.

check_c_source_runs

```
check_c_source_runs(<code> <resultVar>)
```

Check once that the source supplied in **<code>** can be built, linked as an executable, and then run. The **<code>** must contain at least a **main()** function.

The result is stored in the internal cache variable specified by **<resultVar>**. If the code builds and runs with exit code **0**, success is indicated by boolean **true**. Failure to build or run is indicated by boolean **false**, such as an empty string or an error message.

See also *check_source_runs()* for a more general command syntax.

The compile and link commands can be influenced by setting any of the following variables prior to calling **check_c_source_runs()**:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;-list of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the *INCLUDE_DIRECTORIES* directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckCXXCompilerFlag

Check once whether the CXX compiler supports a given flag.

check_cxx_compiler_flag

```
check_cxx_compiler_flag(<flag> <resultVar>)
```

Check once that the **<flag>** is accepted by the compiler without a diagnostic. The result is stored in the internal cache variable specified by **<resultVar>**, with boolean **true** for success and boolean **false** for failure.

true indicates only that the compiler did not issue a diagnostic message when given the flag. Whether the flag has any effect is beyond the scope of this module.

Internally, *try_compile()* is used to perform the check. If *CMAKE_TRY_COMPILE_TARGET_TYPE* is set to **EXECUTABLE** (default), the check compiles and links an executable program. If set to **STATIC_LIBRARY**, the check is compiled but not linked.

See also *check_compiler_flag()* for a more general command syntax.

The compile and link commands can be influenced by setting any of the following variables prior to calling **check_cxx_compiler_flag()**. Unknown flags in these variables can cause a false negative result.

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;-list of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckCXXSourceCompiles

Check once if C++ source code can be built.

check_cxx_source_compiles

```
check_cxx_source_compiles(<code> <resultVar>
                           [FAIL_REGEX <regex1> [<regex2>...]])
```

Check once that the source supplied in `<code>` can be built. The result is stored in the internal cache variable specified by `<resultVar>`, with boolean **true** for success and boolean **false** for failure.

If **FAIL_REGEX** is provided, then failure is determined by checking if anything in the compiler output matches any of the specified regular expressions.

Internally, `try_compile()` is used to compile the source. If `CMAKE_TRY_COMPILE_TARGET_TYPE` is set to **EXECUTABLE** (default), the source is compiled and linked as an executable program. If set to **STATIC_LIBRARY**, the source is compiled but not linked. In any case, all functions must be declared as usual.

See also `check_source_compiles()` for a more general command syntax.

See also `check_source_runs()` to run compiled source.

The compile and link commands can be influenced by setting any of the following variables prior to calling `check_cxx_source_compiles()`:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a `;-list` will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A `;-list` of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A `;-list` of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A `;-list` of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A `;-list` of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A `;-list` of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckCXXSourceRuns

Check once if given C++ source compiles and links into an executable and can subsequently be run.

check_cxx_source_runs

```
check_cxx_source_runs(<code> <resultVar>)
```

Check once that the source supplied in `<code>` can be built, linked as an executable, and then run. The `<code>` must contain at least a `main()` function.

The result is stored in the internal cache variable specified by `<resultVar>`. If the code builds and runs with exit code `0`, success is indicated by boolean `true`. Failure to build or run is indicated by boolean `false`, such as an empty string or an error message.

See also `check_source_runs()` for a more general command syntax.

The compile and link commands can be influenced by setting any of the following variables prior

to calling `check_cxx_source_runs()`:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a `;-list` will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A `;-list` of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A `;-list` of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A `;-list` of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A `;-list` of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A `;-list` of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckCXXSymbolExists

Check if a symbol exists as a function, variable, or macro in C++.

check_cxx_symbol_exists

```
check_cxx_symbol_exists(<symbol> <files> <variable>)
```

Check that the `<symbol>` is available after including given header `<files>` and store the result in a `<variable>`. Specify the list of files in one argument as a semicolon-separated list. `check_cxx_symbol_exists()` can be used to check for symbols as seen by the C++ compiler, as opposed to `check_symbol_exists()`, which always uses the C compiler.

If the header files define the symbol as a macro it is considered available and assumed to work. If the header files declare the symbol as a function or variable then the symbol must also be available for linking. If the symbol is a type, enum value, or C++ template it will not be recognized: consider using the `CheckTypeSize` or `CheckSourceCompiles` module instead.

NOTE:

This command is unreliable when `<symbol>` is (potentially) an overloaded function. Since there is no reliable way to predict whether a given function in the system environment may be defined as an

overloaded function or may be an overloaded function on other systems or will become so in the future, it is generally advised to use the *CheckSourceCompiles* module for checking any function symbol (unless somehow you surely know the checked function is not overloaded on other systems or will not be so in the future).

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated configuration-specific variable are automatically added to the compiler command before the contents of *CMAKE_REQUIRED_FLAGS*.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by *<resultVar>* will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;-list of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the *INCLUDE_DIRECTORIES* directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

For example:

```
include(CheckCXXSymbolExists)

# Check for macro SEEK_SET
check_cxx_symbol_exists(SEEK_SET "cstdio" HAVE_SEEK_SET)
# Check for function std::fopen
check_cxx_symbol_exists(std::fopen "cstdio" HAVE_STD_FOPEN)
```

CheckFortranCompilerFlag

Added in version 3.3.

Check once whether the Fortran compiler supports a given flag.

check_fortran_compiler_flag

```
check_fortran_compiler_flag(<flag> <resultVar>)
```

Check once that the **<flag>** is accepted by the compiler without a diagnostic. The result is stored in the internal cache variable specified by **<resultVar>**, with boolean **true** for success and boolean **false** for failure.

true indicates only that the compiler did not issue a diagnostic message when given the flag. Whether the flag has any effect is beyond the scope of this module.

Internally, *try_compile()* is used to perform the check. If *CMAKE_TRY_COMPILE_TARGET_TYPE* is set to **EXECUTABLE** (default), the check compiles and links an executable program. If set to **STATIC_LIBRARY**, the check is compiled but not linked.

See also *check_compiler_flag()* for a more general command syntax.

The compile and link commands can be influenced by setting any of the following variables prior to calling **check_fortran_compiler_flag()**. Unknown flags in these variables can cause a false negative result.

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a *;-list* will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A *;-list* of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A *;-list* of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the *INCLUDE_DIRECTORIES* directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A *;-list* of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A *;-list* of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A *;-list* of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckFortranFunctionExists

Check if a Fortran function exists.

check_fortran_function_exists

```
check_fortran_function_exists(<function> <result>)
```

where

<function>

the name of the Fortran function

<result>

variable to store the result; will be created as an internal cache variable.

NOTE:

This command does not detect functions in Fortran modules. In general it is recommended to use *CheckSourceCompiles* instead to determine if a Fortran function or subroutine is available.

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CheckFortranSourceCompiles

Added in version 3.1.

Check once if Fortran source code can be built.

check_fortran_source_compiles

```
check_fortran_source_compiles(<code> <resultVar>
    [FAIL_REGEX <regex>...]
    [SRC_EXT <extension>]
)
```

Check once that the source supplied in **<code>** can be built. The result is stored in the internal cache variable specified by **<resultVar>**, with boolean **true** for success and boolean **false** for failure.

If **FAIL_REGEX** is provided, then failure is determined by checking if anything in the compiler output matches any of the specified regular expressions.

By default, the test source file will be given a **.F** file extension. The **SRC_EXT** option can be used to override this with **<extension>** instead-- **.F90** is a typical choice.

See also `check_source_compiles()` for a more general command syntax.

See also `check_source_runs()` to run compiled source.

Internally, `try_compile()` is used to compile the source. If `CMAKE_TRY_COMPILE_TARGET_TYPE` is set to **EXECUTABLE** (default), the source is compiled and linked as an executable program. If set to **STATIC_LIBRARY**, the source is compiled but not linked. In any case, all functions must be declared as usual.

The compile and link commands can be influenced by setting any of the following variables prior to calling `check_fortran_source_compiles()`:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a `;-list` will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A `;-list` of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A `;-list` of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A `;-list` of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A `;-list` of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A `;-list` of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckFortranSourceRuns

Added in version 3.14.

Check once if given Fortran source compiles and links into an executable and can subsequently be run.

check_fortran_source_runs

```
check_fortran_source_runs(<code> <resultVar>
    [SRC_EXT <extension>])
```

Check once that the source supplied in `<code>` can be built, linked as an executable, and then run. The `<code>` must contain a Fortran **program**.

The result is stored in the internal cache variable specified by `<resultVar>`. If the code builds and runs with exit code **0**, success is indicated by boolean **true**. Failure to build or run is indicated by boolean **false**, such as an empty string or an error message.

```
check_fortran_source_runs("program test
real :: x[*]
call co_sum(x)
end program"
HAVE_COARRAY)
```

By default, the test source file will be given a **.F90** file extension. The **SRC_EXT** option can be used to override this with `<extension>` instead.

See also `check_source_runs()` for a more general command syntax.

The compile and link commands can be influenced by setting any of the following variables prior to calling `check_fortran_source_runs()`:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a `;-list` will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A `;-list` of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A `;-list` of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A `;-list` of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A `;-list` of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A `;-list` of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckFunctionExists

Check once if a C function can be linked from system libraries.

check_function_exists

```
check_function_exists(<function> <variable>)
```

Checks that the **<function>** is provided by libraries on the system and store the result in internal cache variable **<variable>**.

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;-list of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the *INCLUDE_DIRECTORIES* directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

NOTE:

Prefer using *CheckSymbolExists* or *CheckSourceCompiles* instead of this module, for the following reasons:

- **check_function_exists()** can't detect functions that are inlined in headers or specified as a macro.
- **check_function_exists()** can't detect anything in the 32-bit versions of the Win32 API, because of a mismatch in calling conventions.

- **check_function_exists()** only verifies linking, it does not verify that the function is declared in system headers.

CheckIncludeFileCXX

Provides a macro to check if a header file can be included in **CXX**.

check_include_file_cxx

```
check_include_file_cxx(<include> <variable> [<flags>])
```

Check if the given **<include>** file may be included in a **CXX** source file and store the result in an internal cache entry named **<variable>**. The optional third argument may be used to add compilation flags to the check (or use **CMAKE_REQUIRED_FLAGS** below).

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a **;-list** will not work. The contents of **CMAKE_<LANG>_FLAGS** and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A **;-list** of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A **;-list** of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the **INCLUDE_DIRECTORIES** directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A **;-list** of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A **;-list** of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A **;-list** of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

See modules *CheckIncludeFile* and *CheckIncludeFiles* to check for one or more **C** headers.

CheckIncludeFile

Provides a macro to check if a header file can be included in **C**.

check_include_file

```
check_include_file(<include> <variable> [<flags>])
```

Check if the given **<include>** file may be included in a C source file and store the result in an internal cache entry named **<variable>**. The optional third argument may be used to add compilation flags to the check (or use **CMAKE_REQUIRED_FLAGS** below).

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of **CMAKE_<LANG>_FLAGS** and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;-list of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the **INCLUDE_DIRECTORIES** directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

See the *CheckIncludeFiles* module to check for multiple headers at once. See the *CheckIncludeFileCXX* module to check for headers using the **CXX** language.

CheckIncludeFiles

Provides a macro to check if a list of one or more header files can be included together.

check_include_files

```
check_include_files("<includes>" <variable> [LANGUAGE <language>])
```

Check if the given **<includes>** list may be included together in a source file and store the result in an internal cache entry named **<variable>**. Specify the **<includes>** argument as a ;-list of header

file names.

If **LANGUAGE** is set, the specified compiler will be used to perform the check. Acceptable values are **C** and **CXX**. If not set, the C compiler will be used if enabled. If the C compiler is not enabled, the C++ compiler will be used if enabled.

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;-list of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the *INCLUDE_DIRECTORIES* directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

See modules *CheckIncludeFile* and *CheckIncludeFileCXX* to check for a single header file in **C** or **CXX** languages.

CheckIPOSupported

Added in version 3.9.

Check whether the compiler supports an interprocedural optimization (IPO/LTO). Use this before enabling the *INTERPROCEDURAL_OPTIMIZATION* target property.

check_ipo_supported

```
check_ipo_supported([RESULT <result>] [OUTPUT <output>]
                    [LANGUAGES <lang>...])
```

Options are:

RESULT *<result>*

Set *<result>* variable to **YES** if IPO is supported by the compiler and **NO** otherwise. If this option is not given then the command will issue a fatal error if IPO is not supported.

OUTPUT *<output>*

Set *<output>* variable with details about any error.

LANGUAGES *<lang>...*

Specify languages whose compilers to check.

The following languages are supported:

- **C**
- **CXX**
- **CUDA**

Added in version 3.25.

- **Fortran**

If this option is not given, the default languages are picked from the current *ENABLED_LANGUAGES* global property.

NOTE:

To use **check_ipo_supported()**, policy *CMP0069* must be set to **NEW**; otherwise, a fatal error will occur.

Added in version 3.13: Support for Visual Studio generators.

Added in version 3.24: The check uses the caller's *CMAKE_<LANG>_FLAGS* and *CMAKE_<LANG>_FLAGS_<CONFIG>* values. See policy *CMP0138*.

Examples

```
check_ipo_supported() # fatal error if IPO is not supported
set_property(TARGET foo PROPERTY INTERPROCEDURAL_OPTIMIZATION TRUE)

# Optional IPO. Do not use IPO if it's not supported by compiler.
check_ipo_supported(RESULT result OUTPUT output)
if(result)
    set_property(TARGET foo PROPERTY INTERPROCEDURAL_OPTIMIZATION TRUE)
else()
    message(WARNING "IPO is not supported: ${output}")
endif()
```

CheckLanguage

Check whether a language can be enabled by the *enable_language()* or *project()* commands:

check_language

```
check_language(<lang>)
```

Try enabling language *<lang>* in a test project and record results in the cache:

CMAKE_<LANG>_COMPILER

If the language can be enabled, this variable is set to the compiler that was found. If the language cannot be enabled, this variable is set to **NOTFOUND**.

If this variable is already set, either explicitly or cached by a previous call, the check is skipped.

CMAKE_<LANG>_HOST_COMPILER

This variable is set when **<lang>** is **CUDA** or **HIP**.

If the check detects an explicit host compiler that is required for compilation, this variable will be set to that compiler. If the check detects that no explicit host compiler is needed, this variable will be cleared.

If this variable is already set, its value is preserved only if **CMAKE_<LANG>_COMPILER** is also set. Otherwise, the check runs and overwrites **CMAKE_<LANG>_HOST_COMPILER** with a new result. Note that **CMAKE_<LANG>_HOST_COMPILER** documents it should not be set without also setting **CMAKE_<LANG>_COMPILER** to a NVCC compiler.

CMAKE_<LANG>_PLATFORM

This variable is set to the detected GPU platform when **<lang>** is **HIP**.

If the variable is already set its value is always preserved. Only compatible values will be considered for **CMAKE_<LANG>_COMPILER**.

For example:

```
check_language(Fortran)
if(CMAKE_Fortran_COMPILER)
  enable_language(Fortran)
else()
  message(STATUS "No Fortran support")
endif()
```

CheckLibraryExists

Check once if the function exists in system or specified library.

check_library_exists

```
check_library_exists(<library> <function> <location> <variable>)
```

Check that the library **<library>** exists in the given location **<location>** and has the specified **<function>**. The result is stored in an internal cache variable **<variable>**. If **<location>** is empty string, default directories are searched.

Prefer using *CheckSymbolExists* or *CheckSourceCompiles* instead of this module for more robust detection if a function is available in a library.

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of **CMAKE_<LANG>_FLAGS** and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A ;–list of compiler definitions of the form **–DFOO** or **–DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;–list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;–list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;–list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

Examples

This module can be useful for performing so-called sanity checks to verify that the specified library provides the expected functionality and is indeed the correct one being located.

For example, to check if the **curl** library exists in the default paths and has the **curl_easy_perform** function:

```
include(CheckLibraryExists)
check_library_exists(curl curl_easy_perform "" HAVE_LIBRARY_CURL)
```

To check if library exists in specific non-standard location and has a specified function:

```
include(CheckLibraryExists)
check_library_exists(curl curl_easy_perform "/opt/curl/lib" HAVE_LIBRARY_CURL)
```

Also *IMPORTED library* (for example, from the **find_package()** call) can be used:

```
find_package(CURL)

# ...

if(TARGET CURL::libcurl)
    include(CheckLibraryExists)
    check_library_exists(CURL::libcurl curl_easy_perform "" HAVE_LIBRARY_CURL)
endif()
```

CheckLinkerFlag

Added in version 3.18.

Check whether the compiler supports a given link flag.

check_linker_flag

```
check_linker_flag(<lang> <flag> <var>)
```

Check that the link **<flag>** is accepted by the **<lang>** compiler without a diagnostic. Stores the result in an internal cache entry named **<var>**.

This command temporarily sets the **CMAKE_REQUIRED_LINK_OPTIONS** variable and calls the *check_source_compiles()* command from the *CheckSourceCompiles* module. See that module's documentation for a listing of variables that can otherwise modify the build.

The underlying implementation relies on the *LINK_OPTIONS* property to check the specified flag. The **LINKER:** prefix, as described in the *target_link_options()* command, can be used as well.

A positive result from this check indicates only that the compiler did not issue a diagnostic message when given the link flag. Whether the flag has any effect or even a specific one is beyond the scope of this module.

NOTE:

Since the *try_compile()* command forwards flags from variables like *CMAKE_<LANG>_FLAGS*, unknown flags in such variables may cause a false negative for this check.

CheckOBJCCompilerFlag

Added in version 3.16.

Check once whether the Objective-C compiler supports a given flag.

check_objc_compiler_flag

```
check_objc_compiler_flag(<flag> <resultVar>)
```

Check once that the **<flag>** is accepted by the compiler without a diagnostic. The result is stored in the internal cache variable specified by **<resultVar>**, with boolean **true** for success and boolean **false** for failure.

true indicates only that the compiler did not issue a diagnostic message when given the flag. Whether the flag has any effect is beyond the scope of this module.

Internally, *try_compile()* is used to perform the check. If *CMAKE_TRY_COMPILE_TARGET_TYPE* is set to **EXECUTABLE** (default), the check compiles and links an executable program. If set to **STATIC_LIBRARY**, the check is compiled but not linked.

See also *check_compiler_flag()* for a more general command syntax.

The compile and link commands can be influenced by setting any of the following variables prior to calling **check_objc_compiler_flag()**. Unknown flags in these variables can case a false negative result.

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a *;-list* will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A *;-list* of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;–list of header search paths to pass to the compiler. These will be the only header search paths used—the contents of the *INCLUDE_DIRECTORIES* directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;–list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;–list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;–list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckOBJCSourceCompiles

Added in version 3.16.

Check once if Objective-C source can be built.

check_objc_source_compiles

```
check_objc_source_compiles(<code> <resultVar>
                           [FAIL_REGEX <regex1> [<regex2>...]])
```

Check once that the source supplied in **<code>** can be built. The result is stored in the internal cache variable specified by **<resultVar>**, with boolean **true** for success and boolean **false** for failure.

If **FAIL_REGEX** is provided, then failure is determined by checking if anything in the compiler output matches any of the specified regular expressions.

Internally, *try_compile()* is used to compile the source. If *CMAKE_TRY_COMPILE_TARGET_TYPE* is set to **EXECUTABLE** (default), the source is compiled and linked as an executable program. If set to **STATIC_LIBRARY**, the source is compiled but not linked. In any case, all functions must be declared as usual.

See also *check_source_compiles()* for a more general command syntax.

See also *check_source_runs()* to run compiled source.

The compile and link commands can be influenced by setting any of the following variables prior to calling **check_objc_source_compiles()**

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited—`;-a` will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A `;-list` of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A `;-list` of header search paths to pass to the compiler. These will be the only header search paths used—the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A `;-list` of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A `;-list` of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A `;-list` of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckOBJCSourceRuns

Added in version 3.16.

Check once if given Objective-C source compiles and links into an executable and can subsequently be run.

check_objc_source_runs

```
check_objc_source_runs(<code> <resultVar>)
```

Check once that the source supplied in `<code>` can be built, linked as an executable, and then run. The `<code>` must contain at least a `main()` function.

The result is stored in the internal cache variable specified by `<resultVar>`. If the code builds and runs with exit code `0`, success is indicated by boolean `true`. Failure to build or run is indicated by boolean `false`, such as an empty string or an error message.

See also `check_source_runs()` for a more general command syntax.

The compile and link commands can be influenced by setting any of the following variables prior

to calling `check_objc_source_runs()`

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a `;-list` will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A `;-list` of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A `;-list` of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A `;-list` of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A `;-list` of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A `;-list` of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckOBJCXXCompilerFlag

Added in version 3.16.

Check once whether the Objective-C++ compiler supports a given flag.

check_objcxx_compiler_flag

```
check_objcxx_compiler_flag(<flag> <resultVar>)
```

Check once that the `<flag>` is accepted by the compiler without a diagnostic. The result is stored in the internal cache variable specified by `<resultVar>`, with boolean **true** for success and boolean **false** for failure.

true indicates only that the compiler did not issue a diagnostic message when given the flag. Whether the flag has any effect is beyond the scope of this module.

Internally, `try_compile()` is used to perform the check. If `CMAKE_TRY_COMPILE_TARGET_TYPE` is set to **EXECUTABLE** (default), the check compiles and links an executable program. If set to **STATIC_LIBRARY**, the check is compiled but not linked.

See also `check_compiler_flag()` for a more general command syntax.

The compile and link commands can be influenced by setting any of the following variables prior to calling `check_objcxx_compiler_flag()`. Unknown flags in these variables can cause a false negative result.

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a `;-list` will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A `;-list` of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A `;-list` of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A `;-list` of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A `;-list` of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A `;-list` of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckOBJCXXSourceCompiles

Added in version 3.16.

Check once if Objective-C++ source can be built.

check_objcxx_source_compiles

```
check_objcxx_source_compiles(<code> <resultVar>
                             [FAIL_REGEX <regex1> [<regex2>...]])
```

Check once that the source supplied in `<code>` can be built. The result is stored in the internal cache variable specified by `<resultVar>`, with boolean **true** for success and boolean **false** for failure.

If **FAIL_REGEX** is provided, then failure is determined by checking if anything in the compiler output matches any of the specified regular expressions.

Internally, `try_compile()` is used to compile the source. If `CMAKE_TRY_COMPILE_TARGET_TYPE` is set to **EXECUTABLE** (default), the source is compiled and linked as an executable program. If set to **STATIC_LIBRARY**, the source is compiled but not linked. In any case, all functions must be declared as usual.

See also `check_source_compiles()` for a more general command syntax.

See also `check_source_runs()` to run compiled source.

The compile and link commands can be influenced by setting any of the following variables prior to calling `check_objcxx_source_compiles()`

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a `;-list` will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A `;-list` of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A `;-list` of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A `;-list` of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A `;-list` of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A `;-list` of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckOBJCXXSourceRuns

Added in version 3.16.

Check once if given Objective-C++ source compiles and links into an executable and can subsequently be run.

check_objcxx_source_runs

```
check_objcxx_source_runs(<code> <resultVar>)
```

Check once that the source supplied in `<code>` can be built, linked as an executable, and then run. The `<code>` must contain at least a `main()` function.

The result is stored in the internal cache variable specified by `<resultVar>`. If the code builds and runs with exit code `0`, success is indicated by boolean `true`. Failure to build or run is indicated by boolean `false`, such as an empty string or an error message.

See also `check_source_runs()` for a more general command syntax.

The compile and link commands can be influenced by setting any of the following variables prior to calling `check_objcxx_source_runs()`

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a `;-list` will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A `;-list` of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A `;-list` of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A `;-list` of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A `;-list` of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A `;-list` of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckPIESupported

Added in version 3.14.

Check whether the linker supports Position Independent Code (PIE) or No Position Independent Code (NO_PIE) for executables. Use this to ensure that the `POSITION_INDEPENDENT_CODE` target property for executables will be honored at link time.

check_pie_supported

```
check_pie_supported([OUTPUT_VARIABLE <output>]
                   [LANGUAGES <lang>...])
```

Options are:

OUTPUT_VARIABLE <output>

Set <output> variable with details about any error. If the check is bypassed because it uses cached results from a previous call, the output will be empty even if errors were present in the previous call.

LANGUAGES <lang>...

Check the linkers used for each of the specified languages. If this option is not provided, the command checks all enabled languages.

C, **CXX**, **Fortran** are supported.

Added in version 3.23: **OBJC**, **OBJCXX**, **CUDA**, and **HIP** are supported.

It makes no sense to use this module when *CMP0083* is set to **OLD**, so the command will return an error in this case. See policy *CMP0083* for details.

Variables

For each language checked, two boolean cache variables are defined.

CMAKE_<lang>_LINK_PIE_SUPPORTED

Set to true if **PIE** is supported by the linker and false otherwise.

CMAKE_<lang>_LINK_NO_PIE_SUPPORTED

Set to true if **NO_PIE** is supported by the linker and false otherwise.

Examples

```
check_pie_supported()
set_property(TARGET foo PROPERTY POSITION_INDEPENDENT_CODE TRUE)

# Retrieve any error message.
check_pie_supported(OUTPUT_VARIABLE output LANGUAGES C)
set_property(TARGET foo PROPERTY POSITION_INDEPENDENT_CODE TRUE)
if(NOT CMAKE_C_LINK_PIE_SUPPORTED)
    message(WARNING "PIE is not supported at link time: ${output}.\n"
            "PIE link options will not be passed to linker.")
endif()
```

CheckPrototypeDefinition

Check if a C function has the expected prototype.

check_prototype_definition

```
check_prototype_definition(<function> <prototype> <return> <headers> <variables>)
```

<function>

The name of the function whose prototype is being checked.

<prototype>

The expected prototype of the function, provided as a string.

<return>

The return value of the function. This will be used as a return value in the function definition body of the generated test program to verify that the function's return type matches

the expected type.

<headers>

A *semicolon-separated* list of header file names required for checking the function prototype.

<variable>

The name of the variable to store the check result. This variable will be created as an internal cache variable.

This command generates a test program and verifies that it builds without errors. The generated test program includes specified **<headers>**, defines a function with given literal **<prototype>** and **<return>** value and then uses the specified **<function>**. The simplified test program can be illustrated as:

```
#include <headers>
// ...
<prototype> { return <return>; }
int main(...) { ...<function>()... }
```

The following variables may be set before calling this function to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a *;-list* will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A *;-list* of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A *;-list* of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A *;-list* of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A *;-list* of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A *;-list* of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

Examples

Checking if the **getpwent_r()** function on Solaris/illumos systems has the expected prototype:

```
include(CheckPrototypeDefinition)

check_prototype_definition(
  getpwent_r
  "struct passwd *getpwent_r(struct passwd *src, char *buf, int buflen)"
  "NULL"
  "unistd.h;pwd.h"
  HAVE_SOLARIS_GETPWENT_R
)
```

CheckSourceCompiles

Added in version 3.19.

Check once if source code can be built for a given language.

check_source_compiles

```
check_source_compiles(<lang> <code> <resultVar>
                     [FAIL_REGEX <regex1> [<regex2>...]]
                     [SRC_EXT <extension>])
```

Check once that the source supplied in **<code>** can be built for code language **<lang>**. The result is stored in the internal cache variable specified by **<resultVar>**, with boolean **true** for success and boolean **false** for failure.

If **FAIL_REGEX** is provided, then failure is determined by checking if anything in the compiler output matches any of the specified regular expressions.

By default, the test source file will be given a file extension that matches the requested language. The **SRC_EXT** option can be used to override this with **.<extension>** instead.

The C example checks if the compiler supports the **noreturn** attribute:

```
set(CMAKE_TRY_COMPILE_TARGET_TYPE "STATIC_LIBRARY")

check_source_compiles(C
  "#if !__has_c_attribute(noreturn)
  #error \"No noreturn attribute\"
  #endif"
  HAVE_NORETURN)
```

The Fortran example checks if the compiler supports the **pure** procedure attribute:

```
set(CMAKE_TRY_COMPILE_TARGET_TYPE "STATIC_LIBRARY")

check_source_compiles(Fortran
  "pure subroutine foo()
  end subroutine"
  HAVE_PURE)
```

Internally, `try_compile()` is used to compile the source. If `CMAKE_TRY_COMPILE_TARGET_TYPE` is set to **EXECUTABLE** (default), the source is

compiled and linked as an executable program. If set to **STATIC_LIBRARY**, the source is compiled but not linked. In any case, all functions must be declared as usual.

See also `check_source_runs()` to run compiled source.

The compile and link commands can be influenced by setting any of the following variables prior to calling `check_source_compiles()`:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;-list of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckSourceRuns

Added in version 3.19.

Check if given source compiles and links into an executable and can subsequently be run.

check_source_runs

```
check_source_runs(<lang> <code> <resultVar>
                  [SRC_EXT <extension>])
```

Check once that the `<lang>` source supplied in `<code>` can be built, linked as an executable, and then run. The `<code>` must contain at least a `main()` function, or in Fortran a `program`.

The result is stored in the internal cache variable specified by `<resultVar>`. If the code builds and runs with exit code `0`, success is indicated by boolean `true`. Failure to build or run is indicated by boolean `false`, such as an empty string or an error message.

By default, the test source file will be given a file extension that matches the requested language. The `SRC_EXT` option can be used to override this with `.<extension>` instead.

The `<code>` must contain a valid main program. For example:

```
check_source_runs(C
"#include <stdlib.h>
#include <stdnoreturn.h>
noreturn void f(){ exit(0); }
int main(void) { f(); return 1; }"
HAVE_NORETURN)

check_source_runs(Fortran
"program test
real :: x[*]
call co_sum(x)
end program"
HAVE_COARRAY)
```

The compile and link commands can be influenced by setting any of the following variables prior to calling `check_source_runs()`

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a `;-list` will not work. The contents of `CMAKE_<LANG>_FLAGS` and its associated configuration-specific variable are automatically added to the compiler command before the contents of `CMAKE_REQUIRED_FLAGS`.

CMAKE_REQUIRED_DEFINITIONS

A `;-list` of compiler definitions of the form `-DFOO` or `-DFOO=bar`. A definition for the name specified by `<resultVar>` will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A `;-list` of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the `INCLUDE_DIRECTORIES` directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A `;-list` of options to add to the link command (see `try_compile()` for further details).

CMAKE_REQUIRED_LIBRARIES

A `;-list` of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see `try_compile()` for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A `;-list` of libraries search paths to pass to the linker (see `try_compile()` for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CheckStructHasMember

Check if the given struct or class has the specified member variable

check_struct_has_member

```
check_struct_has_member(<struct> <member> <headers> <variable>
                        [LANGUAGE <language>])
```

Check that the struct or class **<struct>** has the specified **<member>** after including the given header(s) **<headers>** where the prototype should be declared. Specify the list of header files in one argument as a semicolon-separated list. The result is stored in an internal cache variable **<variable>**.

The options are:

LANGUAGE <language>

Use the **<language>** compiler to perform the check. Acceptable values are **C** and **CXX**. If not specified, it defaults to **C**.

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of **CMAKE_<LANG>_FLAGS** and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;-list of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the **INCLUDE_DIRECTORIES** directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

Example

```
include(CheckStructHasMember)

check_struct_has_member("struct timeval" tv_sec sys/select.h
                        HAVE_TIMEVAL_TV_SEC LANGUAGE C)
```

CheckSymbolExists

Provides a macro to check if a symbol exists as a function, variable, or macro in C.

check_symbol_exists

```
check_symbol_exists(<symbol> <files> <variable>)
```

Check that the **<symbol>** is available after including given header **<files>** and store the result in a **<variable>**. Specify the list of files in one argument as a semicolon-separated list. **<variable>** will be created as an internal cache variable.

If the header files define the symbol as a macro it is considered available and assumed to work. If the header files declare the symbol as a function or variable then the symbol must also be available for linking (so intrinsics may not be detected). If the symbol is a type, enum value, or intrinsic it will not be recognized (consider using *CheckTypeSize* or *CheckSourceCompiles*). If the check needs to be done in C++, consider using *CheckCXXSymbolExists* instead.

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a *;-list* will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A *;-list* of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A *;-list* of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the *INCLUDE_DIRECTORIES* directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A *;-list* of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A *;-list* of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;–list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

For example:

```
include(CheckSymbolExists)

# Check for macro SEEK_SET
check_symbol_exists(SEEK_SET "stdio.h" HAVE_SEEK_SET)
# Check for function fopen
check_symbol_exists(fopen "stdio.h" HAVE_FOPEN)
```

CheckTypeSize

Check sizeof a type

check_type_size

```
check_type_size(<type> <variable> [BUILTIN_TYPES_ONLY]
               [LANGUAGE <language>])
```

Check if the type exists and determine its size. Results are reported in the following variables:

HAVE_<variable>

Holds a true or false value indicating whether the type exists.

<variable>

Holds one of the following values:

<size> Type has non-zero size **<size>**.

0 Type has architecture-dependent size. This may occur when *CMAKE_OSX_ARCHITECTURES* has multiple architectures. In this case **<variable>_CODE** contains C preprocessor tests mapping from each architecture macro to the corresponding type size. The list of architecture macros is stored in **<variable>_KEYS**, and the value for each key is stored in **<variable>–<key>**.

"" (empty string)

Type does not exist.

<variable>_CODE

Holds C preprocessor code to define the macro **<variable>** to the size of the type, or to leave the macro undefined if the type does not exist.

The options are:

BUILTIN_TYPES_ONLY

Support only compiler-builtin types. If *not* given, the macro checks for headers **<sys/types.h>**, **<stdint.h>**, and **<stddef.h>**, and saves results in **HAVE_SYS_TYPES_H**, **HAVE_STDINT_H**, and **HAVE_STDDEF_H**. The type size check automatically includes the available headers, thus supporting checks of types defined in the headers.

LANGUAGE <language>

Use the <language> compiler to perform the check. Acceptable values are **C** and **CXX**.
If not specified, it defaults to **C**.

Despite the name of the macro you may use it to check the size of more complex expressions, too. To check e.g. for the size of a struct member you can do something like this:

```
check_type_size("((struct something*)0)->member" sizeof_MEMBER)
```

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by **<resultVar>** will also be added automatically.

CMAKE_REQUIRED_INCLUDES

A ;-list of header search paths to pass to the compiler. These will be the only header search paths used--the contents of the *INCLUDE_DIRECTORIES* directory property will be ignored.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_LINK_DIRECTORIES

Added in version 3.31.

A ;-list of libraries search paths to pass to the linker (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CMAKE_EXTRA_INCLUDE_FILES

list of extra headers to include.

Examples

Consider the code:

```
include(CheckTypeSize)

# Check for size of long.
check_type_size(long sizeof_LONG)
message("HAVE_SIZEOF_LONG: ${HAVE_SIZEOF_LONG}")
```

```
message("sizeof_long: ${sizeof_long}")
message("sizeof_long_code: ${sizeof_long_code}")
```

On a 64-bit architecture, the output may look something like this:

```
HAVE_SIZEOF_LONG: TRUE
sizeof_long: 8
sizeof_long_code: #define sizeof_long 8
```

On Apple platforms, when *CMAKE_OSX_ARCHITECTURES* has multiple architectures, types may have architecture-dependent sizes. For example, with the code

```
include(CheckTypeSize)

check_type_size(long sizeof_long)
message("HAVE_SIZEOF_LONG: ${HAVE_SIZEOF_LONG}")
message("sizeof_long: ${sizeof_long}")
foreach(key IN LISTS SIZE_OF_LONG_KEYS)
  message("key: ${key}")
  message("value: ${SIZE_OF_LONG-${key}}")
endforeach()
message("sizeof_long_code:
${sizeof_long_code}")
```

the result may be:

```
HAVE_SIZEOF_LONG: TRUE
sizeof_long: 0
key: __i386
value: 4
key: __x86_64
value: 8
sizeof_long_code:
#if defined(__i386)
# define SIZE_OF_LONG 4
#elif defined(__x86_64)
# define SIZE_OF_LONG 8
#else
# error SIZE_OF_LONG unknown
#endif
```

CheckVariableExists

Check if the variable exists.

check_variable_exists

```
check_variable_exists(<var> <variable>)
```

Check if the variable **<var>** exists and store the result in an internal cache variable **<variable>**.

This macro is only for **C** variables.

The following variables may be set before calling this macro to modify the way the check is run:

CMAKE_REQUIRED_FLAGS

String of additional flags to pass to the compiler. The string must be space-delimited--a ;-list will not work. The contents of *CMAKE_<LANG>_FLAGS* and its associated configuration-specific variable are automatically added to the compiler command before the contents of **CMAKE_REQUIRED_FLAGS**.

CMAKE_REQUIRED_DEFINITIONS

A ;-list of compiler definitions of the form **-DFOO** or **-DFOO=bar**. A definition for the name specified by *<resultVar>* will also be added automatically.

CMAKE_REQUIRED_LINK_OPTIONS

Added in version 3.14.

A ;-list of options to add to the link command (see *try_compile()* for further details).

CMAKE_REQUIRED_LIBRARIES

A ;-list of libraries to add to the link command. These can be the name of system libraries or they can be *Imported Targets* (see *try_compile()* for further details).

CMAKE_REQUIRED_QUIET

Added in version 3.1.

If this variable evaluates to a boolean true value, all status messages associated with the check will be suppressed.

CMakeAddFortranSubdirectory

Add a fortran-only subdirectory, find a fortran compiler, and build.

The **cmake_add_fortran_subdirectory** function adds a subdirectory to a project that contains a fortran-only subproject. The module will check the current compiler and see if it can support fortran. If no fortran compiler is found and the compiler is MSVC, then this module will find the MinGW gfortran. It will then use an external project to build with the MinGW tools. It will also create imported targets for the libraries created. This will only work if the fortran code is built into a dll, so *BUILD_SHARED_LIBS* is turned on in the project. In addition the *CMAKE_GNUtoMS* option is set to on, so that Microsoft *.lib* files are created. Usage is as follows:

```
cmake_add_fortran_subdirectory(
  <subdir>                # name of subdirectory
  PROJECT <project_name>  # project name in subdir top CMakeLists.txt
  ARCHIVE_DIR <dir>       # dir where project places .lib files
  RUNTIME_DIR <dir>       # dir where project places .dll files
  LIBRARIES <lib>...      # names of library targets to import
  LINK_LIBRARIES          # link interface libraries for LIBRARIES
  [LINK_LIBS <lib> <dep>...].
  CMAKE_COMMAND_LINE ...  # extra command line flags to pass to cmake
  NO_EXTERNAL_INSTALL     # skip installation of external project
)
```

Relative paths in **ARCHIVE_DIR** and **RUNTIME_DIR** are interpreted with respect to the build directory corresponding to the source directory in which the function is invoked.

Limitations:

NO_EXTERNAL_INSTALL is required for forward compatibility with a future version that supports installation of the external project binaries during **make install**.

CMakeBackwardCompatibilityCXX

define a bunch of backwards compatibility variables

```
CMAKE_ANSI_CXXFLAGS - flag for ansi c++
CMAKE_HAS_ANSI_STRING_STREAM - has <sstream>
include(TestForANSIStreamHeaders)
include(CheckIncludeFileCXX)
include(TestForSTDNamespace)
include(TestForANSIForScope)
```

CMakeDependentOption

Macro to provide an option dependent on other options.

This macro presents an option to the user only if a set of other conditions are true.

cmake_dependent_option

```
cmake_dependent_option(<option> "<help_text>" <value> <depends> <force>)
```

Makes **<option>** available to the user if the *semicolon-separated list* of conditions in **<depends>** are all true. Otherwise, a local variable named **<option>** is set to **<force>**.

When **<option>** is available, the given **<help_text>** and initial **<value>** are used. Otherwise, any value set by the user is preserved for when **<depends>** is satisfied in the future.

Note that the **<option>** variable only has a value which satisfies the **<depends>** condition within the scope of the caller because it is a local variable.

Added in version 3.22: Full *Condition Syntax* is now supported. See policy *CMP0127*.

Examples

Semicolon-separated list of conditions:

```
cmake_dependent_option(USE_FOO "Use Foo" ON "USE_BAR;NOT USE_ZOT" OFF)
```

If **USE_BAR** is true and **USE_ZOT** is false, this provides an option called **USE_FOO** that defaults to ON. Otherwise, it sets **USE_FOO** to OFF and hides the option from the user. If the status of **USE_BAR** or **USE_ZOT** ever changes, any value for the **USE_FOO** option is saved so that when the option is re-enabled it retains its old value.

Full condition syntax:

```
cmake_dependent_option(USE_FOO "Use Foo" ON "USE_A AND (USE_B OR USE_C)" OFF)
```

Similar to the previous example, if the argument with full condition syntax evaluates to true, this provides an option called **USE_FOO** that defaults to ON. Otherwise, it sets **USE_FOO** to OFF and hides the option from the user in the GUI. When condition changes, option is saved in similar way as described above. This enables using entire condition syntax as being the **if** clause argument, such as grouping conditions with parens and similar.

CMakeFindDependencyMacro**find_dependency**

The **find_dependency()** macro wraps a *find_package()* call for a package dependency:

```
find_dependency(<dep> [...])
```

It is designed to be used in a *Package Configuration File* (<PackageName>Config.cmake). **find_dependency** forwards the correct parameters for **QUIET** and **REQUIRED** which were passed to the original *find_package()* call. Any additional arguments specified are forwarded to *find_package()*.

If the dependency could not be found it sets an informative diagnostic message and calls *return()* to end processing of the calling package configuration file and return to the *find_package()* command that loaded it.

NOTE:

The call to *return()* makes this macro unsuitable to call from *Find Modules*.

Package Dependency Search Optimizations

If **find_dependency** is called with arguments identical to a previous call in the same directory, perhaps due to diamond-shaped package dependencies, the underlying call to *find_package()* is optimized out. This optimization is important to support large package dependency graphs while avoiding a combinatorial explosion of repeated searches. However, the heuristic cannot account for ambient variables that affect package behavior, such as <PackageName>_USE_STATIC_LIBS, offered by some packages. Therefore package configuration files should avoid setting such variables before their calls to **find_dependency**.

Changed in version 3.15: Previously, the underlying call to *find_package()* was always optimized out if the package had already been found. CMake 3.15 removed the optimization to support cases in which **find_dependency** call arguments request different components.

Changed in version 3.26: The pre-3.15 optimization was restored, but with the above-described heuristic to account for varying **find_dependency** call arguments.

CMakeFindPackageMode

This file is executed by cmake when invoked with *--find-package*. It expects that the following variables are set using *-D*:

NAME name of the package

COMPILER_ID

the CMake compiler ID for which the result is, i.e. GNU/Intel/Clang/MSVC, etc.

LANGUAGE

language for which the result will be used, i.e. C/CXX/Fortran/ASM

MODE

EXIST only check for existence of the given package

COMPILE

print the flags needed for compiling an object file which uses the given package

LINK print the flags needed for linking when using the given package

QUIET

if TRUE, don't print anything

CMakeGraphVizOptions

The builtin Graphviz support of CMake.

Generating Graphviz files

CMake can generate *Graphviz* files showing the dependencies between the targets in a project, as well as external libraries which are linked against.

When running CMake with the *--graphviz=foo.dot* option, it produces:

- a **foo.dot** file, showing all dependencies in the project
- a **foo.dot.<target>** file for each target, showing on which other targets it depends
- a **foo.dot.<target>.dependers** file for each target, showing which other targets depend on it

Those .dot files can be converted to images using the *dot* command from the Graphviz package:

```
dot -Tpng -o foo.png foo.dot
```

Added in version 3.10: The different dependency types **PUBLIC**, **INTERFACE** and **PRIVATE** are represented as solid, dashed and dotted edges.

Variables specific to the Graphviz support

The resulting graphs can be huge. The look and content of the generated graphs can be controlled using the file **CMakeGraphVizOptions.cmake**. This file is first searched in *CMAKE_BINARY_DIR*, and then in *CMAKE_SOURCE_DIR*. If found, the variables set in it are used to adjust options for the generated Graphviz files.

GRAPHVIZ_GRAPH_NAME

The graph name.

- Mandatory: NO
- Default: value of *CMAKE_PROJECT_NAME*

GRAPHVIZ_GRAPH_HEADER

The header written at the top of the Graphviz files.

- Mandatory: NO
- Default: "node [fontsize = "12"];"

GRAPHVIZ_NODE_PREFIX

The prefix for each node in the Graphviz files.

- Mandatory: NO
- Default: "node"

GRAPHVIZ_EXECUTABLES

Set to FALSE to exclude executables from the generated graphs.

- Mandatory: NO
- Default: TRUE

GRAPHVIZ_STATIC_LIBS

Set to FALSE to exclude static libraries from the generated graphs.

- Mandatory: NO
- Default: TRUE

GRAPHVIZ_SHARED_LIBS

Set to FALSE to exclude shared libraries from the generated graphs.

- Mandatory: NO
- Default: TRUE

GRAPHVIZ_MODULE_LIBS

Set to FALSE to exclude module libraries from the generated graphs.

- Mandatory: NO

- Default: TRUE

GRAPHVIZ_INTERFACE_LIBS

Set to FALSE to exclude interface libraries from the generated graphs.

- Mandatory: NO
- Default: TRUE

GRAPHVIZ_OBJECT_LIBS

Set to FALSE to exclude object libraries from the generated graphs.

- Mandatory: NO
- Default: TRUE

GRAPHVIZ_UNKNOWN_LIBS

Set to FALSE to exclude unknown libraries from the generated graphs.

- Mandatory: NO
- Default: TRUE

GRAPHVIZ_EXTERNAL_LIBS

Set to FALSE to exclude external libraries from the generated graphs.

- Mandatory: NO
- Default: TRUE

GRAPHVIZ_CUSTOM_TARGETS

Set to TRUE to include custom targets in the generated graphs.

- Mandatory: NO
- Default: FALSE

GRAPHVIZ_IGNORE_TARGETS

A list of regular expressions for names of targets to exclude from the generated graphs.

- Mandatory: NO
- Default: empty

GRAPHVIZ_GENERATE_PER_TARGET

Set to FALSE to not generate per-target graphs **foo.dot.<target>**.

- Mandatory: NO
- Default: TRUE

GRAPHVIZ_GENERATE_DEPENDERS

Set to FALSE to not generate depender graphs **foo.dot.<target>.dependers**.

- Mandatory: NO
- Default: TRUE

CMakePackageConfigHelpers

Helper functions for creating config files that can be included by other projects to find and use a package.

Generating a Package Configuration File**configure_package_config_file**

Create a config file for a project:

```
configure_package_config_file(<input> <output>
  INSTALL_DESTINATION <path>
  [PATH_VARS <var1> <var2> ... <varN>]
  [NO_SET_AND_CHECK_MACRO]
  [NO_CHECK_REQUIRED_COMPONENTS_MACRO]
```

```
[INSTALL_PREFIX <path>]
)
```

configure_package_config_file() should be used instead of the plain *configure_file()* command when creating the **<PackageName>Config.cmake** or **<PackageName>-config.cmake** file for installing a project or library. It helps make the resulting package relocatable by avoiding hardcoded paths in the installed **<PackageName>Config.cmake** file.

In a **FooConfig.cmake** file there may be code like this to make the install destinations known to the using project:

```
set(FOO_INCLUDE_DIR "@CMAKE_INSTALL_FULL_INCLUDEDIR" )
set(FOO_DATA_DIR "@CMAKE_INSTALL_PREFIX@/RELATIVE_DATA_INSTALL_DIR" )
set(FOO_ICONS_DIR "@CMAKE_INSTALL_PREFIX@/share/icons" )
#...logic to determine installedPrefix from the own location...
set(FOO_CONFIG_DIR "${installedPrefix}/@CONFIG_INSTALL_DIR" )
```

All four options shown above are not sufficient. The first three hardcode the absolute directory locations. The fourth case works only if the logic to determine the **installedPrefix** is correct, and if **CONFIG_INSTALL_DIR** contains a relative path, which in general cannot be guaranteed. This has the effect that the resulting **FooConfig.cmake** file would work poorly under Windows and macOS, where users are used to choosing the install location of a binary package at install time, independent from how **CMAKE_INSTALL_PREFIX** was set at build/cmake time.

Using **configure_package_config_file()** helps. If used correctly, it makes the resulting **FooConfig.cmake** file relocatable. Usage:

1. Write a **FooConfig.cmake.in** file as you are used to.
2. Insert a line at the top containing only the string **@PACKAGE_INIT@**.
3. Instead of **set(FOO_DIR "@SOME_INSTALL_DIR")**, use **set(FOO_DIR "@PACKAGE_SOME_INSTALL_DIR")** (this must be after the **@PACKAGE_INIT@** line).
4. Instead of using the normal *configure_file()* command, use **configure_package_config_file()**.

The **<input>** and **<output>** arguments are the input and output file, the same way as in *configure_file()*.

The **<path>** given to **INSTALL_DESTINATION** must be the destination where the **FooConfig.cmake** file will be installed to. This path can either be absolute, or relative to the **INSTALL_PREFIX** path.

The variables **<var1>** to **<varN>** given as **PATH_VARS** are the variables which contain install destinations. For each of them, the macro will create a helper variable **PACKAGE_<var...>**. These helper variables must be used in the **FooConfig.cmake.in** file for setting the installed location. They are calculated by **configure_package_config_file()** so that they are always relative to the installed location of the package. This works both for relative and also for absolute locations. For absolute locations, it works only if the absolute location is a subdirectory of **INSTALL_PREFIX**.

Added in version 3.30: The variable **PACKAGE_PREFIX_DIR** will always be defined after the **@PACKAGE_INIT@** line. It will hold the value of the base install location. In general, variables defined via the **PATH_VARS** mechanism should be used instead, but **PACKAGE_PREFIX_DIR** can be used for those cases not easily handled by **PATH_VARS**, such as for files installed directly to the base install location rather than a subdirectory of it.

NOTE:

When consumers of the generated file use CMake 3.29 or older, the value of **PACKAGE_PREFIX_DIR** can be changed by a call to *find_dependency()* or *find_package()*. If a project relies on

PACKAGE_PREFIX_DIR, it is the project's responsibility to ensure that the value of **PACKAGE_PREFIX_DIR** is preserved across any such calls, or any other calls which might include another file generated by **configure_package_config_file()**.

Added in version 3.1: If the **INSTALL_PREFIX** argument is passed, this is used as the base path to calculate all the relative paths. The **<path>** argument must be an absolute path. If this argument is not passed, the **CMAKE_INSTALL_PREFIX** variable will be used instead. The default value is good when generating a **FooConfig.cmake** file to use your package from the install tree. When generating a **FooConfig.cmake** file to use your package from the build tree, this option should be used.

By default, **configure_package_config_file()** also generates two helper macros, **set_and_check()** and **check_required_components()**, into the **FooConfig.cmake** file.

set_and_check() should be used instead of the normal **set()** command for setting directories and file locations. In addition to setting the variable, it also checks that the referenced file or directory actually exists and fails with a fatal error if it doesn't. This ensures that the generated **FooConfig.cmake** file does not contain wrong references. Add the **NO_SET_AND_CHECK_MACRO** option to prevent the generation of the **set_and_check()** macro in the **FooConfig.cmake** file.

check_required_components(<PackageName>) should be called at the end of the **FooConfig.cmake** file. This macro checks whether all requested, non-optional components have been found, and if this is not the case, it sets the **Foo_FOUND** variable to **FALSE** so that the package is considered to be not found. It does that by testing the **Foo_<Component>_FOUND** variables for all requested required components. This macro should be called even if the package doesn't provide any components to make sure users are not specifying components erroneously. Add the **NO_CHECK_REQUIRED_COMPONENTS_MACRO** option to prevent the generation of the **check_required_components()** macro in the **FooConfig.cmake** file.

See also *Example Generating Package Files*.

Generating a Package Version File

write_basic_package_version_file

Create a version file for a project:

```
write_basic_package_version_file(<filename>
    [VERSION <major.minor.patch>]
    COMPATIBILITY <AnyNewerVersion|SameMajorVersion|SameMinorVersion|ExactVersion>
    [ARCH_INDEPENDENT] )
```

Writes a file for use as a **<PackageName>ConfigVersion.cmake** file to **<filename>**. See the documentation of **find_package()** for details on such files.

<filename> is the output filename, which should be in the build tree. **<major.minor.patch>** is the version number of the project to be installed.

If no **VERSION** is given, the **PROJECT_VERSION** variable is used. If this hasn't been set, it errors out.

The **COMPATIBILITY** mode **AnyNewerVersion** means that the installed package version will be considered compatible if it is newer or exactly the same as the requested version. This mode should be used for packages which are fully backward compatible, also across major versions. If **SameMajorVersion** is used instead, then the behavior differs from **AnyNewerVersion** in that the major version number must be the same as requested, e.g. version 2.0 will not be considered compatible if 1.0 is requested. This mode should be used for packages which guarantee backward compatibility within the same major version. If **SameMinorVersion** is used, the behavior is the same as **SameMajorVersion**, but both major and minor version

must be the same as requested, e.g version 0.2 will not be compatible if 0.1 is requested. If **ExactVersion** is used, then the package is only considered compatible if the requested version matches exactly its own version number (not considering the tweak version). For example, version 1.2.3 of a package is only considered compatible to requested version 1.2.3. This mode is for packages without compatibility guarantees. If your project has more elaborate version matching rules, you will need to write your own custom **<PackageName>ConfigVersion.cmake** file instead of using this macro.

Added in version 3.11: The **SameMinorVersion** compatibility mode.

Added in version 3.14: If **ARCH_INDEPENDENT** is given, the installed package version will be considered compatible even if it was built for a different architecture than the requested architecture. Otherwise, an architecture check will be performed, and the package will be considered compatible only if the architecture matches exactly. For example, if the package is built for a 32-bit architecture, the package is only considered compatible if it is used on a 32-bit architecture, unless **ARCH_INDEPENDENT** is given, in which case the package is considered compatible on any architecture.

NOTE:

ARCH_INDEPENDENT is intended for header-only libraries or similar packages with no binaries.

Added in version 3.19: The version file generated by **AnyNewerVersion**, **SameMajorVersion** and **SameMinorVersion** arguments of **COMPATIBILITY** handle the version range, if one is specified (see *find_package()* command for the details). **ExactVersion** mode is incompatible with version ranges and will display an author warning if one is specified.

Internally, this macro executes *configure_file()* to create the resulting version file. Depending on the **COMPATIBILITY**, the corresponding **BasicConfigVersion-<COMPATIBILITY>.cmake.in** file is used. Please note that these files are internal to CMake and you should not call *configure_file()* on them yourself, but they can be used as a starting point to create more sophisticated custom **<PackageName>ConfigVersion.cmake** files.

Generating an Apple Platform Selection File

generate_apple_platform_selection_file

Added in version 3.29.

Create an Apple platform selection file:

```
generate_apple_platform_selection_file(<filename>
  INSTALL_DESTINATION <path>
  [INSTALL_PREFIX <path>]
  [MACOS_INCLUDE_FILE <file>]
  [IOS_INCLUDE_FILE <file>]
  [IOS_SIMULATOR_INCLUDE_FILE <file>]
  [IOS_CATALYST_INCLUDE_FILE <file>]
  [TVOS_INCLUDE_FILE <file>]
  [TVOS_SIMULATOR_INCLUDE_FILE <file>]
  [WATCHOS_INCLUDE_FILE <file>]
  [WATCHOS_SIMULATOR_INCLUDE_FILE <file>]
  [VISIONOS_INCLUDE_FILE <file>]
  [VISIONOS_SIMULATOR_INCLUDE_FILE <file>]
  [ERROR_VARIABLE <variable>]
)
```

Write a file that includes an Apple-platform-specific **.cmake** file, e.g., for use as **<Package-Name>Config.cmake**. This can be used in conjunction with the **XCFRAMEWORK_LOCATION** argument of *export(SETUP)* to export packages in a way that a project built for any Apple platform can use them.

INSTALL_DESTINATION **<path>**

Path to which the generated file will be installed by the caller, e.g., via *install(FILES)*. The path may be either relative to the **INSTALL_PREFIX** or absolute.

INSTALL_PREFIX **<path>**

Path prefix to which the package will be installed by the caller. The **<path>** argument must be an absolute path. If this argument is not passed, the **CMAKE_INSTALL_PREFIX** variable will be used instead.

MACOS_INCLUDE_FILE **<file>**

File to include if the platform is macOS.

IOS_INCLUDE_FILE **<file>**

File to include if the platform is iOS.

IOS_SIMULATOR_INCLUDE_FILE **<file>**

File to include if the platform is iOS Simulator.

IOS_CATALYST_INCLUDE_FILE **<file>**

Added in version 3.31.

File to include if the platform is iOS Catalyst.

TVOS_INCLUDE_FILE **<file>**

File to include if the platform is tvOS.

TVOS_SIMULATOR_INCLUDE_FILE **<file>**

File to include if the platform is tvOS Simulator.

WATCHOS_INCLUDE_FILE **<file>**

File to include if the platform is watchOS.

WATCHOS_SIMULATOR_INCLUDE_FILE **<file>**

File to include if the platform is watchOS Simulator.

VISIONOS_INCLUDE_FILE **<file>**

File to include if the platform is visionOS.

VISIONOS_SIMULATOR_INCLUDE_FILE **<file>**

File to include if the platform is visionOS Simulator.

ERROR_VARIABLE **<variable>**

If the consuming project is built for an unsupported platform, set **<variable>** to an error message. The includer may use this information to pretend the package was not found. If this option is not given, the default behavior is to issue a fatal error.

If any of the optional include files is not specified, and the consuming project is built for its corresponding platform, the generated file will consider the platform to be unsupported. The behavior is determined by the **ERROR_VARIABLE** option.

Generating an Apple Architecture Selection File

generate_apple_architecture_selection_file

Added in version 3.29.

Create an Apple architecture selection file:

```

generate_apple_architecture_selection_file(<filename>
  INSTALL_DESTINATION <path>
  [INSTALL_PREFIX <path>]
  [SINGLE_ARCHITECTURES <arch>...]
  SINGLE_ARCHITECTURE_INCLUDE_FILES <file>...]
  [UNIVERSAL_ARCHITECTURES <arch>...]
  UNIVERSAL_INCLUDE_FILE <file>]
  [ERROR_VARIABLE <variable>]
)

```

Write a file that includes an Apple-architecture-specific **.cmake** file based on *CMAKE_OSX_ARCHITECTURES*, e.g., for inclusion from an Apple-specific **<PackageName>Config.cmake** file.

INSTALL_DESTINATION <path>

Path to which the generated file will be installed by the caller, e.g., via *install(FILES)*. The path may be either relative to the **INSTALL_PREFIX** or absolute.

INSTALL_PREFIX <path>

Path prefix to which the package will be installed by the caller. The **<path>** argument must be an absolute path. If this argument is not passed, the *CMAKE_INSTALL_PREFIX* variable will be used instead.

SINGLE_ARCHITECTURES <arch>...

Architectures provided by entries of **SINGLE_ARCHITECTURE_INCLUDE_FILES**.

SINGLE_ARCHITECTURE_INCLUDE_FILES <file>...

Architecture-specific files. One of them will be loaded when *CMAKE_OSX_ARCHITECTURES* contains a single architecture matching the corresponding entry of **SINGLE_ARCHITECTURES**.

UNIVERSAL_ARCHITECTURES <arch>...

Architectures provided by the **UNIVERSAL_INCLUDE_FILE**.

The list may include **\$(ARCHS_STANDARD)** to support consumption using the *Xcode* generator, but the architectures should always be listed individually too.

UNIVERSAL_INCLUDE_FILE <file>

A file to load when *CMAKE_OSX_ARCHITECTURES* contains a (non-strict) subset of the **UNIVERSAL_ARCHITECTURES** and does not match any one of the **SINGLE_ARCHITECTURES**.

ERROR_VARIABLE <variable>

If the consuming project is built for an unsupported architecture, set **<variable>** to an error message. The includer may use this information to pretend the package was not found. If this option is not given, the default behavior is to issue a fatal error.

Example Generating Package Files

Example using both the *configure_package_config_file()* and *write_basic_package_version_file()* commands:

CMakeLists.txt

```

include(GNUInstallDirs)
set(INCLUDE_INSTALL_DIR ${CMAKE_INSTALL_INCLUDEDIR}/Foo
  CACHE PATH "Location of header files" )
set(SYSCONFIG_INSTALL_DIR ${CMAKE_INSTALL_SYSCONFDIR}/foo
  CACHE PATH "Location of configuration files" )
#...

```

```

include(CMakePackageConfigHelpers)
configure_package_config_file(FooConfig.cmake.in
    ${CMAKE_CURRENT_BINARY_DIR}/FooConfig.cmake
    INSTALL_DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/Foo
    PATH_VARS INCLUDE_INSTALL_DIR SYSCONFIG_INSTALL_DIR)
write_basic_package_version_file(
    ${CMAKE_CURRENT_BINARY_DIR}/FooConfigVersion.cmake
    VERSION 1.2.3
    COMPATIBILITY SameMajorVersion )
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/FooConfig.cmake
        ${CMAKE_CURRENT_BINARY_DIR}/FooConfigVersion.cmake
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/Foo )

```

FooConfig.cmake.in

```

set(FOO_VERSION x.y.z)
...
@PACKAGE_INIT@
...
set_and_check(FOO_INCLUDE_DIR "@PACKAGE_INCLUDE_INSTALL_DIR@")
set_and_check(FOO_SYSCONFIG_DIR "@PACKAGE_SYSCONFIG_INSTALL_DIR@")

check_required_components(Foo)

```

CMakePrintHelpers

Convenience functions for printing properties and variables, useful e.g. for debugging.

Commands

cmake_print_properties

```

cmake_print_properties(<TARGETS      [<target1> ...] |
                      SOURCES        [<source1> ...] |
                      DIRECTORIES    [<dir1> ...]   |
                      TESTS          [<test1> ...]   |
                      CACHE_ENTRIES  [<entry1> ...]  >
                      PROPERTIES     [<prop1> ...])

```

This function prints the values of the properties of the given targets, source files, directories, tests or cache entries. Exactly one of the scope keywords must be used. The scope keyword and its arguments must come before the **PROPERTIES** keyword in the arguments list.

cmake_print_variables

```
cmake_print_variables([var1 [var2 ... [varN]]])
```

This function prints the name of each variable followed by its value.

Examples

Printing the **LOCATION** and **INTERFACE_INCLUDE_DIRECTORIES** properties for both targets **foo** and **bar**:

```

include(CMakePrintHelpers)

cmake_print_properties(
    TARGETS foo bar
    PROPERTIES LOCATION INTERFACE_INCLUDE_DIRECTORIES
)

```

Gives:

```
--
Properties for TARGET foo:
  foo.LOCATION = "/usr/lib/libfoo.so"
  foo.INTERFACE_INCLUDE_DIRECTORIES = "/usr/include;/usr/include/foo"
Properties for TARGET bar:
  bar.LOCATION = "/usr/lib/libbar.so"
  bar.INTERFACE_INCLUDE_DIRECTORIES = "/usr/include;/usr/include/bar"
```

Printing given variables:

```
include(CMakePrintHelpers)

cmake_print_variables(CMAKE_C_COMPILER CMAKE_MAJOR_VERSION NOT_EXISTS)
```

Gives:

```
-- CMAKE_C_COMPILER="/usr/bin/cc" ; CMAKE_MAJOR_VERSION="3" ; NOT_EXISTS=""
```

CMakePrintSystemInformation

Print system information.

This module serves diagnostic purposes. Just include it in a project to see various internal CMake variables.

CMakePushCheckState

This module provides macros for managing the state of variables that influence how various CMake check commands (e.g., **check_symbol_exists()**, etc.) are performed. These macros save, reset, and restore the following variables:

- **CMAKE_REQUIRED_FLAGS**
- **CMAKE_REQUIRED_DEFINITIONS**
- **CMAKE_REQUIRED_INCLUDES**
- **CMAKE_REQUIRED_LINK_OPTIONS**
- **CMAKE_REQUIRED_LIBRARIES**
- **CMAKE_REQUIRED_LINK_DIRECTORIES**
- **CMAKE_REQUIRED_QUIET**
- **CMAKE_EXTRA_INCLUDE_FILES**

Macros

cmake_push_check_state

Saves (pushes) the current states of the above variables onto a stack. This is typically used to preserve the current configuration before making temporary modifications for specific checks.

```
cmake_push_check_state([RESET])
```

RESET

When this option is specified, the macro not only saves the current states of the listed variables but also resets them to empty, allowing them to be reconfigured from a clean state.

cmake_reset_check_state

Resets (clears) the contents of the variables listed above to empty states.

```
cmake_reset_check_state()
```

This macro can be used, for example, when performing multiple sequential checks that require entirely new configurations, ensuring no previous configuration unintentionally carries over.

cmake_pop_check_state

Restores the states of the variables listed above to their values at the time of the most recent **cmake_push_check_state()** call.

```
cmake_pop_check_state()
```

This macro is used to revert temporary changes made during a check. To prevent unexpected behavior, pair each **cmake_push_check_state()** with a corresponding **cmake_pop_check_state()**.

These macros are useful for scoped configuration, for example, in *Find modules* or when performing checks in a controlled environment, ensuring that temporary modifications are isolated to the scope of the check and do not propagate into other parts of the build system.

NOTE:

Other CMake variables, such as **CMAKE_<LANG>_FLAGS**, propagate to all checks regardless of these macros, as those fundamental variables are designed to influence the global state of the build system.

Examples

```
include(CMakePushCheckState)

# Save and reset the current state
cmake_push_check_state(RESET)

# Perform check with specific compile definitions
set(CMAKE_REQUIRED_DEFINITIONS -D_GNU_SOURCE)
include(CheckSymbolExists)
check_symbol_exists(memfd_create "sys/mman.h" HAVE_MEMFD_CREATE)

# Restore the original state
cmake_pop_check_state()
```

Variable states can be pushed onto the stack multiple times, allowing for nested or sequential configurations. Each **cmake_pop_check_state()** restores the most recent pushed states.

```
include(CMakePushCheckState)

# Save and reset the current state
cmake_push_check_state(RESET)

# Perform the first check with additional libraries
set(CMAKE_REQUIRED_LIBRARIES ${CMAKE_DL_LIBS})
include(CheckSymbolExists)
check_symbol_exists(dlopen "dlfcn.h" HAVE_DLOPEN)

# Save current state
cmake_push_check_state()

# Perform the second check with libraries and additional compile definitions
set(CMAKE_REQUIRED_DEFINITIONS -D_GNU_SOURCE)
check_symbol_exists(dladdr "dlfcn.h" HAVE_DLADDR)

message(STATUS "${CMAKE_REQUIRED_DEFINITIONS}")
```

```

# Output: -D_GNU_SOURCE

# Restore the previous state
cmake_pop_check_state()

message(STATUS "${CMAKE_REQUIRED_DEFINITIONS}")
# Output here is empty

# Reset variables to prepare for the next check
cmake_reset_check_state()

# Perform the next check only with additional compile definitions
set(CMAKE_REQUIRED_DEFINITIONS -D_GNU_SOURCE)
check_symbol_exists(dl_iterate_phdr "link.h" HAVE_DL_ITERATE_PHDR)

# Restore the original state
cmake_pop_check_state()

```

CMakeVerifyManifest

Use this script to verify that embedded manifests and side-by-side manifests for a project match.

This script first recursively globs ***.manifest** files from the current directory and creates a list of allowed version. Additional versions can be passed by setting **allow_versions** from the invocation command. Next, the script globs ***.exe** and ***.dll** files. Each **.exe** and **.dll** file is scanned for embedded manifests and the versions of CRT are checked to be in the list of allowed version.

Example

To run this script, navigate to a directory and run the script with **cmake -P**.

```
cmake -Dallow_versions=8.0.50608.0 -PCMakeVerifyManifest.cmake
```

This call allows an embedded manifest of 8.0.50608.0 to be used in a project, even if that version was not found in a **.manifest** file.

CPack

Configure generators for binary installers and source packages.

Introduction

The CPack module generates the configuration files **CPackConfig.cmake** and **CPackSourceConfig.cmake**. They are intended for use in a subsequent run of the *cpack* program where they steer the generation of installers or/and source packages.

Depending on the CMake generator, the CPack module may also add two new build targets, **package** and **package_source**. See the *packaging targets* section below for details.

The generated binary installers will contain all files that have been installed via CMake's *install()* command (and the deprecated commands *install_files()*, *install_programs()*, and *install_targets()*). Note that the **DESTINATION** option of the *install()* command must be a relative path; otherwise installed files are ignored by CPack.

Certain kinds of binary installers can be configured such that users can select individual application components to install. See the *CPackComponent* module for further details.

Source packages (configured through **CPackSourceConfig.cmake** and generated by the *CPack Archive Generator*) will contain all source files in the project directory except those specified in **CPACK_SOURCE_IGNORE_FILES**.

CPack Generators

The `CPACK_GENERATOR` variable has different meanings in different contexts. In a `CMakeLists.txt` file, `CPACK_GENERATOR` is a *list of generators*: and when `cpack` is run with no other arguments, it will iterate over that list and produce one package for each generator. In a `CPACK_PROJECT_CONFIG_FILE`, `CPACK_GENERATOR` is a *string naming a single generator*. If you need per-cpack-generator logic to control other cpack settings, then you need a `CPACK_PROJECT_CONFIG_FILE`. If set, the `CPACK_PROJECT_CONFIG_FILE` is included automatically on a per-generator basis. It only need contain overrides.

Here's how it works:

- `cpack` runs
- it includes `CPackConfig.cmake`
- it iterates over the generators given by the `-G` command line option, or if no such option was specified, over the list of generators given by the `CPACK_GENERATOR` variable set in the `CPackConfig.cmake` input file.
- foreach generator, it then
 - sets `CPACK_GENERATOR` to the one currently being iterated
 - includes the `CPACK_PROJECT_CONFIG_FILE`
 - produces the package for that generator

This is the key: For each generator listed in `CPACK_GENERATOR` in `CPackConfig.cmake`, `cpack` will *re-set* `CPACK_GENERATOR` internally to *the one currently being used* and then include the `CPACK_PROJECT_CONFIG_FILE`.

For a list of available generators, see `cpack-generators(7)`.

Targets `package` and `package_source`

If CMake is run with the Makefile, Ninja, or Xcode generator, then `include(CPack)` generates a target **package**. This makes it possible to build a binary installer from CMake, Make, or Ninja: Instead of **cpack**, one may call `cmake --build . --target package` or `make package` or `ninja package`. The VS generator creates an uppercase target **PACKAGE**.

If CMake is run with the Makefile or Ninja generator, then `include(CPack)` also generates a target **package_source**. To build a source package, instead of `cpack -G TGZ --config CPackSourceConfig.cmake` one may call `cmake --build . --target package_source`, `make package_source`, or `ninja package_source`.

Variables common to all CPack Generators

Before including this CPack module in your `CMakeLists.txt` file, there are a variety of variables that can be set to customize the resulting installers. The most commonly-used variables are:

`CPACK_PACKAGE_NAME`

The name of the package (or application). If not specified, it defaults to the project name.

`CPACK_PACKAGE_VENDOR`

The name of the package vendor. (e.g., "Kitware"). The default is "Humanity".

`CPACK_PACKAGE_DIRECTORY`

The directory in which CPack is doing its packaging. If it is not set then this will default (internally) to the build dir. This variable may be defined in a CPack config file or from the `cpack` command line option `-B`. If set, the command line option overrides the value found in the config file.

`CPACK_PACKAGE_VERSION_MAJOR`

Package major version. This variable will always be set, but its default value depends on whether or not version details were given to the `project()` command in the top level `CMakeLists.txt` file. If

version details were given, the default value will be `CMAKE_PROJECT_VERSION_MAJOR`. If no version details were given, a default version of 0.1.1 will be assumed, leading to `CPACK_PACKAGE_VERSION_MAJOR` having a default value of 0.

CPACK_PACKAGE_VERSION_MINOR

Package minor version. The default value is determined based on whether or not version details were given to the `project()` command in the top level `CMakeLists.txt` file. If version details were given, the default value will be `CMAKE_PROJECT_VERSION_MINOR`, but if no minor version component was specified then `CPACK_PACKAGE_VERSION_MINOR` will be left unset. If no project version was given at all, a default version of 0.1.1 will be assumed, leading to `CPACK_PACKAGE_VERSION_MINOR` having a default value of 1.

CPACK_PACKAGE_VERSION_PATCH

Package patch version. The default value is determined based on whether or not version details were given to the `project()` command in the top level `CMakeLists.txt` file. If version details were given, the default value will be `CMAKE_PROJECT_VERSION_PATCH`, but if no patch version component was specified then `CPACK_PACKAGE_VERSION_PATCH` will be left unset. If no project version was given at all, a default version of 0.1.1 will be assumed, leading to `CPACK_PACKAGE_VERSION_PATCH` having a default value of 1.

CPACK_PACKAGE_DESCRIPTION

A description of the project, used in places such as the introduction screen of CPack-generated Windows installers. If not set, the value of this variable is populated from the file named by `CPACK_PACKAGE_DESCRIPTION_FILE`.

CPACK_PACKAGE_DESCRIPTION_FILE

A text file used to describe the project when `CPACK_PACKAGE_DESCRIPTION` is not explicitly set. The default value for `CPACK_PACKAGE_DESCRIPTION_FILE` points to a built-in template file `Templates/CPack.GenericDescription.txt`.

CPACK_PACKAGE_DESCRIPTION_SUMMARY

Short description of the project (only a few words). If the `CMAKE_PROJECT_DESCRIPTION` variable is set, it is used as the default value, otherwise the default will be a string generated by CMake based on `CMAKE_PROJECT_NAME`.

CPACK_PACKAGE_HOMEPAGE_URL

Project homepage URL. The default value is taken from the `CMAKE_PROJECT_HOMEPAGE_URL` variable, which is set by the top level `project()` command, or else the default will be empty if no URL was provided to `project()`.

CPACK_PACKAGE_FILE_NAME

The name of the package file to generate, not including the extension. For example, `cmake-2.6.1-Linux-i686`. The default value is:

```
${CPACK_PACKAGE_NAME}-${CPACK_PACKAGE_VERSION}-${CPACK_SYSTEM_NAME}
```

CPACK_PACKAGE_INSTALL_DIRECTORY

Installation directory on the target system. This may be used by some CPack generators like NSIS to create an installation directory e.g., "CMake 2.5" below the installation prefix. All installed elements will be put inside this directory.

CPACK_PACKAGE_ICON

A branding image that will be displayed inside the installer (used by GUI installers).

CPACK_PACKAGE_CHECKSUM

Added in version 3.7.

An algorithm that will be used to generate an additional file with the checksum of the package. The output file name will be:

`${CPACK_PACKAGE_FILE_NAME}.${CPACK_PACKAGE_CHECKSUM}`

Supported algorithms are those listed by the *string(<HASH>)* command.

CPACK_PROJECT_CONFIG_FILE

CPack-time project CPack configuration file. This file is included at cpack time, once per generator after CPack has set *CPACK_GENERATOR* to the actual generator being used. It allows per-generator setting of **CPACK_*** variables at cpack time.

CPACK_RESOURCE_FILE_LICENSE

License to be embedded in the installer. It will typically be displayed to the user by the produced installer (often with an explicit "Accept" button, for graphical installers) prior to installation. This license file is NOT added to the installed files but is used by some CPack generators like NSIS. If you want to use UTF-8 characters, the file needs to be encoded in UTF-8 BOM. If you want to install a license file (may be the same as this one) along with your project, you must add an appropriate CMake *install()* command in your **CMakeLists.txt**.

CPACK_RESOURCE_FILE_README

ReadMe file to be embedded in the installer. It typically describes in some detail the purpose of the project during the installation. Not all CPack generators use this file.

CPACK_RESOURCE_FILE_WELCOME

Welcome file to be embedded in the installer. It welcomes users to this installer. Typically used in the graphical installers on Windows and Mac OS X.

CPACK_MONOLITHIC_INSTALL

Disables the component-based installation mechanism. When set, the component specification is ignored and all installed items are put in a single "MONOLITHIC" package. Some CPack generators do monolithic packaging by default and may be asked to do component packaging by setting **CPACK_<GENNAME>_COMPONENT_INSTALL** to **TRUE**.

CPACK_GENERATOR

List of CPack generators to use. If not specified, CPack will create a set of options following the naming pattern *CPACK_BINARY_<GENNAME>* (e.g. **CPACK_BINARY_NSIS**) allowing the user to enable/disable individual generators. If the *-G* option is given on the *cpack* command line, it will override this variable and any **CPACK_BINARY_<GENNAME>** options.

CPACK_OUTPUT_CONFIG_FILE

The name of the CPack binary configuration file. This file is the CPack configuration generated by the CPack module for binary installers. Defaults to **CPackConfig.cmake**.

CPACK_PACKAGE_EXECUTABLES

Lists each of the executables and associated text label to be used to create Start Menu shortcuts. For example, setting this to the list **ccmake;CMake** will create a shortcut named "CMake" that will execute the installed executable **ccmake**. Not all CPack generators use it (at least NSIS, Inno Setup and WIX do).

CPACK_STRIP_FILES

List of files to be stripped. Starting with CMake 2.6.0, **CPACK_STRIP_FILES** will be a boolean variable which enables stripping of all files (a list of files evaluates to **TRUE** in CMake, so this change is compatible).

CPACK_VERBATIM_VARIABLES

Added in version 3.4.

If set to **TRUE**, values of variables prefixed with **CPACK_** will be escaped before being written to the configuration files, so that the cpack program receives them exactly as they were specified. If not, characters like quotes and backslashes can cause parsing errors or alter the value received by the cpack program. Defaults to **FALSE** for backwards compatibility.

CPACK_THREADS

Added in version 3.20.

Number of threads to use when performing parallelized operations, such as compressing the installer package.

Some compression methods used by CPack generators such as Debian or Archive may take advantage of multiple CPU cores to speed up compression. **CPACK_THREADS** can be set to specify how many threads will be used for compression.

A positive integer can be used to specify an exact desired thread count.

When given a negative integer CPack will use the absolute value as the upper limit but may choose a lower value based on the available hardware concurrency.

Given 0 CPack will try to use all available CPU cores.

By default **CPACK_THREADS** is set to 1.

The following compression methods may take advantage of multiple cores:

xz Supported if CMake is built with a **liblzma** that supports parallel compression.

Added in version 3.21: Official CMake binaries available on **cmake.org** now ship with a **liblzma** that supports parallel compression. Older versions did not.

zstd Added in version 3.24.

Supported if CMake is built with libarchive 3.6 or higher. Official CMake binaries available on **cmake.org** support it.

Other compression methods ignore this value and use only one thread.

Variables for Source Package Generators

The following CPack variables are specific to source packages, and will not affect binary packages:

CPACK_SOURCE_PACKAGE_FILE_NAME

The name of the source package. For example **cmake-2.6.1**.

CPACK_SOURCE_STRIP_FILES

List of files in the source tree that will be stripped. Starting with CMake 2.6.0, **CPACK_SOURCE_STRIP_FILES** will be a boolean variable which enables stripping of all files (a list of files evaluates to **TRUE** in CMake, so this change is compatible).

CPACK_SOURCE_GENERATOR

List of generators used for the source packages. As with **CPACK_GENERATOR**, if this is not specified then CPack will create a set of options (e.g. **CPACK_SOURCE_ZIP**) allowing users to select which packages will be generated.

CPACK_SOURCE_OUTPUT_CONFIG_FILE

The name of the CPack source configuration file. This file is the CPack configuration generated by the CPack module for source installers. Defaults to **CPackSourceConfig.cmake**.

CPACK_SOURCE_IGNORE_FILES

Pattern of files in the source tree that won't be packaged when building a source package. This is a list of regular expression patterns (that must be properly escaped), e.g.,

`/CVS;/\.\svn;\.\swp$;\.\#;/#;.*~;cscope.*`

Variables for Advanced Use

The following variables are for advanced uses of CPack:

CPACK_CMAKE_GENERATOR

What CMake generator should be used if the project is a CMake project. Defaults to the value of *CMAKE_GENERATOR*. Few users will want to change this setting.

CPACK_INSTALL_CMAKE_PROJECTS

List of four values that specify what project to install. The four values are: Build directory, Project Name, Project Component, Directory. If omitted, CPack will build an installer that installs everything.

CPACK_SYSTEM_NAME

System name, defaults to the value of *CMAKE_SYSTEM_NAME*, except on Windows where it will be **win32** or **win64**.

CPACK_PACKAGE_VERSION

Package full version, used internally. By default, this is built from *CPACK_PACKAGE_VERSION_MAJOR*, *CPACK_PACKAGE_VERSION_MINOR*, and *CPACK_PACKAGE_VERSION_PATCH*.

CPACK_TOPLEVEL_TAG

Directory for the installed files.

CPACK_INSTALL_COMMANDS

Extra commands to install components. The environment variable **CMAKE_INSTALL_PREFIX** is set to the temporary install directory during execution.

CPACK_INSTALL_SCRIPTS

Added in version 3.16.

Extra CMake scripts executed by CPack during its local staging installation. They are executed before installing the files to be packaged. The scripts are not called by a standalone install (e.g.: **make install**). For every script, the following variables will be set: *CMAKE_CURRENT_SOURCE_DIR*, *CMAKE_CURRENT_BINARY_DIR* and *CMAKE_INSTALL_PREFIX* (which is set to the staging install directory). The singular form **CMAKE_INSTALL_SCRIPT** is supported as an alternative variable for historical reasons, but its value is ignored if **CMAKE_INSTALL_SCRIPTS** is set and a warning will be issued.

See also *CPACK_PRE_BUILD_SCRIPTS* and *CPACK_POST_BUILD_SCRIPTS* which can be used to specify scripts to be executed later in the packaging process.

CPACK_PRE_BUILD_SCRIPTS

Added in version 3.19.

List of CMake scripts to execute after CPack has installed the files to be packaged into a staging directory and before producing the package(s) from those files. See also *CPACK_INSTALL_SCRIPTS* and *CPACK_POST_BUILD_SCRIPTS*.

CPACK_POST_BUILD_SCRIPTS

Added in version 3.19.

List of CMake scripts to execute after CPack has produced the resultant packages and before copying them back to the build directory. See also *CPACK_INSTALL_SCRIPTS*, *CPACK_PRE_BUILD_SCRIPTS* and *CPACK_PACKAGE_FILES*.

CPACK_PACKAGE_FILES

Added in version 3.19.

List of package files created in the staging directory, with each file provided as a full absolute path. This variable is populated by CPack just before invoking the post-build scripts listed in *CPACK_POST_BUILD_SCRIPTS*. It is the preferred way for the post-build scripts to know the set of package files to operate on. Projects should not try to set this variable themselves.

CPACK_INSTALLED_DIRECTORIES

Extra directories to install.

CPACK_PACKAGE_INSTALL_REGISTRY_KEY

Registry key used when installing this project. This is only used by installers for Windows. The default value is based on the installation directory.

CPACK_CREATE_DESKTOP_LINKS

List of desktop links to create. Each desktop link requires a corresponding start menu shortcut as created by *CPACK_PACKAGE_EXECUTABLES*.

CPACK_BINARY_<GENNAME>

CPack generated options for binary generators. The **CPack.cmake** module generates (when *CPACK_GENERATOR* is not set) a set of CMake options (see CMake *option()* command) which may then be used to select the CPack generator(s) to be used when building the **package** target or when running *cpack* without the *-G* option.

CPACK_READSELF_EXECUTABLE

Added in version 3.25.

Specify the **readelf** executable path used by CPack. The default value will be taken from the **CMAKE_READSELF** variable, if set, which may be populated CMake when enabling languages. If **CMAKE_READSELF** is not set, CPack will use *find_program()* to determine the **readelf** path when needed.

CPACK_OBJCOPY_EXECUTABLE

Added in version 3.25.

Specify the **objcopy** executable path used by CPack. The default value will be taken from the **CMAKE_OBJCOPY** variable, if set, which may be populated by CMake when enabling languages. If **CMAKE_OBJCOPY** is not set, CPack will use *find_program()* to determine the **objcopy** path when needed.

CPACK_OBJDUMP_EXECUTABLE

Added in version 3.25.

Specify the **objdump** executable path used by CPack. The default value will be taken from the **CMAKE_OBJDUMP** variable, which may be populated by CMake when enabling languages. If **CMAKE_OBJDUMP** is not set, CPack will use *find_program()* to determine the **objdump** path when needed.

CPackComponent

Configure components for binary installers and source packages.

Introduction

This module is automatically included by *CPack*.

Certain binary installers (especially the graphical installers) generated by CPack allow users to select

individual application *components* to install. This module allows developers to configure the packaging of such components.

Contents is assigned to components by the **COMPONENT** argument of CMake's *install()* command. Components can be annotated with user-friendly names and descriptions, inter-component dependencies, etc., and grouped in various ways to customize the resulting installer, using the commands described below.

To specify different groupings for different CPack generators use a `CPACK_PROJECT_CONFIG_FILE`.

Variables

The following variables influence the component-specific packaging:

CPACK_COMPONENTS_ALL

The list of component to install.

The default value of this variable is computed by CPack and contains all components defined by the project. The user may set it to only include the specified components.

Instead of specifying all the desired components, it is possible to obtain a list of all defined components and then remove the unwanted ones from the list. The `get_cmake_property()` command can be used to obtain the **COMPONENTS** property, then the `list(REMOVE_ITEM)` command can be used to remove the unwanted ones. For example, to use all defined components except **foo** and **bar**:

```
get_cmake_property(CPACK_COMPONENTS_ALL COMPONENTS)
list(REMOVE_ITEM CPACK_COMPONENTS_ALL "foo" "bar")
```

CPACK_<GENNAME>_COMPONENT_INSTALL

Enable/Disable component install for CPack generator <GENNAME>.

Each CPack Generator (RPM, DEB, ARCHIVE, NSIS, DMG, etc...) has a legacy default behavior. e.g. RPM builds monolithic whereas NSIS builds component. One can change the default behavior by setting this variable to 0/1 or OFF/ON.

CPACK_COMPONENTS_GROUPING

Specify how components are grouped for multi-package component-aware CPack generators.

Some generators like RPM or ARCHIVE (TGZ, ZIP, ...) may generate several packages files when there are components, depending on the value of this variable:

- **ONE_PER_GROUP** (default): create one package per component group
- **IGNORE** : create one package per component (ignore the groups)
- **ALL_COMPONENTS_IN_ONE** : create a single package with all requested components

CPACK_COMPONENT_<compName>_DISPLAY_NAME

The name to be displayed for a component.

CPACK_COMPONENT_<compName>_DESCRIPTION

The description of a component.

CPACK_COMPONENT_<compName>_GROUP

The group of a component.

CPACK_COMPONENT_<compName>_DEPENDS

The dependencies (list of components) on which this component depends.

CPACK_COMPONENT_<compName>_HIDDEN

True if this component is hidden from the user.

CPACK_COMPONENT_<compName>_REQUIRED

True if this component is required.

CPACK_COMPONENT_<compName>_DISABLED

True if this component is not selected to be installed by default.

Commands**Add component****cpack_add_component**

Describe an installation component.

```
cpack_add_component(compname
    [DISPLAY_NAME name]
    [DESCRIPTION description]
    [HIDDEN | REQUIRED | DISABLED ]
    [GROUP group]
    [DEPENDS comp1 comp2 ... ]
    [INSTALL_TYPES type1 type2 ... ]
    [DOWNLOADED]
    [ARCHIVE_FILE filename]
    [PLIST filename])
```

compname is the name of an installation component, as defined by the **COMPONENT** argument of one or more CMake *install()* commands. With the **cpack_add_component** command one can set a name, a description, and other attributes of an installation component. One can also assign a component to a component group.

DISPLAY_NAME is the displayed name of the component, used in graphical installers to display the component name. This value can be any string.

DESCRIPTION is an extended description of the component, used in graphical installers to give the user additional information about the component. Descriptions can span multiple lines using `\n` as the line separator. Typically, these descriptions should be no more than a few lines long.

HIDDEN indicates that this component will be hidden in the graphical installer, so that the user cannot directly change whether it is installed or not.

REQUIRED indicates that this component is required, and therefore will always be installed. It will be visible in the graphical installer, but it cannot be unselected. (Typically, required components are shown grayed out).

DISABLED indicates that this component should be disabled (unselected) by default. The user is free to select this component for installation, unless it is also **HIDDEN**.

DEPENDS lists the components on which this component depends. If this component is selected, then each of the components listed must also be selected. The dependency information is encoded within the installer itself, so that users cannot install inconsistent sets of components.

GROUP names the component group of which this component is a part. If not provided, the component will be a standalone component, not part of any component group. Component groups are described with the `cpack_add_component_group` command, detailed below.

INSTALL_TYPES lists the installation types of which this component is a part. When one of these installation types is selected, this component will automatically be selected. Installation types are described with the `cpack_add_install_type` command, detailed below.

DOWNLOADED indicates that this component should be downloaded on-the-fly by the installer, rather than packaged in with the installer itself. For more information, see the `cpack_configure_downloads` command.

ARCHIVE_FILE provides a name for the archive file created by CPack to be used for downloaded components. If not supplied, CPack will create a file with some name based on `CPACK_PACKAGE_FILE_NAME` and the name of the component. See `cpack_configure_downloads` for more information.

PLIST gives a filename that is passed to `pkgbuild` with the `--component-plist` argument when using the `productbuild` generator.

Add component group

`cpack_add_component_group`

Describes a group of related CPack installation components.

```
cpack_add_component_group(groupname
                           [DISPLAY_NAME name]
                           [DESCRIPTION description]
                           [PARENT_GROUP parent]
                           [EXPANDED]
                           [BOLD_TITLE])
```

The `cpack_add_component_group` describes a group of installation components, which will be placed together within the listing of options. Typically, component groups allow the user to select/deselect all of the components within a single group via a single group-level option. Use component groups to reduce the complexity of installers with many options. `groupname` is an arbitrary name used to identify the group in the `GROUP` argument of the `cpack_add_component` command, which is used to place a component in a group. The name of the group must not conflict with the name of any component.

`DISPLAY_NAME` is the displayed name of the component group, used in graphical installers to display the component group name. This value can be any string.

`DESCRIPTION` is an extended description of the component group, used in graphical installers to give the user additional information about the components within that group. Descriptions can span multiple lines using `\n` as the line separator. Typically, these descriptions should be no more than a few lines long.

`PARENT_GROUP`, if supplied, names the parent group of this group. Parent groups are used to establish a hierarchy of groups, providing an arbitrary hierarchy of groups.

`EXPANDED` indicates that, by default, the group should show up as "expanded", so that the user immediately sees all of the components within the group. Otherwise, the group will initially show up as a single entry.

`BOLD_TITLE` indicates that the group title should appear in bold, to call the user's attention to the group.

Add installation type

`cpack_add_install_type`

Add a new installation type containing a set of predefined component selections to the graphical installer.

```
cpack_add_install_type(typename
                       [DISPLAY_NAME name])
```

The `cpack_add_install_type` command identifies a set of preselected components that represents a common

use case for an application. For example, a "Developer" install type might include an application along with its header and library files, while an "End user" install type might just include the application's executable. Each component identifies itself with one or more install types via the `INSTALL_TYPES` argument to `cpack_add_component`.

`DISPLAY_NAME` is the displayed name of the install type, which will typically show up in a drop-down box within a graphical installer. This value can be any string.

Configure downloads

`cpack_configure_downloads`

Configure CPack to download selected components on-the-fly as part of the installation process.

```
cpack_configure_downloads(site
                           [UPLOAD_DIRECTORY dirname]
                           [ALL]
                           [ADD_REMOVE|NO_ADD_REMOVE])
```

The `cpack_configure_downloads` command configures installation-time downloads of selected components. For each downloadable component, CPack will create an archive containing the contents of that component, which should be uploaded to the given site. When the user selects that component for installation, the installer will download and extract the component in place. This feature is useful for creating small installers that only download the requested components, saving bandwidth. Additionally, the installers are small enough that they will be installed as part of the normal installation process, and the "Change" button in Windows Add/Remove Programs control panel will allow one to add or remove parts of the application after the original installation. On Windows, the downloaded-components functionality requires the ZipDLL plug-in for NSIS, available at:

http://nsis.sourceforge.net/ZipDLL_plug-in

On macOS, installers that download components on-the-fly can only be built and installed on system using macOS 10.5 or later.

The `site` argument is a URL where the archives for downloadable components will reside, e.g., - <https://cmake.org/files/v3.25/> All of the archives produced by CPack should be uploaded to that location.

`UPLOAD_DIRECTORY` is the local directory where CPack will create the various archives for each of the components. The contents of this directory should be uploaded to a location accessible by the URL given in the `site` argument. If omitted, CPack will use the directory `CPackUploads` inside the CMake binary directory to store the generated archives.

The `ALL` flag indicates that all components be downloaded. Otherwise, only those components explicitly marked as `DOWNLOADED` or that have a specified `ARCHIVE_FILE` will be downloaded. Additionally, the `ALL` option implies `ADD_REMOVE` (unless `NO_ADD_REMOVE` is specified).

`ADD_REMOVE` indicates that CPack should install a copy of the installer that can be called from Windows' Add/Remove Programs dialog (via the "Modify" button) to change the set of installed components. `NO_ADD_REMOVE` turns off this behavior. This option is ignored on Mac OS X.

CPackIFW

Added in version 3.1.

This module looks for the location of the command-line utilities supplied with the *Qt Installer Framework* (QtIFW).

The module also defines several commands to control the behavior of the *CPack IFW Generator*.

Commands

The module defines the following commands:

cpack_ifw_configure_component

Sets the arguments specific to the CPack IFW generator.

```
cpack_ifw_configure_component(<compname> [COMMON] [ESSENTIAL] [VIRTUAL]
                             [FORCED_INSTALLATION] [REQUIRES_ADMIN_RIGHTS]
                             [NAME <name>]
                             [DISPLAY_NAME <display_name>] # Note: Internationalization
                             [DESCRIPTION <description>] # Note: Internationalization
                             [UPDATE_TEXT <update_text>]
                             [VERSION <version>]
                             [RELEASE_DATE <release_date>]
                             [SCRIPT <script>]
                             [PRIORITY|SORTING_PRIORITY <sorting_priority>] # Note: I
                             [DEPENDS|DEPENDENCIES <com_id> ...]
                             [AUTO_DEPEND_ON <comp_id> ...]
                             [LICENSES <display_name> <file_path> ...]
                             [DEFAULT <value>]
                             [USER_INTERFACES <file_path> <file_path> ...]
                             [TRANSLATIONS <file_path> <file_path> ...]
                             [REPLACES <comp_id> ...]
                             [CHECKABLE <value>])
```

This command should be called after *cpack_add_component()* command.

COMMON

if set, then the component will be packaged and installed as part of a group to which it belongs.

ESSENTIAL

Added in version 3.6.

if set, then the package manager stays disabled until that component is updated.

VIRTUAL

Added in version 3.8.

if set, then the component will be hidden from the installer. It is a equivalent of the **HIDDEN** option from the *cpack_add_component()* command.

FORCED_INSTALLATION

Added in version 3.8.

if set, then the component must always be installed. It is a equivalent of the **REQUIRED** option from the *cpack_add_component()* command.

REQUIRES_ADMIN_RIGHTS

Added in version 3.8.

set it if the component needs to be installed with elevated permissions.

NAME is used to create domain-like identification for this component. By default used origin component name.

DISPLAY_NAME

Added in version 3.8.

set to rewrite original name configured by *cpack_add_component()* command.

DESCRIPTION

Added in version 3.8.

set to rewrite original description configured by *cpack_add_component()* command.

UPDATE_TEXT

Added in version 3.8.

will be added to the component description if this is an update to the component.

VERSION

is version of component. By default used *CPACK_PACKAGE_VERSION*.

RELEASE_DATE

Added in version 3.8.

keep empty to auto generate.

SCRIPT

is a relative or absolute path to operations script for this component.

SORTING_PRIORITY

Added in version 3.8.

is priority of the component in the tree.

PRIORITY

Deprecated since version 3.8: Old name for **SORTING_PRIORITY**.

DEPENDS, DEPENDENCIES

Added in version 3.8.

list of dependency component or component group identifiers in QtIFW style.

Added in version 3.21.

Component or group names listed as dependencies may contain hyphens. This requires QtIFW 3.1 or later.

AUTO_DEPEND_ON

Added in version 3.8.

list of identifiers of component or component group in QtIFW style that this component

has an automatic dependency on.

LICENSES

pair of <display_name> and <file_path> of license text for this component. You can specify more than one license.

DEFAULT

Added in version 3.8.

Possible values are: TRUE, FALSE, and SCRIPT. Set to FALSE to disable the component in the installer or to SCRIPT to resolved during runtime (don't forget add the file of the script as a value of the **SCRIPT** option).

USER_INTERFACES

Added in version 3.7.

is a list of <file_path> ('.ui' files) representing pages to load.

TRANSLATIONS

Added in version 3.8.

is a list of <file_path> ('.qm' files) representing translations to load.

REPLACES

Added in version 3.10.

list of identifiers of component or component group to replace.

CHECKABLE

Added in version 3.10.

Possible values are: TRUE, FALSE. Set to FALSE if you want to hide the checkbox for an item. This is useful when only a few subcomponents should be selected instead of all.

cpack_ifw_configure_component_group

Sets the arguments specific to the CPack IFW generator.

```
cpack_ifw_configure_component_group(<groupname> [VIRTUAL]
                                     [FORCED_INSTALLATION] [REQUIRES_ADMIN_RIGHTS]
                                     [NAME <name>]
                                     [DISPLAY_NAME <display_name>] # Note: Internationalizat
                                     [DESCRIPTION <description>] # Note: Internationalization
                                     [UPDATE_TEXT <update_text>]
                                     [VERSION <version>]
                                     [RELEASE_DATE <release_date>]
                                     [SCRIPT <script>]
                                     [PRIORITY|SORTING_PRIORITY <sorting_priority>] # Note: I
                                     [DEPENDS|DEPENDENCIES <com_id> ...]
                                     [AUTO_DEPEND_ON <comp_id> ...]
                                     [LICENSES <display_name> <file_path> ...]
                                     [DEFAULT <value>]
                                     [USER_INTERFACES <file_path> <file_path> ...]
                                     [TRANSLATIONS <file_path> <file_path> ...]
                                     [REPLACES <comp_id> ...]
```

[CHECKABLE <value>)]

This command should be called after *cpack_add_component_group()* command.

VIRTUAL

Added in version 3.8.

if set, then the group will be hidden from the installer. Note that setting this on a root component does not work.

FORCED_INSTALLATION

Added in version 3.8.

if set, then the group must always be installed.

REQUIRES_ADMIN_RIGHTS

Added in version 3.8.

set it if the component group needs to be installed with elevated permissions.

NAME is used to create domain-like identification for this component group. By default used origin component group name.

DISPLAY_NAME

Added in version 3.8.

set to rewrite original name configured by *cpack_add_component_group()* command.

DESCRIPTION

Added in version 3.8.

set to rewrite original description configured by *cpack_add_component_group()* command.

UPDATE_TEXT

Added in version 3.8.

will be added to the component group description if this is an update to the component group.

VERSION

is version of component group. By default used *CPACK_PACKAGE_VERSION*.

RELEASE_DATE

Added in version 3.8.

keep empty to auto generate.

SCRIPT

is a relative or absolute path to operations script for this component group.

SORTING_PRIORITY

is priority of the component group in the tree.

PRIORITY

Deprecated since version 3.8: Old name for **SORTING_PRIORITY**.

DEPENDS, DEPENDENCIES

Added in version 3.8.

list of dependency component or component group identifiers in QtIFW style.

Added in version 3.21.

Component or group names listed as dependencies may contain hyphens. This requires QtIFW 3.1 or later.

AUTO_DEPEND_ON

Added in version 3.8.

list of identifiers of component or component group in QtIFW style that this component group has an automatic dependency on.

LICENSES

pair of <display_name> and <file_path> of license text for this component group. You can specify more than one license.

DEFAULT

Added in version 3.8.

Possible values are: TRUE, FALSE, and SCRIPT. Set to TRUE to preselect the group in the installer (this takes effect only on groups that have no visible child components) or to SCRIPT to resolved during runtime (don't forget add the file of the script as a value of the **SCRIPT** option).

USER_INTERFACES

Added in version 3.7.

is a list of <file_path> ('.ui' files) representing pages to load.

TRANSLATIONS

Added in version 3.8.

is a list of <file_path> ('.qm' files) representing translations to load.

REPLACES

Added in version 3.10.

list of identifiers of component or component group to replace.

CHECKABLE

Added in version 3.10.

Possible values are: TRUE, FALSE. Set to FALSE if you want to hide the checkbox for

an item. This is useful when only a few subcomponents should be selected instead of all.

cpack_ifw_add_repository

Add QtIFW specific remote repository to binary installer.

```
cpack_ifw_add_repository(<reponame> [DISABLED]
                        URL <url>
                        [USERNAME <username>]
                        [PASSWORD <password>]
                        [DISPLAY_NAME <display_name>])
```

This command will also add the <reponame> repository to a variable *CPACK_IFW_REPOSITORIES_ALL*.

DISABLED

if set, then the repository will be disabled by default.

URL is points to a list of available components.

USERNAME

is used as user on a protected repository.

PASSWORD

is password to use on a protected repository.

DISPLAY_NAME

is string to display instead of the URL.

cpack_ifw_update_repository

Added in version 3.6.

Update QtIFW specific repository from remote repository.

```
cpack_ifw_update_repository(<reponame>
                          [[ADD|REMOVE] URL <url>]|
                          [REPLACE OLD_URL <old_url> NEW_URL <new_url>]]
                          [USERNAME <username>]
                          [PASSWORD <password>]
                          [DISPLAY_NAME <display_name>])
```

This command will also add the <reponame> repository to a variable *CPACK_IFW_REPOSITORIES_ALL*.

URL is points to a list of available components.

OLD_URL

is points to a list that will be replaced.

NEW_URL

is points to a list that will replace to.

USERNAME

is used as user on a protected repository.

PASSWORD

is password to use on a protected repository.

DISPLAY_NAME

is string to display instead of the URL.

cpack_ifw_add_package_resources

Added in version 3.7.

Add additional resources in the installer binary.

```
cpack_ifw_add_package_resources(<file_path> <file_path> ...)
```

This command will also add the specified files to a variable `CPACK_IFW_PACKAGE_RESOURCES`.

CPackIFWConfigureFile

Added in version 3.8.

The module defines `configure_file()` similar command to configure file templates prepared in QtIFW/SDK/Creator style.

Commands

The module defines the following commands:

cpack_ifw_configure_file

Copy a file to another location and modify its contents.

```
cpack_ifw_configure_file(<input> <output>)
```

Copies an **<input>** file to an **<output>** file and substitutes variable values referenced as **%{VAR}** or **%VAR%** in the input file content. Each variable reference will be replaced with the current value of the variable, or the empty string if the variable is not defined.

CSharpUtilities

Added in version 3.8.

Functions to make configuration of CSharp/.NET targets easier.

A collection of CMake utility functions useful for dealing with CSharp targets for Visual Studio generators from version 2010 and later.

The following functions are provided by this module:

Main functions

- `csharp_set_windows_forms_properties()`
- `csharp_set_designer_cs_properties()`
- `csharp_set_xaml_cs_properties()`

Helper functions

- `csharp_get_filename_keys()`
- `csharp_get_filename_key_base()`
- `csharp_get_dependentupon_name()`

Main functions provided by the module**csharp_set_windows_forms_properties**

Sets source file properties for use of Windows Forms. Use this, if your CSharp target uses Windows Forms:

```
csharp_set_windows_forms_properties([<file1> [<file2> [...]]])
```

<fileN>

List of all source files which are relevant for setting the `VS_CSHARP_<tagname>` properties (including `.cs`, `.resx` and `.Designer.cs` extensions).

In the list of all given files for all files ending with `.Designer.cs` and `.resx` is searched. For every *designer* or *resource* file a file with the same base name but only `.cs` as extension is searched. If this is found, the `VS_CSHARP_<tagname>` properties are set as follows:

for the `.cs` file:

- `VS_CSHARP_SubType` "Form"

for the `.Designer.cs` file (if it exists):

- `VS_CSHARP_DependentUpon` <cs-filename>
- `VS_CSHARP_DesignTime` "" (delete tag if previously defined)
- `VS_CSHARP_AutoGen` "" (delete tag if previously defined)

for the `.resx` file (if it exists):

- `VS_RESOURCE_GENERATOR` "" (delete tag if previously defined)
- `VS_CSHARP_DependentUpon` <cs-filename>
- `VS_CSHARP_SubType` "Designer"

csharp_set_designer_cs_properties

Sets source file properties of `.Designer.cs` files depending on sibling filenames. Use this, if your CSharp target does **not** use Windows Forms (for Windows Forms use `csharp_set_windows_forms_properties()` instead):

```
csharp_set_designer_cs_properties([<file1> [<file2> [...]]])
```

<fileN>

List of all source files which are relevant for setting the `VS_CSHARP_<tagname>` properties (including `.cs`, `.resx`, `.settings` and `.Designer.cs` extensions).

In the list of all given files for all files ending with `.Designer.cs` is searched. For every *designer* file all files with the same base name but different extensions are searched. If a match is found, the source file properties of the *designer* file are set depending on the extension of the matched file:

if match is `.resx` file:

- `VS_CSHARP_AutoGen` "True"
- `VS_CSHARP_DesignTime` "True"
- `VS_CSHARP_DependentUpon` <resx-filename>

if match is `.cs` file:

- `VS_CSHARP_DependentUpon` <cs-filename>

if match is `.settings` file:

- `VS_CSHARP_AutoGen` "True"
- `VS_CSHARP_DesignTimeSharedInput` "True"
- `VS_CSHARP_DependentUpon` <settings-filename>

NOTE:

Because the source file properties of the **.Designer.cs** file are set according to the found matches and every match sets the **VS_CSHARP_DependentUpon** property, there should only be one match for each **Designer.cs** file.

csharp_set_xaml_cs_properties

Sets source file properties for use of Windows Presentation Foundation (WPF) and XAML. Use this, if your CSharp target uses WPF/XAML:

```
csharp_set_xaml_cs_properties([<file1> [<file2> [...]]])
<fileN>
    List of all source files which are relevant for setting the VS_CSHARP_<tagname> properties (including .cs, .xaml, and .xaml.cs extensions).
```

In the list of all given files for all files ending with **.xaml.cs** is searched. For every *xaml-cs* file, a file with the same base name but extension **.xaml** is searched. If a match is found, the source file properties of the **.xaml.cs** file are set:

- **VS_CSHARP_DependentUpon** <xaml-filename>

Helper functions which are used by the above ones

csharp_get_filename_keys

Helper function which computes a list of key values to identify source files independently of relative/absolute paths given in cmake and eliminates case sensitivity:

```
csharp_get_filename_keys(OUT [<file1> [<file2> [...]]])
OUT    Name of the variable in which the list of keys is stored
<fileN>
    filename(s) as given to CSharp target using add_library() or add_executable()
```

In some way the function applies a canonicalization to the source names. This is necessary to find file matches if the files have been added to the target with different directory prefixes:

```
add_library(lib
  myfile.cs
  ${CMAKE_CURRENT_SOURCE_DIR}/myfile.Designer.cs)

set_source_files_properties(myfile.Designer.cs PROPERTIES
  VS_CSHARP_DependentUpon myfile.cs)

# this will fail, because in cmake
# - ${CMAKE_CURRENT_SOURCE_DIR}/myfile.Designer.cs
# - myfile.Designer.cs
# are not the same source file. The source file property is not set.
```

csharp_get_filename_key_base

Returns the full filepath and name **without** extension of a key. **KEY** is expected to be a key from **csharp_get_filename_keys**. In **BASE** the value of **KEY** without the file extension is returned:

```
csharp_get_filename_key_base(BASE KEY)
BASE    Name of the variable with the computed "base" of KEY.
KEY     The key of which the base will be computed. Expected to be a upper case full filename.
```

csharp_get_dependentupon_name

Computes a string which can be used as value for the source file property **VS_CSHARP_<tagname>** with *target* being **DependentUpon**:

```
csharp_get_dependentupon_name(NAME FILE)
```

NAME Name of the variable with the result value

FILE Filename to convert to <**DependentUpon**> value

Actually this is only the filename without any path given at the moment.

CTest

Configure a project for testing with CTest/CDash

Include this module in the top CMakeLists.txt file of a project to enable testing with CTest and dashboard submissions to CDash:

```
project(MyProject)
...
include(CTest)
```

The module automatically creates the following variables:

BUILD_TESTING

Option selecting whether **include(CTest)** calls *enable_testing()*. The option is **ON** by default when created by the module.

After including the module, use code like:

```
if(BUILD_TESTING)
# ... CMake code to create tests ...
endif()
```

to creating tests when testing is enabled.

To enable submissions to a CDash server, create a **CTestConfig.cmake** file at the top of the project with content such as:

```
set(CTEST_NIGHTLY_START_TIME "01:00:00 UTC")
set(CTEST_SUBMIT_URL "http://my.cdash.org/submit.php?project=MyProject")
```

(the CDash server can provide the file to a project administrator who configures **MyProject**). Settings in the config file are shared by both this **CTest** module and the *ctest(1)* command-line *Dashboard Client* mode (*ctest -S*).

While building a project for submission to CDash, CTest scans the build output for errors and warnings and reports them with surrounding context from the build log. This generic approach works for all build tools, but does not give details about the command invocation that produced a given problem. One may get more detailed reports by setting the *CTEST_USE_LAUNCHERS* variable:

```
set(CTEST_USE_LAUNCHERS 1)
```

in the **CTestConfig.cmake** file.

CTestCoverageCollectGCOV

Added in version 3.2.

This module provides the **ctest_coverage_collect_gcov** function.

This function runs gcov on all .gcda files found in the binary tree and packages the resulting .gcov files into a tar file. This tarball also contains the following:

- *data.json* defines the source and build directories for use by CDash.
- *Labels.json* indicates any *LABELS* that have been set on the source files.
- The *uncovered* directory holds any uncovered files found by *CTEST_EXTRA_COVERAGE_GLOB*.

After generating this tar file, it can be sent to CDash for display with the *ctest_submit*(*CDASH_UPLOAD*) command.

ctest_coverage_collect_gcov

```
ctest_coverage_collect_gcov(TARBALL <tarfile>
  [SOURCE <source_dir>][BUILD <build_dir>]
  [GCOV_COMMAND <gcov_command>]
  [GCOV_OPTIONS <options>...]
)
```

Run gcov and package a tar file for CDash. The options are:

TARBALL <tarfile>

Specify the location of the **.tar** file to be created for later upload to CDash. Relative paths will be interpreted with respect to the top-level build directory.

TARBALL_COMPRESSION <option>

Added in version 3.18.

Specify a compression algorithm for the **TARBALL** data file. Using this option reduces the size of the data file before it is submitted to CDash. **<option>** must be one of **GZIP**, **BZIP2**, **XZ**, **ZSTD**, **FROM_EXT**, or an expression that CMake evaluates as **FALSE**. The default value is **BZIP2**.

If **FROM_EXT** is specified, the resulting file will be compressed based on the file extension of the **<tarfile>** (i.e. **.tar.gz** will use **GZIP** compression). File extensions that will produce compressed output include **.tar.gz**, **.tgz**, **.tar.bzip2**, **.tbz**, **.tar.xz**, and **.txz**.

SOURCE <source_dir>

Specify the top-level source directory for the build. Default is the value of *CTEST_SOURCE_DIRECTORY*.

BUILD <build_dir>

Specify the top-level build directory for the build. Default is the value of *CTEST_BINARY_DIRECTORY*.

GCOV_COMMAND <gcov_command>

Specify the full path to the **gcov** command on the machine. Default is the value of *CTEST_COVERAGE_COMMAND*.

GCOV_OPTIONS <options>...

Specify options to be passed to gcov. The **gcov** command is run as **gcov <options>... -o <gcov-dir> <file>.gcda**. If not specified, the default option is just **-b -x**.

GLOB Added in version 3.6.

Recursively search for .gcda files in build_dir rather than determining search locations by reading TargetDirectories.txt.

DELETE

Added in version 3.6.

Delete coverage files after they've been packaged into the .tar.

QUIET

Suppress non-error messages that otherwise would have been printed out by this function.

Added in version 3.3: Added support for the *CTEST_CUSTOM_COVERAGE_EXCLUDE* variable.

CTestScriptMode

This file is read by ctest in script mode (-S)

CTestUseLaunchers

Set the *RULE_LAUNCH_** global properties when *CTEST_USE_LAUNCHERS* is on.

CTestUseLaunchers is automatically included when you include(CTest). However, it is split out into its own module file so projects can use the *CTEST_USE_LAUNCHERS* functionality independently.

To use launchers, set *CTEST_USE_LAUNCHERS* to ON in a *ctest -S* dashboard script, and then also set it in the cache of the configured project. Both cmake and ctest need to know the value of it for the launchers to work properly. CMake needs to know in order to generate proper build rules, and ctest, in order to produce the proper error and warning analysis.

For convenience, you may set the ENV variable *CTEST_USE_LAUNCHERS_DEFAULT* in your *ctest -S* script, too. Then, as long as your CMakeLists uses include(CTest) or include(CTestUseLaunchers), it will use the value of the ENV variable to initialize a *CTEST_USE_LAUNCHERS* cache variable. This cache variable initialization only occurs if *CTEST_USE_LAUNCHERS* is not already defined.

Added in version 3.8: If *CTEST_USE_LAUNCHERS* is on in a *ctest -S* script the *ctest_configure* command will add *-DCTEST_USE_LAUNCHERS:BOOL=TRUE* to the cmake command used to configure the project.

DeployQt4

Functions to help assemble a standalone Qt4 executable.

A collection of CMake utility functions useful for deploying Qt4 executables.

The following functions are provided by this module:

```
write_qt4_conf
resolve_qt4_paths
fixup_qt4_executable
install_qt4_plugin_path
install_qt4_plugin
install_qt4_executable
```

Requires CMake 2.6 or greater because it uses function and PARENT_SCOPE. Also depends on BundleUtilities.cmake.

```
write_qt4_conf(<qt_conf_dir> <qt_conf_contents>)
```

Writes a qt.conf file with the <qt_conf_contents> into <qt_conf_dir>.

```
resolve_qt4_paths(<paths_var> [<executable_path>])
```

Loop through <paths_var> list and if any don't exist resolve them relative to the <executable_path> (if supplied) or the CMAKE_INSTALL_PREFIX.

```
fixup_qt4_executable(<executable>
  [<qtplugins> <libs> <dirs> <plugins_dir> <request_qt_conf>])
```

Copies Qt plugins, writes a Qt configuration file (if needed) and fixes up a Qt4 executable using BundleUtilities so it is standalone and can be drag-and-drop copied to another machine as long as all of the system libraries are compatible.

<executable> should point to the executable to be fixed-up.

<qtplugins> should contain a list of the names or paths of any Qt plugins to be installed.

<libs> will be passed to BundleUtilities and should be a list of any already installed plugins, libraries or executables to also be fixed-up.

<dirs> will be passed to BundleUtilities and should contain and directories to be searched to find library dependencies.

<plugins_dir> allows an custom plugins directory to be used.

<request_qt_conf> will force a qt.conf file to be written even if not needed.

```
install_qt4_plugin_path(plugin executable copy installed_plugin_path_var
  <plugins_dir> <component> <configurations>)
```

Install (or copy) a resolved <plugin> to the default plugins directory (or <plugins_dir>) relative to <executable> and store the result in <installed_plugin_path_var>.

If <copy> is set to TRUE then the plugins will be copied rather than installed. This is to allow this module to be used at CMake time rather than install time.

If <component> is set then anything installed will use this COMPONENT.

```
install_qt4_plugin(plugin executable copy installed_plugin_path_var
  <plugins_dir> <component>)
```

Install (or copy) an unresolved <plugin> to the default plugins directory (or <plugins_dir>) relative to <executable> and store the result in <installed_plugin_path_var>. See documentation of INSTALL_QT4_PLUGIN_PATH.

```
install_qt4_executable(<executable>
  [<qtplugins> <libs> <dirs> <plugins_dir> <request_qt_conf> <component>])
```

Installs Qt plugins, writes a Qt configuration file (if needed) and fixes up a Qt4 executable using BundleUtilities so it is standalone and can be drag-and-drop copied to another machine as long as all of the system libraries are compatible. The executable will be fixed-up at install time. <component> is the COMPONENT used for bundle fixup and plugin installation. See documentation of FIXUP_QT4_BUNDLE.

ExternalData

Manage data files stored outside source tree

Introduction

Use this module to unambiguously reference data files stored outside the source tree and fetch them at build time from arbitrary local and remote content-addressed locations. Functions provided by this module recognize arguments with the syntax **DATA{<name>}** as references to external data, replace them with full paths to local copies of those data, and create build rules to fetch and update the local copies.

For example:

```
include(ExternalData)
set(ExternalData_URL_TEMPLATES "file:///local/%(algo)/%(hash)"
                                "file:///host/share/%(algo)/%(hash)"
                                "http://data.org/%(algo)/%(hash)")

ExternalData_Add_Test(MyData
  NAME MyTest
  COMMAND MyExe DATA{MyInput.png}
)
ExternalData_Add_Target(MyData)
```

When test **MyTest** runs the **DATA{MyInput.png}** argument will be replaced by the full path to a real instance of the data file **MyInput.png** on disk. If the source tree contains a content link such as **MyInput.png.md5** then the **MyData** target creates a real **MyInput.png** in the build tree.

Module Functions

ExternalData_Expand_Arguments

The **ExternalData_Expand_Arguments** function evaluates **DATA{}** references in its arguments and constructs a new list of arguments:

```
ExternalData_Expand_Arguments(
  <target>    # Name of data management target
  <outVar>    # Output variable
  [args...]  # Input arguments, DATA{} allowed
)
```

It replaces each **DATA{}** reference in an argument with the full path of a real data file on disk that will exist after the **<target>** builds.

ExternalData_Add_Test

The **ExternalData_Add_Test** function wraps around the CMake *add_test()* command but supports **DATA{}** references in its arguments:

```
ExternalData_Add_Test(
  <target>    # Name of data management target
  ...        # Arguments of add_test(), DATA{} allowed
)
```

It passes its arguments through **ExternalData_Expand_Arguments** and then invokes the *add_test()* command using the results.

Changed in version 3.31: If the arguments after **<target>** define a test with an executable that is a CMake target, empty values in the *TEST_LAUNCHER* and *CROSSCOMPILING_EMULATOR* properties of that target are preserved. See policy *CMP0178*.

ExternalData_Add_Target

The **ExternalData_Add_Target** function creates a custom target to manage local instances of data files stored externally:

```

ExternalData_Add_Target(
  <target>                # Name of data management target
  [SHOW_PROGRESS <ON|OFF>] # Show progress during the download
)

```

It creates custom commands in the target as necessary to make data files available for each **DATA{}** reference previously evaluated by other functions provided by this module. Data files may be fetched from one of the URL templates specified in the **ExternalData_URL_TEMPLATES** variable, or may be found locally in one of the paths specified in the **ExternalData_OBJECT_STORES** variable.

Added in version 3.20: The **SHOW_PROGRESS** argument may be passed to suppress progress information during the download of objects. If not provided, it defaults to **OFF** for *Ninja* and *Ninja Multi-Config* generators and **ON** otherwise.

Typically only one target is needed to manage all external data within a project. Call this function once at the end of configuration after all data references have been processed.

Module Variables

The following variables configure behavior. They should be set before calling any of the functions provided by this module.

ExternalData_BINARY_ROOT

The **ExternalData_BINARY_ROOT** variable may be set to the directory to hold the real data files named by expanded **DATA{}** references. The default is **CMAKE_BINARY_DIR**. The directory layout will mirror that of content links under **ExternalData_SOURCE_ROOT**.

ExternalData_CUSTOM_SCRIPT_<key>

Added in version 3.2.

Specify a full path to a **.cmake** custom fetch script identified by <key> in entries of the **ExternalData_URL_TEMPLATES** list. See *Custom Fetch Scripts*.

ExternalData_HTTPHEADERS

Added in version 4.0.

The **ExternalData_HTTPHEADERS** variable may be used to supply a list of headers, each element containing one header with the form **Key: Value**. See the *file(DO WNLOAD)* command's **HTTPHEADER** option.

ExternalData_LINK_CONTENT

The **ExternalData_LINK_CONTENT** variable may be set to the name of a supported hash algorithm to enable automatic conversion of real data files referenced by the **DATA{}** syntax into content links. For each such <file> a content link named <file><ext> is created. The original file is renamed to the form **.ExternalData_<algo>_<hash>** to stage it for future transmission to one of the locations in the list of URL templates (by means outside the scope of this module). The data fetch rule created for the content link will use the staged object if it cannot be found using any URL template.

ExternalData_NO_SYMLINKS

Added in version 3.3.

The real data files named by expanded **DATA{}** references may be made available under **ExternalData_BINARY_ROOT** using symbolic links on some platforms. The

ExternalData_NO_SYMLINKS variable may be set to disable use of symbolic links and enable use of copies instead.

ExternalData_OBJECT_STORES

The **ExternalData_OBJECT_STORES** variable may be set to a list of local directories that store objects using the layout `<dir>/%(algo)/%(hash)`. These directories will be searched first for a needed object. If the object is not available in any store then it will be fetched remotely using the URL templates and added to the first local store listed. If no stores are specified the default is a location inside the build tree.

ExternalData_SERIES_PARSE

ExternalData_SERIES_PARSE_PREFIX

ExternalData_SERIES_PARSE_NUMBER

ExternalData_SERIES_PARSE_SUFFIX

ExternalData_SERIES_MATCH

See *Referencing File Series*.

ExternalData_SOURCE_ROOT

The **ExternalData_SOURCE_ROOT** variable may be set to the highest source directory containing any path named by a **DATA{}** reference. The default is **CMAKE_SOURCE_DIR**. **ExternalData_SOURCE_ROOT** and **CMAKE_SOURCE_DIR** must refer to directories within a single source distribution (e.g. they come together in one tarball).

ExternalData_TIMEOUT_ABSOLUTE

The **ExternalData_TIMEOUT_ABSOLUTE** variable sets the download absolute timeout, in seconds, with a default of **300** seconds. Set to **0** to disable enforcement.

ExternalData_TIMEOUT_INACTIVITY

The **ExternalData_TIMEOUT_INACTIVITY** variable sets the download inactivity timeout, in seconds, with a default of **60** seconds. Set to **0** to disable enforcement.

ExternalData_URL_ALGO_<algo>_<key>

Added in version 3.3.

Specify a custom URL component to be substituted for URL template placeholders of the form **%(algo:<key>)**, where **<key>** is a valid C identifier, when fetching an object referenced via hash algorithm **<algo>**. If not defined, the default URL component is just **<algo>** for any **<key>**.

ExternalData_URL_TEMPLATES

The **ExternalData_URL_TEMPLATES** may be set to provide a list of URL templates using the placeholders **%(algo)** and **%(hash)** in each template. Data fetch rules try each URL template in order by substituting the hash algorithm name for **%(algo)** and the hash value for **%(hash)**. Alternatively one may use **%(algo:<key>)** with **ExternalData_URL_ALGO_<algo>_<key>** variables to gain more flexibility in remote URLs.

Referencing Files

Referencing Single Files

The **DATA{}** syntax is literal and the **<name>** is a full or relative path within the source tree. The source tree must contain either a real data file at **<name>** or a "content link" at **<name><ext>** containing a hash of the real file using a hash algorithm corresponding to **<ext>**. For example, the argument **DATA{img.png}** may be satisfied by either a real **img.png** file in the current source directory or a **img.png.md5** file containing its MD5 sum.

Added in version 3.8: Multiple content links of the same name with different hash algorithms are supported (e.g. **img.png.sha256** and **img.png.sha1**) so long as they all correspond to the same real file. This allows objects to be fetched from sources indexed by different hash algorithms.

Referencing File Series

The **DATA{}** syntax can be told to fetch a file series using the form **DATA{<name>,;}**, where the **:** is literal. If the source tree contains a group of files or content links named like a series then a reference to one member adds rules to fetch all of them. Although all members of a series are fetched, only the file originally named by the **DATA{}** argument is substituted for it. The default configuration recognizes file series names ending with **#.ext**, **_.ext**, **#.ext**, or **-#.ext** where **#** is a sequence of decimal digits and **.ext** is any single extension. Configure it with a regex that parses **<number>** and **<suffix>** parts from the end of **<name>**:

ExternalData_SERIES_PARSE – regex of the form **(<number>)(<suffix>)\$**.

For more complicated cases set:

- **ExternalData_SERIES_PARSE** – regex with at least two **()** groups.
- **ExternalData_SERIES_PARSE_PREFIX** – regex group number of the **<prefix>**, if any.
- **ExternalData_SERIES_PARSE_NUMBER** – regex group number of the **<number>**.
- **ExternalData_SERIES_PARSE_SUFFIX** – regex group number of the **<suffix>**.

Configure series number matching with a regex that matches the **<number>** part of series members named **<prefix><number><suffix>**:

ExternalData_SERIES_MATCH – regex matching **<number>** in all series members

Note that the **<suffix>** of a series does not include a hash–algorithm extension.

Referencing Associated Files

The **DATA{}** syntax can alternatively match files associated with the named file and contained in the same directory. Associated files may be specified by options using the syntax **DATA{<name>,<opt1>,<opt2>,...}**. Each option may specify one file by name or specify a regular expression to match file names using the syntax **REGEX:<regex>**. For example, the arguments:

```
DATA{MyData/MyInput.mhd,MyInput.img}           # File pair
DATA{MyData/MyFrames00.png,REGEX:MyFrames[0-9]+\..png} # Series
```

will pass **MyInput.mha** and **MyFrames00.png** on the command line but ensure that the associated files are present next to them.

Referencing Directories

The **DATA{}** syntax may reference a directory using a trailing slash and a list of associated files. The form **DATA{<name>/,<opt1>,<opt2>,...}** adds rules to fetch any files in the directory that match one of the associated file options. For example, the argument **DATA{MyDataDir/,REGEX:.*}** will pass the full path to a **MyDataDir** directory on the command line and ensure that the directory contains files corresponding to every file or content link in the **MyDataDir** source directory.

Added in version 3.3: In order to match associated files in subdirectories, specify a **RECURSE:** option, e.g. **DATA{MyDataDir/,RECURSE:.,REGEX:.*}**.

Hash Algorithms

The following hash algorithms are supported:

%(algo)	<ext>	Description
MD5	.md5	Message-Digest Algorithm 5, RFC 1321
SHA1	.sha1	US Secure Hash Algorithm 1, RFC 3174
SHA224	.sha224	US Secure Hash Algorithms, RFC 4634
SHA256	.sha256	US Secure Hash Algorithms, RFC 4634
SHA384	.sha384	US Secure Hash Algorithms, RFC 4634
SHA512	.sha512	US Secure Hash Algorithms, RFC 4634
SHA3_224	.sha3-224	Keccak SHA-3
SHA3_256	.sha3-256	Keccak SHA-3
SHA3_384	.sha3-384	Keccak SHA-3
SHA3_512	.sha3-512	Keccak SHA-3

Added in version 3.8: Added the **SHA3_*** hash algorithms.

Note that the hashes are used only for unique data identification and download verification.

Custom Fetch Scripts

Added in version 3.2.

When a data file must be fetched from one of the URL templates specified in the **ExternalData_URL_TEMPLATES** variable, it is normally downloaded using the *file(DOWNLOAD)* command. One may specify usage of a custom fetch script by using a URL template of the form **ExternalDataCustomScript://<key>/<loc>**. The **<key>** must be a C identifier, and the **<loc>** must contain the **%(algo)** and **%(hash)** placeholders. A variable corresponding to the key, **ExternalData_CUSTOM_SCRIPT_<key>**, must be set to the full path to a **.cmake** script file. The script will be included to perform the actual fetch, and provided with the following variables:

ExternalData_CUSTOM_LOCATION

When a custom fetch script is loaded, this variable is set to the location part of the URL, which will contain the substituted hash algorithm name and content hash value.

ExternalData_CUSTOM_FILE

When a custom fetch script is loaded, this variable is set to the full path to a file in which the script must store the fetched content. The name of the file is unspecified and should not be interpreted in any way.

The custom fetch script is expected to store fetched content in the file or set a variable:

ExternalData_CUSTOM_ERROR

When a custom fetch script fails to fetch the requested content, it must set this variable to a short one-line message describing the reason for failure.

ExternalProject

External Project Definition

ExternalProject_Add

The **ExternalProject_Add()** function creates a custom target to drive download, update/patch, configure, build, install and test steps of an external project:

```
ExternalProject_Add(<name> [<option>...])
```

The individual steps within the process can be driven independently if required (e.g. for CDash submission) and extra custom steps can be defined, along with the ability to control the step dependencies. The directory structure used for the management of the external project can also be customized. The function supports a large number of options which can be used to tailor the external project behavior.

Directory Options

Most of the time, the default directory layout is sufficient. It is largely an implementation detail that the main project usually doesn't need to change. In some circumstances, however, control over the directory layout can be useful or necessary. The directory options are potentially more useful from the point of view that the main build can use the *ExternalProject_Get_Property()* command to retrieve their values, thereby allowing the main project to refer to build artifacts of the external project.

PREFIX <dir>

Root directory for the external project. Unless otherwise noted below, all other directories associated with the external project will be created under here.

TMP_DIR <dir>

Directory in which to store temporary files.

STAMP_DIR <dir>

Directory in which to store the timestamps of each step. Log files from individual steps are also created in here unless overridden by LOG_DIR (see *Logging Options* below).

LOG_DIR <dir>

Added in version 3.14.

Directory in which to store the logs of each step.

DOWNLOAD_DIR <dir>

Directory in which to store downloaded files before unpacking them. This directory is only used by the URL download method, all other download methods use **SOURCE_DIR** directly instead.

SOURCE_DIR <dir>

Source directory into which downloaded contents will be unpacked, or for non-URL download methods, the directory in which the repository should be checked out, cloned, etc. If no download method is specified, this must point to an existing directory where the external project has already been unpacked or cloned/checked out.

NOTE:

If a download method is specified, any existing contents of the source directory may be deleted. Only the URL download method checks whether this directory is either missing or empty before initiating the download, stopping with an error if it is not empty. All other download methods silently discard any previous contents of the source directory.

BINARY_DIR <dir>

Specify the build directory location. This option is ignored if **BUILD_IN_SOURCE** is enabled.

INSTALL_DIR <dir>

Installation prefix to be placed in the **<INSTALL_DIR>** placeholder. This does not actually configure the external project to install to the given prefix. That must be done by passing appropriate arguments to the external project configuration step, e.g. using **<INSTALL_DIR>**.

If any of the above **..._DIR** options are not specified, their defaults are computed as follows. If the **PREFIX** option is given or the **EP_PREFIX** directory property is set, then an external project is built and installed under the specified prefix:

```

TMP_DIR      = <prefix>/tmp
STAMP_DIR    = <prefix>/src/<name>-stamp
DOWNLOAD_DIR = <prefix>/src
SOURCE_DIR   = <prefix>/src/<name>
BINARY_DIR   = <prefix>/src/<name>-build
INSTALL_DIR  = <prefix>
LOG_DIR      = <STAMP_DIR>

```

Otherwise, if the **EP_BASE** directory property is set then components of an external project are stored under the specified base:

```

TMP_DIR      = <base>/tmp/<name>
STAMP_DIR    = <base>/Stamp/<name>
DOWNLOAD_DIR = <base>/Download/<name>
SOURCE_DIR   = <base>/Source/<name>
BINARY_DIR   = <base>/Build/<name>
INSTALL_DIR  = <base>/Install/<name>
LOG_DIR      = <STAMP_DIR>

```

If no **PREFIX**, **EP_PREFIX**, or **EP_BASE** is specified, then the default is to set **PREFIX** to **<name>-prefix**. Relative paths are interpreted with respect to **CMAKE_CURRENT_BINARY_DIR** at the point where **ExternalProject_Add()** is called.

Download Step Options

A download method can be omitted if the **SOURCE_DIR** option is used to point to an existing non-empty directory. Otherwise, one of the download methods below must be specified (multiple download methods should not be given) or a custom **DOWNLOAD_COMMAND** provided.

DOWNLOAD_COMMAND <cmd>...

Overrides the command used for the download step (*generator expressions* are supported). If this option is specified, all other download options will be ignored. Providing an empty string for <cmd> effectively disables the download step.

URL

URL <url1> [<url2>...]

List of paths and/or URL(s) of the external project's source. When more than one URL is given, they are tried in turn until one succeeds. A URL may be an ordinary path in the local file system (in which case it must be the only URL provided) or any downloadable URL supported by the *file(DOWNLOAD)* command. A local filesystem path may refer to either an existing directory or to an archive file, whereas a URL is expected to point to a file which can be treated as an archive. When an archive is used, it will be unpacked automatically unless the **DOWNLOAD_NO_EXTRACT** option is set to prevent it. The archive type is determined by inspecting the actual content rather than using logic based on the file extension.

Changed in version 3.7: Multiple URLs are allowed.

URL_HASH <algo>=<hashValue>

Hash of the archive file to be downloaded. The argument should be of the form <algo>=<hash-Value> where **algo** can be any of the hashing algorithms supported by the *file()* command. Specifying this option is strongly recommended for URL downloads, as it ensures the integrity of the downloaded content. It is also used as a check for a previously downloaded file, allowing connection to the remote location to be avoided altogether if the local directory already has a file from an earlier download that matches the specified hash.

URL_MD5 <md5>

Equivalent to **URL_HASH MD5=<md5>**.

DOWNLOAD_NAME <fname>

File name to use for the downloaded file. If not given, the end of the URL is used to determine the file name. This option is rarely needed, the default name is generally suitable and is not normally used outside of code internal to the **ExternalProject** module.

DOWNLOAD_EXTRACT_TIMESTAMP <bool>

Added in version 3.24.

When specified with a true value, the timestamps of the extracted files will match those in the archive. When false, the timestamps of the extracted files will reflect the time at which the extraction was performed. If the download URL changes, timestamps based off those in the archive can result in dependent targets not being rebuilt when they potentially should have been. Therefore, unless the file timestamps are significant to the project in some way, use a false value for this option. If **DOWNLOAD_EXTRACT_TIMESTAMP** is not given, the default is false. See policy *CMP0135*.

DOWNLOAD_NO_EXTRACT <bool>

Added in version 3.6.

Allows the extraction part of the download step to be disabled by passing a boolean true value for this option. If this option is not given, the downloaded contents will be unpacked automatically if required. If extraction has been disabled, the full path to the downloaded file is available as **<DOWNLOADED_FILE>** in subsequent steps or as the property **DOWNLOADED_FILE** with the *ExternalProject_Get_Property()* command.

DOWNLOAD_NO_PROGRESS <bool>

Can be used to disable logging the download progress. If this option is not given, download progress messages will be logged.

TIMEOUT <seconds>

Maximum time allowed for file download operations.

INACTIVITY_TIMEOUT <seconds>

Added in version 3.19.

Terminate the operation after a period of inactivity.

HTTP_USERNAME <username>

Added in version 3.7.

Username for the download operation if authentication is required.

HTTP_PASSWORD <password>

Added in version 3.7.

Password for the download operation if authentication is required.

HTTP_HEADER <header1> [<header2>...]

Added in version 3.7.

Provides an arbitrary list of HTTP headers for the download operation. This can be useful for

accessing content in systems like AWS, etc.

TLS_VERSION <min>

Added in version 3.30.

Specify minimum TLS version for **https://** URLs. If this option is not provided, the value of the *CMAKE_TLS_VERSION* variable or the *CMAKE_TLS_VERSION* environment variable will be used instead (see *file(DOWNLOAD)*).

This option also applies to **git clone** invocations, although the default behavior is different. If none of the **TLS_VERSION** option, *CMAKE_TLS_VERSION* variable, or *CMAKE_TLS_VERSION* environment variable is specified, the behavior will be determined by git's default or a **http.sslVersion** git config option the user may have set at a global level.

TLS_VERIFY <bool>

Specifies whether certificate verification should be performed for **https://** URLs. If this option is not provided, the value of the *CMAKE_TLS_VERIFY* variable or the *CMAKE_TLS_VERIFY* environment variable will be used instead (see *file(DOWNLOAD)*). If neither of those is set, certificate verification will not be performed. In situations where **URL_HASH** cannot be provided, this option can be an alternative verification measure.

This option also applies to **git clone** invocations, although the default behavior is different. If none of the **TLS_VERIFY** option, *CMAKE_TLS_VERIFY* variable, or *CMAKE_TLS_VERIFY* environment variable is specified, the behavior will be determined by git's default (true) or a **http.sslVerify** git config option the user may have set at a global level.

Changed in version 3.6: Previously this option did not apply to **git clone** invocations.

Changed in version 3.30: Previously the *CMAKE_TLS_VERIFY* environment variable was not checked.

TLS_CAINFO <file>

Specify a custom certificate authority file to use if **TLS_VERIFY** is enabled. If this option is not specified, the value of the *CMAKE_TLS_CAINFO* variable will be used instead (see *file(DOWNLOAD)*).

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 (see *file(DOWNLOAD)*). 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 (see *file(DOWNLOAD)*)

Added in version 3.1: Added support for *tbz2*, *.tar.xz*, *.txz*, and *.7z* extensions.

Git

NOTE: A git version of 1.6.5 or later is required if this download method is used.

GIT_REPOSITORY <url>

URL of the git repository. Any URL understood by the **git** command may be used.

Changed in version 3.27: A relative URL will be resolved based on the parent project's remote, subject to *CMP0150*. See the policy documentation for how the remote is selected, including conditions where the remote selection can fail. Local filesystem remotes should always use absolute paths.

GIT_TAG <tag>

Git branch name, tag or commit hash. Note that branch names and tags should generally be specified as remote names (i.e. **origin/myBranch** rather than simply **myBranch**). This ensures that if the remote end has its tag moved or branch rebased or history rewritten, the local clone will still be updated correctly. In general, however, specifying a commit hash should be preferred for a number of reasons:

- If the local clone already has the commit corresponding to the hash, no **git fetch** needs to be performed to check for changes each time CMake is re-run. This can result in a significant speed up if many external projects are being used.
- Using a specific git hash ensures that the main project's own history is fully traceable to a specific point in the external project's evolution. If a branch or tag name is used instead, then checking out a specific commit of the main project doesn't necessarily pin the whole build to a specific point in the life of the external project. The lack of such deterministic behavior makes the main project lose traceability and repeatability.

If **GIT_SHALLOW** is enabled then **GIT_TAG** works only with branch names and tags. A commit hash is not allowed.

Note that if not provided, **GIT_TAG** defaults to **master**, not the default Git branch name.

GIT_REMOTE_NAME <name>

The optional name of the remote. If this option is not specified, it defaults to **origin**.

GIT_SUBMODULES <module>...

Specific git submodules that should also be updated. If this option is not provided, all git submodules will be updated.

Changed in version 3.16: When *CMP0097* is set to **NEW**, if this value is set to an empty string then no submodules are initialized or updated.

GIT_SUBMODULES_RECURSE <bool>

Added in version 3.17.

Specify whether git submodules (if any) should update recursively by passing the **--recursive** flag to **git submodule update**. If not specified, the default is on.

GIT_SHALLOW <bool>

Added in version 3.6.

When this option is enabled, the **git clone** operation will be given the **--depth 1** option. This performs a shallow clone, which avoids downloading the whole history and instead retrieves just the commit denoted by the **GIT_TAG** option.

GIT_PROGRESS <bool>

Added in version 3.8.

When enabled, this option instructs the **git clone** operation to report its progress by passing it the **--progress** option. Without this option, the clone step for large projects may appear to make the build stall, since nothing will be logged until the clone operation finishes. While this option can be used to provide progress to prevent the appearance of the build having stalled, it may also make the build overly noisy if lots of external projects are used.

GIT_CONFIG <option1> [<option2>...]

Added in version 3.8.

Specify a list of config options to pass to **git clone**. Each option listed will be transformed into its own **--config <option>** on the **git clone** command line, with each option required to be in the form **key=value**.

GIT_REMOTE_UPDATE_STRATEGY <strategy>

Added in version 3.18.

When **GIT_TAG** refers to a remote branch, this option can be used to specify how the update step behaves. The <strategy> must be one of the following:

CHECKOUT

Ignore the local branch and always checkout the branch specified by **GIT_TAG**.

REBASE

Try to rebase the current branch to the one specified by **GIT_TAG**. If there are local uncommitted changes, they will be stashed first and popped again after rebasing. If rebasing or popping stashed changes fail, abort the rebase and halt with an error. When **GIT_REMOTE_UPDATE_STRATEGY** is not present, this is the default strategy unless the default has been overridden with **CMAKE_EP_GIT_REMOTE_UPDATE_STRATEGY** (see below). Note that if the branch specified in **GIT_TAG** is different to the upstream branch currently being tracked, it is not safe to perform a rebase. In that situation, **REBASE** will silently be treated as **CHECKOUT** instead.

REBASE_CHECKOUT

Same as **REBASE** except if the rebase fails, an annotated tag will be created at the original **HEAD** position from before the rebase and then checkout **GIT_TAG** just like the **CHECKOUT** strategy. The message stored on the annotated tag will give information about what was attempted and the tag name will include a timestamp so that each failed run will add a new tag. This strategy ensures no changes will be lost, but updates should always succeed if **GIT_TAG** refers to a valid ref unless there are uncommitted changes that cannot be popped successfully.

The variable **CMAKE_EP_GIT_REMOTE_UPDATE_STRATEGY** can be set to override the default strategy. This variable should not be set by a project, it is intended for the user to set. It is primarily intended for use in continuous integration scripts to ensure that when history is rewritten

on a remote branch, the build doesn't end up with unintended changes or failed builds resulting from conflicts during rebase operations.

Subversion

SVN_REPOSITORY <url>

URL of the Subversion repository.

SVN_REVISION -r<rev>

Revision to checkout from the Subversion repository.

SVN_USERNAME <username>

Username for the Subversion checkout and update.

SVN_PASSWORD <password>

Password for the Subversion checkout and update.

SVN_TRUST_CERT <bool>

Specifies whether to trust the Subversion server site certificate. If enabled, the `--trust-server-cert` option is passed to the `svn` checkout and update commands.

Mercurial

HG_REPOSITORY <url>

URL of the mercurial repository.

HG_TAG <tag>

Mercurial branch name, tag or commit id.

CVS

CVS_REPOSITORY <cvsroot>

CVSROOT of the CVS repository.

CVS_MODULE <mod>

Module to checkout from the CVS repository.

CVS_TAG <tag>

Tag to checkout from the CVS repository.

Update Step Options

Whenever CMake is re-run, by default the external project's sources will be updated if the download method supports updates (e.g. a git repository would be checked if the **GIT_TAG** does not refer to a specific commit).

UPDATE_COMMAND <cmd>...

Overrides the download method's update step with a custom command. The command may use *generator expressions*.

UPDATE_DISCONNECTED <bool>

Added in version 3.2.

When enabled, this option causes the update step to be skipped (but see below for changed behavior where this is not the case). It does not prevent the download step. The update step can still be added as a step target (see `ExternalProject_Add_StepTargets()`) and called manually. This is useful if you want to allow developers to build the project when disconnected from the network (the network may still be needed for the download step though).

Changed in version 3.27: When **UPDATE_DISCONNECTED** is true, the update step will be executed if any details about the update or download step are changed. Furthermore, if using the git download/update method, the update logic will be modified to skip attempts to contact the remote. If the **GIT_TAG** mentions a ref that is not known locally, the update step will halt with a fatal error.

When this option is present, it is generally advisable to make the value a cache variable under the developer's control rather than hard-coding it. If this option is not present, the default value is taken from the **EP_UPDATE_DISCONNECTED** directory property. If that is also not defined, updates are performed as normal. The **EP_UPDATE_DISCONNECTED** directory property is intended as a convenience for controlling the **UPDATE_DISCONNECTED** behavior for an entire section of a project's directory hierarchy and may be a more convenient method of giving developers control over whether or not to perform updates (assuming the project also provides a cache variable or some other convenient method for setting the directory property).

This may cause a step target to be created automatically for the **download** step. See policy *CMP0114*.

Patch Step Options

PATCH_COMMAND <cmd>...

Specifies a custom command to patch the sources after an update. By default, no patch command is defined. Note that it can be quite difficult to define an appropriate patch command that performs robustly, especially for download methods such as git where changing the **GIT_TAG** will not discard changes from a previous patch, but the patch command will be called again after updating to the new tag.

Configure Step Options

The configure step is run after the download and update steps. By default, the external project is assumed to be a CMake project, but this can be overridden if required.

CONFIGURE_COMMAND <cmd>...

The default configure command runs CMake with a few options based on the main project. The options added are typically only those needed to use the same generator as the main project, but the **CMAKE_GENERATOR** option can be given to override this. The project is responsible for adding any toolchain details, flags or other settings it wants to reuse from the main project or otherwise specify (see **CMAKE_ARGS**, **CMAKE_CACHE_ARGS** and **CMAKE_CACHE_DEFAULT_ARGS** below).

For non-CMake external projects, the **CONFIGURE_COMMAND** option must be used to override the default configure command (*generator expressions* are supported). For projects that require no configure step, specify this option with an empty string as the command to execute.

CMAKE_COMMAND /.../cmake

Specify an alternative cmake executable for the configure step (use an absolute path). This is generally not recommended, since it is usually desirable to use the same CMake version throughout the whole build. This option is ignored if a custom configure command has been specified with **CONFIGURE_COMMAND**.

CMAKE_GENERATOR <gen>

Override the CMake generator used for the configure step. Without this option, the same generator as the main build will be used. This option is ignored if a custom configure command has been specified with the **CONFIGURE_COMMAND** option.

CMAKE_GENERATOR_PLATFORM <platform>

Added in version 3.1.

Pass a generator-specific platform name to the CMake command (see *CMAKE_GENERATOR_PLATFORM*). It is an error to provide this option without the **CMAKE_GENERATOR** option.

CMAKE_GENERATOR_TOOLSET <toolset>

Pass a generator-specific toolset name to the CMake command (see *CMAKE_GENERATOR_TOOLSET*). It is an error to provide this option without the **CMAKE_GENERATOR** option.

CMAKE_GENERATOR_INSTANCE <instance>

Added in version 3.11.

Pass a generator-specific instance selection to the CMake command (see *CMAKE_GENERATOR_INSTANCE*). It is an error to provide this option without the **CMAKE_GENERATOR** option.

CMAKE_ARGS <arg>...

The specified arguments are passed to the **cmake** command line. They can be any argument the **cmake** command understands, not just cache values defined by **-D...** arguments (see also *CMake Options*).

Added in version 3.3: Arguments may use *generator expressions*.

CMAKE_CACHE_ARGS <arg>...

This is an alternate way of specifying cache variables where command line length issues may become a problem. The arguments are expected to be in the form **-Dvar:STRING=value**, which are then transformed into CMake *set()* commands with the **FORCE** option used. These *set()* commands are written to a pre-load script which is then applied using the *cmake -C* command line option.

Added in version 3.3: Arguments may use *generator expressions*.

CMAKE_CACHE_DEFAULT_ARGS <arg>...

Added in version 3.2.

This is the same as the **CMAKE_CACHE_ARGS** option except the *set()* commands do not include the **FORCE** keyword. This means the values act as initial defaults only and will not override any variables already set from a previous run. Use this option with care, as it can lead to different behavior depending on whether the build starts from a fresh build directory or reuses previous build contents.

Added in version 3.15: If the CMake generator is the **Green Hills MULTI** and not overridden, the original project's settings for the GHS toolset and target system customization cache variables are propagated into the external project.

SOURCE_SUBDIR <dir>

Added in version 3.7.

When no **CONFIGURE_COMMAND** option is specified, the configure step assumes the external project has a **CMakeLists.txt** file at the top of its source tree (i.e. in **SOURCE_DIR**). The **SOURCE_SUBDIR** option can be used to point to an alternative directory within the source tree to use as the top of the CMake source tree instead. This must be a relative path and it will be interpreted as being relative to **SOURCE_DIR**.

Added in version 3.14: When **BUILD_IN_SOURCE** option is enabled, the **BUILD_COMMAND** is used to point to an alternative directory within the source tree.

CONFIGURE_HANDLED_BY_BUILD <bool>

Added in version 3.20.

Enabling this option relaxes the dependencies of the configure step on other external projects to order-only. This means the configure step will be executed after its external project dependencies are built but it will not be marked dirty when one of its external project dependencies is rebuilt. This option can be enabled when the build step is smart enough to figure out if the configure step needs to be rerun. CMake and Meson are examples of build systems whose build step is smart enough to know if the configure step needs to be rerun.

Build Step Options

If the configure step assumed the external project uses CMake as its build system, the build step will also. Otherwise, the build step will assume a Makefile-based build and simply run **make** with no arguments as the default build step. This can be overridden with custom build commands if required.

If both the main project and the external project use make as their build tool, the build step of the external project is invoked as a recursive make using **\$(MAKE)**. This will communicate some build tool settings from the main project to the external project. If either the main project or external project is not using make, no build tool settings will be passed to the external project other than those established by the configure step (i.e. running **ninja -v** in the main project will not pass **-v** to the external project's build step, even if it also uses **ninja** as its build tool).

BUILD_COMMAND <cmd>...

Overrides the default build command (*generator expressions* are supported). If this option is not given, the default build command will be chosen to integrate with the main build in the most appropriate way (e.g. using recursive **make** for Makefile generators or **cmake --build** if the project uses a CMake build). This option can be specified with an empty string as the command to make the build step do nothing.

BUILD_IN_SOURCE <bool>

When this option is enabled, the build will be done directly within the external project's source tree. This should generally be avoided, the use of a separate build directory is usually preferred, but it can be useful when the external project assumes an in-source build. The **BINARY_DIR** option should not be specified if building in-source.

BUILD_ALWAYS <bool>

Enabling this option forces the build step to always be run. This can be the easiest way to robustly ensure that the external project's own build dependencies are evaluated rather than relying on the default success timestamp-based method. This option is not normally needed unless developers are expected to modify something the external project's build depends on in a way that is not detectable via the step target dependencies (e.g. **SOURCE_DIR** is used without a download method and developers might modify the sources in **SOURCE_DIR**).

BUILD_BYPRODUCTS <file>...

Added in version 3.2.

Specifies files that will be generated by the build command but which might or might not have their modification time updated by subsequent builds. This may also be required to explicitly declare dependencies when using the *Ninja* generator. These ultimately get passed through as **BYPRODUCTS** to the build step's own underlying call to *add_custom_command()*, which has additional documentation.

BUILD_JOB_SERVER_AWARE <bool>

Added in version 3.28.

Specifies that the build step is aware of the GNU Make job server. See the *add_custom_command()* documentation of its **JOB_SERVER_AWARE** option for details. This option is relevant only when an explicit **BUILD_COMMAND** is specified.

Install Step Options

If the configure step assumed the external project uses CMake as its build system, the install step will also. Otherwise, the install step will assume a Makefile-based build and simply run **make install** as the default build step. This can be overridden with custom install commands if required.

INSTALL_COMMAND <cmd>...

The external project's own install step is invoked as part of the main project's *build*. It is done after the external project's build step and may be before or after the external project's test step (see the **TEST_BEFORE_INSTALL** option below). The external project's install rules are not part of the main project's install rules, so if anything from the external project should be installed as part of the main build, these need to be specified in the main build as additional *install()* commands. The default install step builds the **install** target of the external project, but this can be overridden with a custom command using this option (*generator expressions* are supported). Passing an empty string as the <cmd> makes the install step do nothing.

INSTALL_BYPRODUCTS <file>...

Added in version 3.26.

Specifies files that will be generated by the install command but which might or might not have their modification time updated by subsequent installs. This may also be required to explicitly declare dependencies when using the *Ninja* generator. These ultimately get passed through as **BYPRODUCTS** to the install step's own underlying call to *add_custom_command()*, which has additional documentation.

INSTALL_JOB_SERVER_AWARE <bool>

Added in version 4.0.

Specifies that the install step is aware of the GNU Make job server. See the *add_custom_command()* documentation of its **JOB_SERVER_AWARE** option for details. This option is relevant only when an explicit **INSTALL_COMMAND** is specified.

NOTE:

If the *CMAKE_INSTALL_MODE* environment variable is set when the main project is built, it will only have an effect if the following conditions are met:

- The main project's configure step assumed the external project uses CMake as its build system.
- The external project's install command actually runs. Note that due to the way **ExternalProject** may use timestamps internally, if nothing the install step depends on needs to be re-executed, the install command might also not need to run.

Note also that **ExternalProject** does not check whether the *CMAKE_INSTALL_MODE* environment variable changes from one run to another.

Test Step Options

The test step is only defined if at least one of the following **TEST_...** options are provided.

TEST_COMMAND <cmd>...

Overrides the default test command (*generator expressions* are supported). If this option is not given, the default behavior of the test step is to build the external project's own **test** target. This option can be specified with <cmd> as an empty string, which allows the test step to still be defined, but it will do nothing. Do not specify any of the other **TEST_...** options if providing an empty string as the test command, but prefer to omit all **TEST_...** options altogether if the test step target

is not needed.

TEST_BEFORE_INSTALL <bool>

When this option is enabled, the test step will be executed before the install step. The default behavior is for the test step to run after the install step.

TEST_AFTER_INSTALL <bool>

This option is mainly useful as a way to indicate that the test step is desired but all default behavior is sufficient. Specifying this option with a boolean true value ensures the test step is defined and that it comes after the install step. If both **TEST_BEFORE_INSTALL** and **TEST_AFTER_INSTALL** are enabled, the latter is silently ignored.

TEST_EXCLUDE_FROM_MAIN <bool>

Added in version 3.2.

If enabled, the main build's default ALL target will not depend on the test step. This can be a useful way of ensuring the test step is defined but only gets invoked when manually requested. This may cause a step target to be created automatically for either the **install** or **build** step. See policy *CMP0114*.

Output Logging Options

Each of the following **LOG_...** options can be used to wrap the relevant step in a script to capture its output to files. The log files will be created in **LOG_DIR** if supplied or otherwise the **STAMP_DIR** directory with step-specific file names.

LOG_DOWNLOAD <bool>

When enabled, the output of the download step is logged to files.

LOG_UPDATE <bool>

When enabled, the output of the update step is logged to files.

LOG_PATCH <bool>

Added in version 3.14.

When enabled, the output of the patch step is logged to files.

LOG_CONFIGURE <bool>

When enabled, the output of the configure step is logged to files.

LOG_BUILD <bool>

When enabled, the output of the build step is logged to files.

LOG_INSTALL <bool>

When enabled, the output of the install step is logged to files.

LOG_TEST <bool>

When enabled, the output of the test step is logged to files.

LOG_MERGED_STDOUTERR <bool>

Added in version 3.14.

When enabled, stdout and stderr will be merged for any step whose output is being logged to files.

LOG_OUTPUT_ON_FAILURE <bool>

Added in version 3.14.

This option only has an effect if at least one of the other **LOG_<step>** options is enabled. If an error occurs for a step which has logging to file enabled, that step's output will be printed to the

console if **LOG_OUTPUT_ON_FAILURE** is set to true. For cases where a large amount of output is recorded, just the end of that output may be printed to the console.

Terminal Access Options

Added in version 3.4.

Steps can be given direct access to the terminal in some cases. Giving a step access to the terminal may allow it to receive terminal input if required, such as for authentication details not provided by other options. With the *Ninja* generator, these options place the steps in the **console job pool**. Each step can be given access to the terminal individually via the following options:

USES_TERMINAL_DOWNLOAD <bool>

Give the download step access to the terminal.

USES_TERMINAL_UPDATE <bool>

Give the update step access to the terminal.

USES_TERMINAL_PATCH <bool>

Added in version 3.23.

Give the patch step access to the terminal.

USES_TERMINAL_CONFIGURE <bool>

Give the configure step access to the terminal.

USES_TERMINAL_BUILD <bool>

Give the build step access to the terminal.

USES_TERMINAL_INSTALL <bool>

Give the install step access to the terminal.

USES_TERMINAL_TEST <bool>

Give the test step access to the terminal.

Target Options

DEPENDS <targets>...

Specify other targets on which the external project depends. The other targets will be brought up to date before any of the external project's steps are executed. Because the external project uses additional custom targets internally for each step, the **DEPENDS** option is the most convenient way to ensure all of those steps depend on the other targets. Simply doing `add_dependencies(<name> <targets>)` will not make any of the steps dependent on <targets>.

EXCLUDE_FROM_ALL <bool>

When enabled, this option excludes the external project from the default ALL target of the main build.

STEP_TARGETS <step-target>...

Generate custom targets for the specified steps. This is required if the steps need to be triggered manually or if they need to be used as dependencies of other targets. If this option is not specified, the default value is taken from the **EP_STEP_TARGETS** directory property. See *ExternalProject_Add_StepTargets()* below for further discussion of the effects of this option.

INDEPENDENT_STEP_TARGETS <step-target>...

Deprecated since version 3.19: This is allowed only if policy *CMP0114* is not set to **NEW**.

Generates custom targets for the specified steps and prevent these targets from having the usual dependencies applied to them. If this option is not specified, the default value is taken from the **EP_INDEPENDENT_STEP_TARGETS** directory property. This option is mostly useful for allowing individual steps to be driven independently, such as for a CDash setup where each step

should be initiated and reported individually rather than as one whole build. See *ExternalProject_Add_StepTargets()* below for further discussion of the effects of this option.

Miscellaneous Options

LIST_SEPARATOR <sep>

For any of the various **..._COMMAND** options, and **CMAKE_ARGS**, **ExternalProject** will replace <sep> with ; in the specified command lines. This can be used to ensure a command has a literal ; in it where direct usage would otherwise be interpreted as argument separators to CMake APIs instead. Note that the separator should be chosen to avoid being confused for non-list-separator usages of the sequence. For example, using **LIST_SEPARATOR** allows for passing list values to CMake cache variables on the command line:

```
ExternalProject_Add(example
... # Download options, etc.
LIST_SEPARATOR " ,"
CMAKE_ARGS "-DCMAKE_PREFIX_PATH:STRING=${first_prefix},${second_prefix}"
)
```

COMMAND <cmd>...

Any of the other **..._COMMAND** options can have additional commands appended to them by following them with as many **COMMAND ...** options as needed (*generator expressions* are supported). For example:

```
ExternalProject_Add(example
... # Download options, etc.
BUILD_COMMAND ${CMAKE_COMMAND} -E echo "Starting ${CONFIG} build"
COMMAND        ${CMAKE_COMMAND} --build <BINARY_DIR> --config <CONFIG>
COMMAND        ${CMAKE_COMMAND} -E echo "${CONFIG} build complete"
)
```

It should also be noted that each build step is created via a call to *ExternalProject_Add_Step()*. See that command's documentation for the automatic substitutions that are supported for some options.

Obtaining Project Properties

ExternalProject_Get_Property

The **ExternalProject_Get_Property()** function retrieves external project target properties:

```
ExternalProject_Get_Property(<name> <prop1> [<prop2>...])
```

The function stores property values in variables of the same name. Property names correspond to the keyword argument names of **ExternalProject_Add()**. For example, the source directory might be retrieved like so:

```
ExternalProject_Get_property(myExtProj SOURCE_DIR)
message("Source dir of myExtProj = ${SOURCE_DIR}")
```

Explicit Step Management

The **ExternalProject_Add()** function on its own is often sufficient for incorporating an external project into the main build. Certain scenarios require additional work to implement desired behavior, such as adding in a custom step or making steps available as manually triggerable targets. The **ExternalProject_Add_Step()**, **ExternalProject_Add_StepTargets()** and **ExternalProject_Add_StepDependencies** functions provide the lower level control needed to implement such step-level capabilities.

ExternalProject_Add_Step

The **ExternalProject_Add_Step()** function specifies an additional custom step for an external project defined by an earlier call to *ExternalProject_Add()*:

`ExternalProject_Add_Step(<name> <step> [<option>...])`

<name> is the same as the name passed to the original call to *ExternalProject_Add()*. The specified **<step>** must not be one of the pre-defined steps (**mkdir**, **download**, **update**, **patch**, **configure**, **build**, **install** or **test**). The supported options are:

COMMAND <cmd>...

The command line to be executed by this custom step (*generator expressions* are supported). This option can be repeated multiple times to specify multiple commands to be executed in order.

COMMENT "<text>..."

Text to be printed when the custom step executes.

DEPENDS <step>...

Other steps (custom or pre-defined) on which this step depends.

DEPENDERS <step>...

Other steps (custom or pre-defined) that depend on this new custom step.

DEPENDS <file>...

Files on which this custom step depends.

INDEPENDENT <bool>

Added in version 3.19.

Specifies whether this step is independent of the external dependencies specified by the *ExternalProject_Add()*'s **DEPENDS** option. The default is **FALSE**. Steps marked as independent may depend only on other steps marked independent. See policy *CMP0114*.

Note that this use of the term "independent" refers only to independence from external targets specified by the **DEPENDS** option and is orthogonal to a step's dependencies on other steps.

If a step target is created for an independent step by the *ExternalProject_Add()* **STEP_TARGETS** option or by the *ExternalProject_Add_StepTargets()* function, it will not depend on the external targets, but may depend on targets for other steps.

BYPRODUCTS <file>...

Added in version 3.2.

Files that will be generated by this custom step but which might or might not have their modification time updated by subsequent builds. This may also be required to explicitly declare dependencies when using the *Ninja* generator. This list of files will ultimately be passed through as the **BYPRODUCTS** option to the *add_custom_command()* used to implement the custom step internally, which has additional documentation.

ALWAYS <bool>

When enabled, this option specifies that the custom step should always be run (i.e. that it is always considered out of date).

JOB_SERVER_AWARE <bool>

Added in version 3.28.

Specifies that the custom step is aware of the GNU Make job server. See the *add_custom_command()* documentation of its **JOB_SERVER_AWARE** option for details.

EXCLUDE_FROM_MAIN <bool>

When enabled, this option specifies that the external project's main target does not depend on the custom step. This may cause step targets to be created automatically for the steps on which this step depends. See policy *CMP0114*.

WORKING_DIRECTORY <dir>

Specifies the working directory to set before running the custom step's command. If this option is not specified, the directory will be the value of the *CMAKE_CURRENT_BINARY_DIR* at the point where **ExternalProject_Add_Step()** was called.

LOG <bool>

If set, this causes the output from the custom step to be captured to files in the external project's **LOG_DIR** if supplied or **STAMP_DIR**.

USES_TERMINAL <bool>

If enabled, this gives the custom step direct access to the terminal if possible.

The command line, comment, working directory and byproducts of every standard and custom step are processed to replace the tokens <SOURCE_DIR>, <SOURCE_SUBDIR>, <BINARY_DIR>, <INSTALL_DIR>, <TMP_DIR>, <DOWNLOAD_DIR> and <DOWNLOADED_FILE> with their corresponding property values defined in the original call to *ExternalProject_Add()*.

Added in version 3.3: Token replacement is extended to byproducts.

Added in version 3.11: The <DOWNLOAD_DIR> substitution token.

ExternalProject_Add_StepTargets

The **ExternalProject_Add_StepTargets()** function generates targets for the steps listed. The name of each created target will be of the form <name>--<step>:

```
ExternalProject_Add_StepTargets(<name> <step1> [<step2>...])
```

Creating a target for a step allows it to be used as a dependency of another target or to be triggered manually. Having targets for specific steps also allows them to be driven independently of each other by specifying targets on build command lines. For example, you may be submitting to a sub-project based dashboard where you want to drive the configure portion of the build, then submit to the dashboard, followed by the build portion, followed by tests. If you invoke a custom target that depends on a step halfway through the step dependency chain, then all the previous steps will also run to ensure everything is up to date.

Internally, *ExternalProject_Add()* calls *ExternalProject_Add_Step()* to create each step. If any **STEP_TARGETS** were specified, then **ExternalProject_Add_StepTargets()** will also be called after *ExternalProject_Add_Step()*. Even if a step is not mentioned in the **STEP_TARGETS** option, **ExternalProject_Add_StepTargets()** can still be called later to manually define a target for the step.

The **STEP_TARGETS** option for *ExternalProject_Add()* is generally the easiest way to ensure targets are created for specific steps of interest. For custom steps, **ExternalProject_Add_StepTargets()** must be called explicitly if a target should also be created for that custom step. An alternative to these two options is to populate the **EP_STEP_TARGETS** directory property. It acts as a default for the step target options and can save having to repeatedly specify the same set of step targets when multiple external projects are being defined.

Added in version 3.19: If *CMP0114* is set to **NEW**, step targets are fully responsible for holding the custom commands implementing their steps. The primary target created by **ExternalProject_Add** depends on the step targets, and the step targets depend on each other. The target-level dependencies match the file-level dependencies used by the custom commands for each step. The targets for steps created with *ExternalProject_Add_Step()*'s **INDEPENDENT** option do not depend on the external targets specified by *ExternalProject_Add()*'s **DEPENDS** option. The predefined steps **mkdir**, **download**, **update**, and **patch** are independent.

If *CMP0114* is not **NEW**, the following deprecated behavior is available:

- A deprecated **NO_DEPENDS** option may be specified immediately after the *<name>* and before the first step. If the **NO_DEPENDS** option is specified, the step target will not depend on the dependencies of the external project (i.e. on any dependencies of the *<name>* custom target created by *ExternalProject_Add()*). This is usually safe for the **download**, **update** and **patch** steps, since they do not typically require that the dependencies are updated and built. Using **NO_DEPENDS** for any of the other pre-defined steps, however, may break parallel builds. Only use **NO_DEPENDS** where it is certain that the named steps genuinely do not have dependencies. For custom steps, consider whether or not the custom commands require the dependencies to be configured, built and installed.
- The **INDEPENDENT_STEP_TARGETS** option for *ExternalProject_Add()*, or the **EP_INDEPENDENT_STEP_TARGETS** directory property, tells the function to call **ExternalProject_Add_StepTargets()** internally using the **NO_DEPENDS** option for the specified steps.

ExternalProject_Add_StepDependencies

Added in version 3.2.

The **ExternalProject_Add_StepDependencies()** function can be used to add dependencies to a step. The dependencies added must be targets CMake already knows about (these can be ordinary executable or library targets, custom targets or even step targets of another external project):

```
ExternalProject_Add_StepDependencies(<name> <step> <target1> [<target2>...]
```

This function takes care to set both target and file level dependencies and will ensure that parallel builds will not break. It should be used instead of *add_dependencies()* whenever adding a dependency for some of the step targets generated by the **ExternalProject** module.

Examples

The following example shows how to download and build a hypothetical project called *FooBar* from github:

```
include(ExternalProject)
ExternalProject_Add(fooBar
  GIT_REPOSITORY    git@github.com:FooCo/FooBar.git
  GIT_TAG           origin/release/1.2.3
)
```

For the sake of the example, also define a second hypothetical external project called *SecretSauce*, which is downloaded from a web server. Two URLs are given to take advantage of a faster internal network if available, with a fallback to a slower external server. The project is a typical **Makefile** project with no configure step, so some of the default commands are overridden. The build is only required to build the *sauce* target:

```
find_program(MAKE_EXE NAMES gmake nmake make)
ExternalProject_Add(secretsauce
  URL             http://intranet.somecompany.com/artifacts/sauce-2.7.tgz
```

```

        https://www.somecompany.com/downloads/sauce-2.7.zip
    URL_HASH      MD5=d41d8cd98f00b204e9800998ecf8427e
    CONFIGURE_COMMAND ""
    BUILD_COMMAND  ${MAKE_EXE} sauce
)

```

Suppose the build step of **secretsauce** requires that **foobar** must already be built. This could be enforced like so:

```
ExternalProject_Add_StepDependencies(secretsauce build foobar)
```

Another alternative would be to create a custom target for **foobar**'s build step and make **secretsauce** depend on that rather than the whole **foobar** project. This would mean **foobar** only needs to be built, it doesn't need to run its install or test steps before **secretsauce** can be built. The dependency can also be defined along with the **secretsauce** project:

```

ExternalProject_Add_StepTargets(foobar build)
ExternalProject_Add(secretsauce
    URL      http://intranet.somecompany.com/artifacts/sauce-2.7.tgz
             https://www.somecompany.com/downloads/sauce-2.7.zip
    URL_HASH MD5=d41d8cd98f00b204e9800998ecf8427e
    CONFIGURE_COMMAND ""
    BUILD_COMMAND  ${MAKE_EXE} sauce
    DEPENDS        foobar-build
)

```

Instead of calling *ExternalProject_Add_StepTargets()*, the target could be defined along with the **foobar** project itself:

```

ExternalProject_Add(foobar
    GIT_REPOSITORY git@github.com:FooCo/FooBar.git
    GIT_TAG        origin/release/1.2.3
    STEP_TARGETS   build
)

```

If many external projects should have the same set of step targets, setting a directory property may be more convenient. The **build** step target could be created automatically by setting the **EP_STEP_TARGETS** directory property before creating the external projects with *ExternalProject_Add()*:

```
set_property(DIRECTORY PROPERTY EP_STEP_TARGETS build)
```

Lastly, suppose that **secretsauce** provides a script called **makedoc** which can be used to generate its own documentation. Further suppose that the script expects the output directory to be provided as the only parameter and that it should be run from the **secretsauce** source directory. A custom step and a custom target to trigger the script can be defined like so:

```

ExternalProject_Add_Step(secretsauce docs
    COMMAND      <SOURCE_DIR>/makedoc <BINARY_DIR>
    WORKING_DIRECTORY <SOURCE_DIR>
    COMMENT      "Building secretsauce docs"
    ALWAYS       TRUE
    EXCLUDE_FROM_MAIN TRUE
)
ExternalProject_Add_StepTargets(secretsauce docs)

```

The custom step could then be triggered from the main build like so:

```
cmake --build . --target secretsauce-docs
```

FeatureSummary

Functions for generating a summary of enabled/disabled features.

These functions can be used to generate a summary of enabled and disabled packages and/or features for a build tree such as:

```
-- The following features have been enabled:

* Example, usage example

-- The following OPTIONAL packages have been found:

* LibXml2 (required version >= 2.4), XML library, <http://xmlsoft.org>
  Enables HTML-import in MyWordProcessor
  Enables odt-export in MyWordProcessor
* PNG, image library, <http://www.libpng.org/pub/png/>
  Enables saving screenshots

-- The following OPTIONAL packages have not been found:

* Lua, the Lua scripting language, <https://www.lua.org>
  Enables macros in MyWordProcessor
* OpenGL, Open Graphics Library
```

Global Properties

FeatureSummary_PKG_TYPES

Added in version 3.8.

This global property defines a *semicolon-separated list* of package types used by the **FeatureSummary** module.

The order in this list is important, the first package type in the list has the lowest importance, while the last has the highest importance. The type of a package can only be changed to a type with higher importance.

The default package types are **RUNTIME**, **OPTIONAL**, **RECOMMENDED** and **REQUIRED**, with their importance ranked as **RUNTIME** < **OPTIONAL** < **RECOMMENDED** < **REQUIRED**.

FeatureSummary_REQUIRED_PKG_TYPES

Added in version 3.8.

This global property defines a *semicolon-separated list* of package types that are considered required.

If one or more packages in these categories are not found, CMake will abort when the *feature_summary()* command is called with the **FATAL_ON_MISSING_REQUIRED_PACKAGES** option enabled.

The default value for this global property is **REQUIRED**.

FeatureSummary_DEFAULT_PKG_TYPE

Added in version 3.8.

This global property defines the default package type.

When the *feature_summary()* command is called, and the user has not explicitly set a type of some package, its type will be set to this value.

This value must be one of the types defined in the *FeatureSummary_PKG_TYPES* global property.

The default value for this global property is **OPTIONAL**.

FeatureSummary_<TYPE>_DESCRIPTION

Added in version 3.9.

This global property can be defined for each package <TYPE> to a string that will be used in the output titles of the *feature_summary()* command. For example:

The following <FeatureSummary_<TYPE>_DESCRIPTION> have been found:

If not set, default string <TYPE> **packages** is used.

Functions**feature_summary**

```
feature_summary([FILENAME <file>]
               [APPEND]
               [VAR <variable_name>]
               [INCLUDE_QUIET_PACKAGES]
               [FATAL_ON_MISSING_REQUIRED_PACKAGES]
               [DESCRIPTION <description> | DEFAULT_DESCRIPTION]
               [QUIET_ON_EMPTY]
               WHAT (ALL
                   | PACKAGES_FOUND | PACKAGES_NOT_FOUND
                   | <TYPE>_PACKAGES_FOUND | <TYPE>_PACKAGES_NOT_FOUND
                   | ENABLED_FEATURES | DISABLED_FEATURES)
               )
```

This function can be used to print information about enabled or disabled packages and features of a project. By default, only the names of the features/packages will be printed and their required version when one was specified. Use *set_package_properties()* to add more useful information, like e.g. a homepage URL for the respective package or their purpose in the project.

The options are:

WHAT This is the only mandatory option. It specifies what information will be printed:

ALL Print everything.

ENABLED_FEATURES

The list of all features which are enabled.

DISABLED_FEATURES

The list of all features which are disabled.

PACKAGES_FOUND

The list of all packages which have been found.

PACKAGES_NOT_FOUND

The list of all packages which have not been found.

For each package type **<TYPE>** defined by the *FeatureSummary_PKG_TYPES* global property, the following information can also be used:

<TYPE>_PACKAGES_FOUND

The list of only packages of type **<TYPE>** which have been found.

<TYPE>_PACKAGES_NOT_FOUND

The list of only packages of type **<TYPE>** which have not been found.

Changed in version 3.1: The **WHAT** option is now a multi-value keyword, so that these values can be combined, with the exception of the **ALL** value, in order to customize the output. For example:

```
feature_summary(WHAT ENABLED_FEATURES DISABLED_FEATURES)
```

FILENAME <file>

If this option is given, the information is printed into this file instead of the terminal. Relative **<file>** path is interpreted as being relative to the current source directory (i.e. *CMAKE_CURRENT_SOURCE_DIR*).

APPEND

If this option is given, the output is appended to the **<file>** provided by the **FILENAME** option, otherwise the file is overwritten if it already exists.

VAR <variable_name>

If this option is given, the information is stored into the specified variable **<variable_name>** instead of the terminal.

DESCRIPTION <description>

A description or headline which will be printed above the actual content. Without this option, if only one package type was requested, no title is printed, unless a custom string is explicitly set using this option or **DEFAULT_DESCRIPTION** option is used that outputs a default title for the requested type.

DEFAULT_DESCRIPTION

Added in version 3.9.

The default description or headline to be printed above the content as opposed to the customizable **DESCRIPTION <description>**.

INCLUDE_QUIET_PACKAGES

If this option is given, packages which have been searched with *find_package(... QUIET)* will also be listed. By default they are skipped.

FATAL_ON_MISSING_REQUIRED_PACKAGES

If this option is given, CMake will abort with fatal error if a package which is marked as one of the package types listed in the *FeatureSummary_REQUIRED_PKG_TYPES* global property has not been found.

The *FeatureSummary_DEFAULT_PKG_TYPE* global property can be modified to change the default package type assigned when not explicitly assigned by the user.

QUIET_ON_EMPTY

Added in version 3.8.

If this option is given, when only one package type was requested, and no packages belonging to that category were found, then no output (including the **DESCRIPTION**) is printed nor added to the **FILENAME**, or the **VAR** variable.

Example 1, append everything to a file:

```
include(FeatureSummary)
feature_summary(WHAT ALL
                FILENAME ${CMAKE_BINARY_DIR}/all.log APPEND)
```

Example 2, print the enabled features into the variable **enabledFeaturesText**, including the **QUIET** packages:

```
include(FeatureSummary)
feature_summary(WHAT ENABLED_FEATURES
                INCLUDE QUIET_PACKAGES
                DESCRIPTION "Enabled Features:"
                VAR enabledFeaturesText)
message(STATUS "${enabledFeaturesText}")
```

Example 3, add custom package type and print only the categories that are not empty:

```
include(FeatureSummary)
set_property(GLOBAL APPEND PROPERTY FeatureSummary_PKG_TYPES BUILD)
find_package(F00)
set_package_properties(F00 PROPERTIES TYPE BUILD)
feature_summary(WHAT BUILD_PACKAGES_FOUND
                DESCRIPTION "Build tools found:"
                QUIET_ON_EMPTY)
feature_summary(WHAT BUILD_PACKAGES_NOT_FOUND
                DESCRIPTION "Build tools not found:"
                QUIET_ON_EMPTY)
```

set_package_properties

```
set_package_properties(<name> PROPERTIES
                      [URL <url>]
                      [DESCRIPTION <description>]
                      [TYPE (RUNTIME|OPTIONAL|RECOMMENDED|REQUIRED)]
                      [PURPOSE <purpose>]
                      )
```

Use this function to configure and provide information about the package named **<name>**, which can then be displayed using the *feature_summary()* command. This can be performed either directly within the corresponding *Find module* or in the project that uses the module after invoking the *find_package()* call. The features for which information can be set are determined automatically after the *find_package()* command.

URL <url>

This should be the homepage of the package, or something similar. Ideally this is set already directly in the *Find module*.

DESCRIPTION <description>

A short description what that package is, at most one sentence. Ideally this is set already directly in the *Find module*.

TYPE <type>

What type of dependency has the using project on that package. Default is **OPTIONAL**. In this case it is a package which can be used by the project when available at buildtime, but it also work without. **RECOMMENDED** is similar to **OPTIONAL**, i.e. the project will build if the package is not present, but the functionality of the resulting binaries will be severely limited. If a **REQUIRED** package is not available at buildtime, the project may not even build. This can be combined with the *feature_summary(FATAL_ON_MISSING_REQUIRED_PACKAGES)* command option. Last, a **RUNTIME** package is a package which is actually not used at all during the build, but which is required for actually running the resulting binaries. So if such a package is missing, the project can still be built, but it may not work later on. If **set_package_properties()** is called multiple times for the same package with different **TYPE**s, the **TYPE** is only changed to higher **TYPE**s (**RUNTIME** < **OPTIONAL** < **RECOMMENDED** < **REQUIRED**), lower **TYPE**s are ignored. The **TYPE** property is project-specific, so it cannot be set by the *Find module*, but must be set in the project. The accepted types can be changed by setting the *FeatureSummary_PKG_TYPES* global property.

PURPOSE <purpose>

This describes which features this package enables in the project, i.e. it tells the user what functionality they get in the resulting binaries. If **set_package_properties()** is called multiple times for a package, all **PURPOSE** properties are appended to a list of purposes of the package in the project. As the **TYPE** property, also the **PURPOSE** property is project-specific, so it cannot be set by the *Find module*, but must be set in the project.

Example for setting the info for a package:

```
include(FeatureSummary)
find_package(LibXml2)
set_package_properties(LibXml2 PROPERTIES
    DESCRIPTION "XML library"
    URL "http://xmlsoft.org")

# or
set_package_properties(LibXml2 PROPERTIES
    TYPE RECOMMENDED
    PURPOSE "Enables HTML-import in MyWordProcessor")

# or
set_package_properties(LibXml2 PROPERTIES
    TYPE OPTIONAL
    PURPOSE "Enables odt-export in MyWordProcessor")

find_package(DBUS)
set_package_properties(DBUS PROPERTIES
    TYPE RUNTIME
    PURPOSE "Necessary to disable the screensaver during a presentation")

add_feature_info

add_feature_info(<name> <enabled> <description>)
```

Use this function to add information about a feature identified with a given <name>. The

<**enabled**> contains whether this feature is enabled or not. It can be a variable or a list of conditions. <**description**> is a text describing the feature. The information can be displayed using *feature_summary()* for **ENABLED_FEATURES** and **DISABLED_FEATURES** respectively.

Changed in version 3.8: <**enabled**> can be a list of conditions.

Changed in version 4.0: Full *Condition Syntax* is now supported for <**enabled**>. See policy *CMP0183*.

Example for setting the info for a feature:

```
include(FeatureSummary)

option(WITH_FOO "Help for foo" ON)
add_feature_info(Foo WITH_FOO "this feature provides very cool stuff")
```

Example for setting feature info based on a list of conditions:

```
option(WITH_FOO "Help for foo" ON)
option(WITH_BAR "Help for bar" OFF)
add_feature_info(
  FooBar
  "WITH_FOO;NOT WITH_BAR"
  "this feature is enabled when WITH_FOO is ON and WITH_BAR turned OFF"
)
```

Example for setting feature info depending on a full condition syntax:

Unlike semicolon-separated list of conditions, this enables using entire condition syntax as being the **if** clause argument, such as grouping conditions with parens and similar.

```
option(WITH_FOO "Help for foo" ON)
option(WITH_BAR "Help for bar" ON)
option(WITH_BAZ "Help for baz" OFF)
add_feature_info(
  FooBarBaz
  "WITH_FOO AND (WITH_BAR OR WITH_BAZ)"
  "this feature is enabled when the entire condition is true"
)
```

Deprecated Functions

The following legacy and deprecated functions are provided for backward compatibility with previous CMake versions:

set_package_info

Deprecated since version 3.8.

```
set_package_info(<name> <description> [ <url> [<purpose>] ] )
```

Set up information about the package <**name**>, which can then be displayed via *feature_summary()*. This can be done either directly in the *Find module* or in the project which uses the **FeatureSummary** module after the *find_package()* call. The features for which information can be set are added automatically by the **find_package()** command.

This function is deprecated. Use the `set_package_properties()`, and `add_feature_info()` functions instead.

set_feature_info

Deprecated since version 3.8.

```
set_feature_info(<name> <description> [<url>])
```

Does the same as:

```
set_package_info(<name> <description> [<url>])
```

print_enabled_features

Deprecated since version 3.8.

```
print_enabled_features()
```

Does the same as:

```
feature_summary(WHAT ENABLED_FEATURES DESCRIPTION "Enabled features:")
```

print_disabled_features

Deprecated since version 3.8.

```
print_disabled_features()
```

Does the same as:

```
feature_summary(WHAT DISABLED_FEATURES DESCRIPTION "Disabled features:")
```

FetchContent

Added in version 3.11.

NOTE:

The *Using Dependencies Guide* provides a high-level introduction to this general topic. It provides a broader overview of where the **FetchContent** module fits into the bigger picture, including its relationship to the `find_package()` command. The guide is recommended pre-reading before moving on to the details below.

Overview

This module enables populating content at configure time via any method supported by the *ExternalProject* module. Whereas *ExternalProject_Add()* downloads at build time, the **FetchContent** module makes content available immediately, allowing the configure step to use the content in commands like `add_subdirectory()`, `include()` or `file()` operations.

Content population details should be defined separately from the command that performs the actual population. This separation ensures that all the dependency details are defined before anything might try to use them to populate content. This is particularly important in more complex project hierarchies where dependencies may be shared between multiple projects.

The following shows a typical example of declaring content details for some dependencies and then ensuring they are populated with a separate call:

```

FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG         703bd9caab50b139428cea1aaff9974ebee5742e # release-1.10.0
)
FetchContent_Declare(
  myCompanyIcons
  URL      https://intranet.mycompany.com/assets/iconset_1.12.tar.gz
  URL_HASH MD5=5588a7b18261c20068beabfb4f530b87
)

FetchContent_MakeAvailable(googletest myCompanyIcons)

```

The `FetchContent_MakeAvailable()` command ensures the named dependencies have been populated, either by an earlier call, or by populating them itself. When performing the population, it will also add them to the main build, if possible, so that the main build can use the populated projects' targets, etc. See the command's documentation for how these steps are performed.

When using a hierarchical project arrangement, projects at higher levels in the hierarchy are able to override the declared details of content specified anywhere lower in the project hierarchy. The first details to be declared for a given dependency take precedence, regardless of where in the project hierarchy that occurs. Similarly, the first call that tries to populate a dependency "wins", with subsequent populations reusing the result of the first instead of repeating the population again. See the *Examples* which demonstrate this scenario.

The **FetchContent** module also supports defining and populating content in a single call, with no check for whether the content has been populated elsewhere already. This should not be done in projects, but may be appropriate for populating content in *CMake script mode*. See `FetchContent_Populate()` for details.

Commands

FetchContent_Declare

```

FetchContent_Declare(
  <name>
  <contentOptions>...
  [EXCLUDE_FROM_ALL]
  [SYSTEM]
  [OVERRIDE_FIND_PACKAGE |
   FIND_PACKAGE_ARGS args...]
)

```

The **FetchContent_Declare()** function records the options that describe how to populate the specified content. If such details have already been recorded earlier in this project (regardless of where in the project hierarchy), this and all later calls for the same content **<name>** are ignored. This "first to record, wins" approach is what allows hierarchical projects to have parent projects override content details of child projects.

The content **<name>** can be any string without spaces, but good practice would be to use only letters, numbers, and underscores. The name will be treated case-insensitively, and it should be obvious for the content it represents. It is often the name of the child project, or the value given to its top level `project()` command (if it is a CMake project). For well-known public projects, the name should generally be the official name of the project. Choosing an unusual name makes it unlikely that other projects needing that same content will use the same name, leading to the content being populated multiple times.

The **<contentOptions>** can be any of the download, update, or patch options that the *ExternalProject_Add()* command understands. The configure, build, install, and test steps are explicitly disabled, so options related to those steps will be ignored. The **SOURCE_SUBDIR** option is an exception, see *FetchContent_MakeAvailable()* for details on how that affects behavior.

Changed in version 3.30: When policy *CMP0168* is set to **NEW**, some output-related and directory-related options are ignored. See the policy documentation for details.

In most cases, **<contentOptions>** will just be a couple of options defining the download method and method-specific details like a commit tag or archive hash. For example:

```
FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG        703bd9caab50b139428cea1aaff9974ebee5742e # release-1.10.0
)

FetchContent_Declare(
  myCompanyIcons
  URL      https://intranet.mycompany.com/assets/iconset_1.12.tar.gz
  URL_HASH MD5=5588a7b18261c20068beabfb4f530b87
)

FetchContent_Declare(
  myCompanyCertificates
  SVN_REPOSITORY svn+ssh://svn.mycompany.com/srv/svn/trunk/certs
  SVN_REVISION   -r12345
)
```

Where contents are being fetched from a remote location and you do not control that server, it is advisable to use a hash for **GIT_TAG** rather than a branch or tag name. A commit hash is more secure and helps to confirm that the downloaded contents are what you expected.

Changed in version 3.14: Commands for the download, update, or patch steps can access the terminal. This may be needed for things like password prompts or real-time display of command progress.

Added in version 3.22: The *CMAKE_TLS_VERIFY*, *CMAKE_TLS_CAINFO*, *CMAKE_NETRC*, and *CMAKE_NETRC_FILE* variables now provide the defaults for their corresponding content options, just like they do for *ExternalProject_Add()*. Previously, these variables were ignored by the **FetchContent** module.

Added in version 3.24:

FIND_PACKAGE_ARGS

This option is for scenarios where the *FetchContent_MakeAvailable()* command may first try a call to *find_package()* to satisfy the dependency for **<name>**. By default, such a call would be simply **find_package(<name>)**, but **FIND_PACKAGE_ARGS** can be used to provide additional arguments to be appended after the **<name>**. **FIND_PACKAGE_ARGS** can also be given with nothing after it, which indicates that *find_package()* can still be called if *FETCHCONTENT_TRY_FIND_PACKAGE_MODE* is set to **OPT_IN**, or is not set.

It would not normally be appropriate to specify **REQUIRED** as one of the additional arguments after **FIND_PACKAGE_ARGS**. Doing so would mean the *find_package()* call must succeed, so none of the other details specified in the **FetchContent_Declare()** call would get a chance to be used as a fall-back.

Everything after the **FIND_PACKAGE_ARGS** keyword is appended to the *find_package()* call, so all other <contentOptions> must come before the **FIND_PACKAGE_ARGS** keyword. If the *CMAKE_FIND_PACKAGE_TARGETS_GLOBAL* variable is set to true at the time **FetchContent_Declare()** is called, a **GLOBAL** keyword will be appended to the *find_package()* arguments if it was not already specified. It will also be appended if **FIND_PACKAGE_ARGS** was not given, but *FETCHCONTENT_TRY_FIND_PACKAGE_MODE* was set to **ALWAYS**.

OVERRIDE_FIND_PACKAGE cannot be used when **FIND_PACKAGE_ARGS** is given.

Dependency Providers discusses another way that *FetchContent_MakeAvailable()* calls can be redirected. **FIND_PACKAGE_ARGS** is intended for project control, whereas dependency providers allow users to override project behavior.

OVERRIDE_FIND_PACKAGE

When a **FetchContent_Declare(<name> ...)** call includes this option, subsequent calls to **find_package(<name> ...)** will ensure that **FetchContent_MakeAvailable(<name>)** has been called, then use the config package files in the *CMAKE_FIND_PACKAGE_REDIRECTS_DIR* directory (which are usually created by **FetchContent_MakeAvailable()**). This effectively makes *FetchContent_MakeAvailable()* override *find_package()* for the named dependency, allowing the former to satisfy the package requirements of the latter. **FIND_PACKAGE_ARGS** cannot be used when **OVERRIDE_FIND_PACKAGE** is given.

If a *dependency provider* has been set and the project calls *find_package()* for the <name> dependency, **OVERRIDE_FIND_PACKAGE** will not prevent the provider from seeing that call. Dependency providers always have the opportunity to intercept any direct call to *find_package()*, except if that call contains the **BYPASS_PROVIDER** option.

Added in version 3.25:

SYSTEM

If the **SYSTEM** argument is provided, the *SYSTEM* directory property of a subdirectory added by *FetchContent_MakeAvailable()* will be set to true. This will affect non-imported targets created as part of that command. See the *SYSTEM* target property documentation for a more detailed discussion of the effects.

Added in version 3.28:

EXCLUDE_FROM_ALL

If the **EXCLUDE_FROM_ALL** argument is provided, then targets in the subdirectory added by *FetchContent_MakeAvailable()* will not be included in the **ALL** target by default, and may be excluded from IDE project files. See the documentation for the directory property *EXCLUDE_FROM_ALL* for a detailed discussion of the effects.

FetchContent_MakeAvailable

Added in version 3.14.

```
FetchContent_MakeAvailable(<name1> [<name2>...])
```

This command ensures that each of the named dependencies are made available to the project by the time it returns. There must have been a call to *FetchContent_Declare()* for each dependency, and the first such call will control how that dependency will be made available, as described below.

If **<lowercaseName>_SOURCE_DIR** is not set:

- Added in version 3.24: If a *dependency provider* is set, call the provider's command with **FETCHCONTENT_MAKEAVAILABLE_SERIAL** as the first argument, followed by the arguments of the first call to *FetchContent_Declare()* for **<name>**. 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* was set to **NEVER** when the details were declared, any **FIND_PACKAGE_ARGS** will be omitted. The **OVER-RIDE_FIND_PACKAGE** keyword is also always omitted. If the provider fulfilled the request, **FetchContent_MakeAvailable()** will consider that dependency handled, skip the remaining steps below, and move on to the next dependency in the list.
- Added in version 3.24: If permitted, *find_package(<name> [<args>...])* will be called, where **<args>...** may be provided by the **FIND_PACKAGE_ARGS** option in *FetchContent_Declare()*. The value of the *FETCHCONTENT_TRY_FIND_PACKAGE_MODE* variable at the time *FetchContent_Declare()* was called determines whether **FetchContent_MakeAvailable()** can call *find_package()*. If the *CMAKE_FIND_PACKAGE_TARGETS_GLOBAL* variable is set to true when **FetchContent_MakeAvailable()** is called, it still affects any imported targets created when that in turn calls *find_package()*, even if that variable was false when the corresponding details were declared.

If the dependency was not satisfied by a provider or a *find_package()* call, **FetchContent_MakeAvailable()** then uses the following logic to make the dependency available:

- If the dependency has already been populated earlier in this run, set the **<lowercaseName>_POPULATED**, **<lowercaseName>_SOURCE_DIR**, and **<lowercaseName>_BINARY_DIR** variables in the same way as a call to *FetchContent_GetProperties()*, then skip the remaining steps below and move on to the next dependency in the list.
- Populate the dependency using the details recorded by an earlier call to *FetchContent_Declare()*. Halt with a fatal error if no such details have been recorded. *FETCHCONTENT_SOURCE_DIR_<uppercaseName>* can be used to override the declared details and use content provided at the specified location instead.
- Added in version 3.24: Ensure the *CMAKE_FIND_PACKAGE_REDIRECTS_DIR* directory contains a **<lowercaseName>-config.cmake** and a **<lowercaseName>-config-version.cmake** file (or equivalently, **<name>Config.cmake** and **<name>ConfigVersion.cmake**). The directory that the *CMAKE_FIND_PACKAGE_REDIRECTS_DIR* variable points to is cleared at the start of every CMake run. If no config file exists after populating the dependency in the previous step, a minimal one will be written which includes any **<lowercaseName>-extra.cmake** or **<name>Extra.cmake** file with the **OPTIONAL** flag (so the files can be missing and won't generate a warning). Similarly, if no config version file exists, a very simple one will be written which sets **PACKAGE_VERSION_COMPATIBLE** and **PACKAGE_VERSION_EXACT** to true. This ensures all future calls to *find_package()* for the dependency will use the redirected

config file, regardless of any version requirements. CMake cannot automatically determine an arbitrary dependency's version, so it cannot set **PACKAGE_VERSION**. When a dependency is pulled in via `add_subdirectory()` in the next step, it may choose to overwrite the generated config version file in `CMAKE_FIND_PACKAGE_REDIRECTS_DIR` with one that also sets **PACKAGE_VERSION**. The dependency may also write a `<lowercaseName>-extra.cmake` or `<name>Extra.cmake` file to perform custom processing, or define any variables that their normal (installed) package config file would otherwise usually define (many projects don't do any custom processing or set any variables and therefore have no need to do this). If required, the main project can write these files instead if the dependency project doesn't do so. This allows the main project to add missing details from older dependencies that haven't or can't be updated to support this functionality. See *Integrating With `find_package()`* for examples.

- If the top directory of the populated content contains a **CMakeLists.txt** file, call `add_subdirectory()` to add it to the main build. It is not an error for there to be no **CMakeLists.txt** file, which allows the command to be used for dependencies that make downloaded content available at a known location, but which do not need or support being added directly to the build.

Added in version 3.18: The **SOURCE_SUBDIR** option can be given in the declared details to look somewhere below the top directory instead (i.e. the same way that **SOURCE_SUBDIR** is used by the `ExternalProject_Add()` command). The path provided with **SOURCE_SUBDIR** must be relative, and it will be treated as relative to the top directory. It can also point to a directory that does not contain a **CMakeLists.txt** file, or even to a directory that doesn't exist. This can be used to avoid adding a project that contains a **CMakeLists.txt** file in its top directory.

Added in version 3.25: If the **SYSTEM** keyword was included in the call to `FetchContent_Declare()`, the **SYSTEM** keyword will be added to the `add_subdirectory()` command.

Added in version 3.28: If the **EXCLUDE_FROM_ALL** keyword was included in the call to `FetchContent_Declare()`, the **EXCLUDE_FROM_ALL** keyword will be added to the `add_subdirectory()` command.

Added in version 3.29: `CMAKE_EXPORT_FIND_PACKAGE_NAME` is set to the dependency name before calling `add_subdirectory()`.

Projects should aim to declare the details of all dependencies they might use before they call **FetchContent_MakeAvailable()** for any of them. This ensures that if any of the dependencies are also sub-dependencies of one or more of the others, the main project still controls the details that will be used (because it will declare them first before the dependencies get a chance to). In the following code samples, assume that the **uses_other** dependency also uses **FetchContent** to add the **other** dependency internally:

```
# WRONG: Should declare all details first
FetchContent_Declare(uses_other ...)
FetchContent_MakeAvailable(uses_other)
```

```
FetchContent_Declare(other ...)    # Will be ignored, uses_other beat us to
FetchContent_MakeAvailable(other)  # Would use details declared by uses_oth
```

```
# CORRECT: All details declared first, so they will take priority
FetchContent_Declare(uses_other ...)
FetchContent_Declare(other ...)
FetchContent_MakeAvailable(uses_other other)
```

Note that `CMAKE_VERIFY_INTERFACE_HEADER_SETS` is explicitly set to false upon entry to **FetchContent_MakeAvailable()**, and is restored to its original value before the command returns. Developers typically only want to verify header sets from the main project, not those from any dependencies. This local manipulation of the `CMAKE_VERIFY_INTERFACE_HEADER_SETS` variable provides that intuitive behavior. You can use variables like `CMAKE_PROJECT_INCLUDE` or `CMAKE_PROJECT_<PROJECT-NAME>_INCLUDE` to turn verification back on for all or some dependencies. You can also set the `VERIFY_INTERFACE_HEADER_SETS` property of individual targets.

FetchContent_Populate

The **FetchContent_Populate()** command is a self-contained call which can be used to perform content population as an isolated operation. It is rarely the right command to use, projects should almost always use `FetchContent_Declare()` and `FetchContent_MakeAvailable()` instead. The main use case for **FetchContent_Populate()** is in *CMake script mode* as part of implementing some other higher level custom feature.

```
FetchContent_Populate(
  <name>
  [QUIET]
  [SUBBUILD_DIR <subBuildDir>]
  [SOURCE_DIR <srcDir>]
  [BINARY_DIR <binDir>]
  ...
)
```

At least one option must be specified after `<name>`, otherwise the call is interpreted differently (see *below*). The supported options for **FetchContent_Populate()** are the same as those for `FetchContent_Declare()`, with a few exceptions. The following do not relate to populating content with **FetchContent_Populate()** and therefore are not supported:

- **EXCLUDE_FROM_ALL**
- **SYSTEM**
- **OVERRIDE_FIND_PACKAGE**
- **FIND_PACKAGE_ARGS**

The few options shown in the signature above are either specific to **FetchContent_Populate()**, or their behavior is slightly modified from how `ExternalProject_Add()` treats them:

QUIET

The **QUIET** option can be given to hide the output associated with populating the specified content. If the population fails, the output will be shown regardless of whether this option was given or not so that the cause of the failure can be diagnosed. The `FETCHCONTENT_QUIET` variable has no effect on **FetchContent_Populate()** calls of this form where the content details are provided directly.

Changed in version 3.30: The **QUIET** option and `FETCHCONTENT_QUIET` variable have no effect when policy `CMP0168` is set to **NEW**. The output is still quiet by default in that case, but verbosity is controlled by the message logging level (see `CMAKE_MESSAGE_LOG_LEVEL` and `--log-level`).

SUBBUILD_DIR

The **SUBBUILD_DIR** argument can be provided to change the location of the sub-build created to perform the population. The default value is **\${CMAKE_CURRENT_BINARY_DIR}/<lowercaseName>-subbuild**, and it would be unusual to need to override this default. If a relative path is specified, it will be interpreted as relative to *CMAKE_CURRENT_BINARY_DIR*. This option should not be confused with the **SOURCE_SUBDIR** option, which only affects the *FetchContent_MakeAvailable()* command.

Changed in version 3.30: **SUBBUILD_DIR** is ignored when policy *CMP0168* is set to **NEW**, since there is no sub-build in that case.

SOURCE_DIR, BINARY_DIR

The **SOURCE_DIR** and **BINARY_DIR** arguments are supported by *ExternalProject_Add()*, but different default values are used by **FetchContent_Populate()**. **SOURCE_DIR** defaults to **\${CMAKE_CURRENT_BINARY_DIR}/<lowercaseName>-src**, and **BINARY_DIR** defaults to **\${CMAKE_CURRENT_BINARY_DIR}/<lowercaseName>-build**. If a relative path is specified, it will be interpreted as relative to *CMAKE_CURRENT_BINARY_DIR*.

In addition to the above explicit options, any other unrecognized options are passed through unmodified to *ExternalProject_Add()* to set up the download, patch, and update steps. The following options are explicitly prohibited (they are disabled by the **FetchContent_Populate()** command):

- **CONFIGURE_COMMAND**
- **BUILD_COMMAND**
- **INSTALL_COMMAND**
- **TEST_COMMAND**

With this form, the *FETCHCONTENT_FULLY_DISCONNECTED* and *FETCHCONTENT_UPDATES_DISCONNECTED* variables and policy *CMP0170* are ignored.

When this form of **FetchContent_Populate()** returns, the following variables will be set in the scope of the caller:

<lowercaseName>_SOURCE_DIR

The location where the populated content can be found upon return.

<lowercaseName>_BINARY_DIR

A directory originally intended for use as a corresponding build directory, but is unlikely to be relevant when using this form of the command.

If using **FetchContent_Populate()** within *CMake script mode*, be aware that the implementation sets up a sub-build which therefore requires a CMake generator and build tool to be available. If these cannot be found by default, then the *CMAKE_GENERATOR* and potentially the *CMAKE_MAKE_PROGRAM* variables will need to be set appropriately on the command line invoking the script.

Changed in version 3.30: If policy *CMP0168* is set to **NEW**, no sub-build is used. Within *CMake script mode*, that allows **FetchContent_Populate()** to be called without any build tool or CMake generator.

Added in version 3.18: Added support for the **DOWNLOAD_NO_EXTRACT** option.

The command supports another form, although it should no longer be used:

```
FetchContent_Populate(<name>)
```

Changed in version 3.30: This form is deprecated. Policy *CMP0169* provides backward compatibility for projects that still need to use this form, but projects should be updated to use *FetchContent_MakeAvailable()* instead.

In this form, the only argument given to **FetchContent_Populate()** is the **<name>**. When used this way, the command assumes the content details have been recorded by an earlier call to *FetchContent_Declare()*. The details are stored in a global property, so they are unaffected by things like variable or directory scope. Therefore, it doesn't matter where in the project the details were previously declared, as long as they have been declared before the call to **FetchContent_Populate()**. Those saved details are then used to populate the content using a method based on *ExternalProject_Add()* (see policy *CMP0168* for important behavioral aspects of how that is done).

When this form of **FetchContent_Populate()** returns, the following variables will be set in the scope of the caller:

<lowercaseName>_POPULATED

This will always be set to **TRUE** by the call.

<lowercaseName>_SOURCE_DIR

The location where the populated content can be found upon return.

<lowercaseName>_BINARY_DIR

A directory intended for use as a corresponding build directory.

The values of the three variables can also be retrieved from anywhere in the project hierarchy using the *FetchContent_GetProperties()* command.

The implementation ensures that if the content has already been populated in a previous CMake run, that content will be reused rather than repopulating again. For the common case where population involves downloading content, the cost of the download is only paid once. But note that it is an error to call **FetchContent_Populate(<name>)** with the same **<name>** more than once within a single CMake run. See *FetchContent_GetProperties()* for how to determine if population of a **<name>** has already been performed in the current run.

FetchContent_GetProperties

When using saved content details, a call to *FetchContent_MakeAvailable()* or *FetchContent_Populate()* records information in global properties which can be queried at any time. This information may include the source and binary directories associated with the content, and also whether or not the content population has been processed during the current configure run.

```
FetchContent_GetProperties(
    <name>
    [SOURCE_DIR <srcDirVar>]
    [BINARY_DIR <binDirVar>]
    [POPULATED <doneVar>]
)
```

The **SOURCE_DIR**, **BINARY_DIR**, and **POPULATED** options can be used to specify which properties should be retrieved. Each option accepts a value which is the name of the variable in which to store that property. Most of the time though, only **<name>** is given, in which case the call will then set the same variables as a call to *FetchContent_MakeAvailable(name)* or

FetchContent_Populate(name). Note that the **SOURCE_DIR** and **BINARY_DIR** values can be empty if the call is fulfilled by a *dependency provider*.

This command is rarely needed when using *FetchContent_MakeAvailable()*. It is more commonly used as part of implementing the deprecated pattern with *FetchContent_Populate()*, which ensures that the relevant variables will always be defined regardless of whether or not the population has been performed elsewhere in the project already:

```
# WARNING: This pattern is deprecated, don't use it!
#
# Check if population has already been performed
FetchContent_GetProperties(depname)
if(NOT depname_POPULATED)
  # Fetch the content using previously declared details
  FetchContent_Populate(depname)

  # Set custom variables, policies, etc.
  # ...

  # Bring the populated content into the build
  add_subdirectory(${depname_SOURCE_DIR} ${depname_BINARY_DIR})
endif()
```

FetchContent_SetPopulated

Added in version 3.24.

NOTE:

This command should only be called by *dependency providers*. Calling it in any other context is unsupported and future CMake versions may halt with a fatal error in such cases.

```
FetchContent_SetPopulated(
  <name>
  [SOURCE_DIR <srcDir>]
  [BINARY_DIR <binDir>]
)
```

If a provider command fulfills a **FETCHCONTENT_MAKEAVAILABLE_SERIAL** request, it must call this function before returning. The **SOURCE_DIR** and **BINARY_DIR** arguments can be used to specify the values that *FetchContent_GetProperties()* should return for its corresponding arguments. Only provide **SOURCE_DIR** and **BINARY_DIR** if they have the same meaning as if they had been populated by the built-in *FetchContent_MakeAvailable()* implementation.

Variables

A number of cache variables can influence the behavior where details from a *FetchContent_Declare()* call are used to populate content.

NOTE:

All of these variables are intended for the developer to customize behavior. They should not normally be set by the project.

FETCHCONTENT_BASE_DIR

In most cases, the saved details do not specify any options relating to the directories to use for the internal sub-build, final source, and build areas. It is generally best to leave these decisions up to the **FetchContent** module to handle on the project's behalf. The **FETCHCONTENT_BASE_DIR** cache variable controls the point under which all content population

directories are collected, but in most cases, developers would not need to change this. The default location is `${CMAKE_BINARY_DIR}/_deps`, but if developers change this value, they should aim to keep the path short and just below the top level of the build tree to avoid running into path length problems on Windows.

FETCHCONTENT_QUIET

The logging output during population can be quite verbose, making the configure stage quite noisy. This cache option (**ON** by default) hides all population output unless an error is encountered. If experiencing problems with hung downloads, temporarily switching this option off may help diagnose which content population is causing the issue.

Changed in version 3.30: **FETCHCONTENT_QUIET** is ignored if policy *CMP0168* is set to **NEW**. The output is still quiet by default in that case, but verbosity is controlled by the message logging level (see *CMAKE_MESSAGE_LOG_LEVEL* and `--log-level`).

FETCHCONTENT_FULLY_DISCONNECTED

When this option is enabled, no attempt is made to download or update any content. It is assumed that all content has already been populated in a previous run, or the source directories have been pointed at existing contents the developer has provided manually (using options described further below). When the developer knows that no changes have been made to any content details, turning this option **ON** can speed up the configure stage. It is **OFF** by default.

NOTE:

The **FETCHCONTENT_FULLY_DISCONNECTED** variable is not an appropriate way to prevent any network access on the first run in a build directory. Doing so can break projects, lead to misleading error messages, and hide subtle population failures. This variable is specifically intended to only be turned on *after* the first time CMake has been run. If you want to prevent network access even on the first run, use a *dependency provider* and populate the dependency from local content instead.

Changed in version 3.30: The constraint that the source directory has already been populated when **FETCHCONTENT_FULLY_DISCONNECTED** is true is now enforced. See policy *CMP0170*.

FETCHCONTENT_UPDATES_DISCONNECTED

This is a less severe download/update control compared to *FETCHCONTENT_FULLY_DISCONNECTED*. Instead of bypassing all download and update logic, **FETCHCONTENT_UPDATES_DISCONNECTED** only prevents the update step from making connections to remote servers when using the git or hg download methods. Updates still occur if details about the update step change, but the update is attempted with only the information already available locally (so switching to a different tag or commit that is already fetched locally will succeed, but switching to an unknown commit hash will fail). The download step is not affected, so if content has not been downloaded previously, it will still be downloaded when this option is enabled. This can speed up the configure step, but not as much as *FETCHCONTENT_FULLY_DISCONNECTED*. **FETCHCONTENT_UPDATES_DISCONNECTED** is **OFF** by default.

FETCHCONTENT_TRY_FIND_PACKAGE_MODE

Added in version 3.24.

This variable modifies the details that *FetchContent_Declare()* records for a given dependency. While it ultimately controls the behavior of *FetchContent_MakeAvailable()*, it is the variable's value when *FetchContent_Declare()* is called that gets used. It makes no difference what the variable is set to when *FetchContent_MakeAvailable()* is called. Since the variable should only be set by the user and not by projects directly, it will typically have the same value throughout anyway,

so this distinction is not usually noticeable.

FETCHCONTENT_TRY_FIND_PACKAGE_MODE ultimately controls whether *FetchContent_MakeAvailable()* is allowed to call *find_package()* to satisfy a dependency. The variable can be set to one of the following values:

OPT_IN

FetchContent_MakeAvailable() will only call *find_package()* if the *FetchContent_Declare()* call included a **FIND_PACKAGE_ARGS** keyword. This is also the default behavior if **FETCHCONTENT_TRY_FIND_PACKAGE_MODE** is not set.

ALWAYS

find_package() can be called by *FetchContent_MakeAvailable()* regardless of whether the *FetchContent_Declare()* call included a **FIND_PACKAGE_ARGS** keyword or not. If no **FIND_PACKAGE_ARGS** keyword was given, the behavior will be as though **FIND_PACKAGE_ARGS** had been provided, with no additional arguments after it.

NEVER

FetchContent_MakeAvailable() will not call *find_package()*. Any **FIND_PACKAGE_ARGS** given to the *FetchContent_Declare()* call will be ignored.

As a special case, if the *FETCHCONTENT_SOURCE_DIR_<uppercaseName>* variable has a non-empty value for a dependency, it is assumed that the user is overriding all other methods of making that dependency available. **FETCHCONTENT_TRY_FIND_PACKAGE_MODE** will have no effect on that dependency and *FetchContent_MakeAvailable()* will not try to call *find_package()* for it.

In addition to the above, the following variables are also defined for each content name:

FETCHCONTENT_SOURCE_DIR_<uppercaseName>

If this is set, no download or update steps are performed for the specified content and the **<lower-caseName>_SOURCE_DIR** variable returned to the caller is pointed at this location. This gives developers a way to have a separate checkout of the content that they can modify freely without interference from the build. The build simply uses that existing source, but it still defines **<lower-caseName>_BINARY_DIR** to point inside its own build area. Developers are strongly encouraged to use this mechanism rather than editing the sources populated in the default location, as changes to sources in the default location can be lost when content population details are changed by the project.

FETCHCONTENT_UPDATES_DISCONNECTED_<uppercaseName>

This is the per-content equivalent of *FETCHCONTENT_UPDATES_DISCONNECTED*. If the global option or this option is **ON**, then updates for the git and hg methods will not contact any remote for the named content. They will only use information already available locally. Disabling updates for individual content can be useful for content whose details rarely change, while still leaving other frequently changing content with updates enabled.

Examples

Typical Case

This first fairly straightforward example ensures that some popular testing frameworks are available to the main build:

```
include(FetchContent)
FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG         703bd9caab50b139428cea1aaff9974ebee5742e # release-1.10.0
)
FetchContent_Declare(
```

```

    Catch2
    GIT_REPOSITORY https://github.com/catchorg/Catch2.git
    GIT_TAG        605a34765aa5d5ecbf476b4598a862ada971b0cc # v3.0.1
  )

  # After the following call, the CMake targets defined by googletest and
  # Catch2 will be available to the rest of the build
  FetchContent_MakeAvailable(googletest Catch2)

```

Integrating With `find_package()`

For the previous example, if the user wanted to try to find **googletest** and **Catch2** via `find_package()` first before trying to download and build them from source, they could set the `FETCHCONTENT_TRY_FIND_PACKAGE_MODE` variable to **ALWAYS**. This would also affect any other calls to `FetchContent_Declare()` throughout the project, which might not be acceptable. The behavior can be enabled for just these two dependencies instead by adding **FIND_PACKAGE_ARGS** to the declared details and leaving `FETCHCONTENT_TRY_FIND_PACKAGE_MODE` unset, or set to **OPT_IN**:

```

include(FetchContent)
FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG        703bd9caab50b139428cea1aaff9974ebee5742e # release-1.10.0
  FIND_PACKAGE_ARGS NAMES GTest
)
FetchContent_Declare(
  Catch2
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG        605a34765aa5d5ecbf476b4598a862ada971b0cc # v3.0.1
  FIND_PACKAGE_ARGS
)

# This will try calling find_package() first for both dependencies
FetchContent_MakeAvailable(googletest Catch2)

```

For **Catch2**, no additional arguments to `find_package()` are needed, so no additional arguments are provided after the **FIND_PACKAGE_ARGS** keyword. For **googletest**, its package is more commonly called **GTest**, so arguments are added to support it being found by that name.

If the user wanted to disable `FetchContent_MakeAvailable()` from calling `find_package()` for any dependency, even if it provided **FIND_PACKAGE_ARGS** in its declared details, they could set `FETCHCONTENT_TRY_FIND_PACKAGE_MODE` to **NEVER**.

If the project wanted to indicate that these two dependencies should be downloaded and built from source and that `find_package()` calls should be redirected to use the built dependencies, the **OVER-RIDE_FIND_PACKAGE** option should be used when declaring the content details:

```

include(FetchContent)
FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG        703bd9caab50b139428cea1aaff9974ebee5742e # release-1.10.0
  OVERRIDE_FIND_PACKAGE
)
FetchContent_Declare(
  Catch2

```

```

    GIT_REPOSITORY https://github.com/catchorg/Catch2.git
    GIT_TAG          605a34765aa5d5ecbf476b4598a862ada971b0cc # v3.0.1
    OVERRIDE_FIND_PACKAGE
)

# The following will automatically forward through to FetchContent_MakeAvailable(
find_package(googletest)
find_package(Catch2)

```

CMake provides a FindGTest module which defines some variables that older projects may use instead of linking to the imported targets. To support those cases, we can provide an extra file. In keeping with the "first to define, wins" philosophy of **FetchContent**, we only write out that file if something else hasn't already done so.

```

FetchContent_MakeAvailable(googletest)

if(NOT EXISTS ${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/googletest-extra.cmake AND
   NOT EXISTS ${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/googletestExtra.cmake)
  file(WRITE ${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/googletest-extra.cmake
    [=[
    if("${GTEST_LIBRARIES}" STREQUAL "" AND TARGET GTest::gtest)
      set(GTEST_LIBRARIES GTest::gtest)
    endif()
    if("${GTEST_MAIN_LIBRARIES}" STREQUAL "" AND TARGET GTest::gtest_main)
      set(GTEST_MAIN_LIBRARIES GTest::gtest_main)
    endif()
    if("${GTEST_BOTH_LIBRARIES}" STREQUAL "")
      set(GTEST_BOTH_LIBRARIES ${GTEST_LIBRARIES} ${GTEST_MAIN_LIBRARIES})
    endif()
    ]=])
endif()

```

Projects will also likely be using **find_package(GTest)** rather than **find_package(googletest)**, but it is possible to make use of the *CMAKE_FIND_PACKAGE_REDIRECTS_DIR* area to pull in the latter as a dependency of the former. This is likely to be sufficient to satisfy a typical **find_package(GTest)** call.

```

FetchContent_MakeAvailable(googletest)

if(NOT EXISTS ${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/gtest-config.cmake AND
   NOT EXISTS ${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/GTestConfig.cmake)
  file(WRITE ${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/gtest-config.cmake
    [=[
    include(CMakeFindDependencyMacro)
    find_dependency(googletest)
    ]=])
endif()

if(NOT EXISTS ${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/gtest-config-version.cmake AND
   NOT EXISTS ${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/GTestConfigVersion.cmake)
  file(WRITE ${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/gtest-config-version.cmake
    [=[
    include(${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/googletest-config-version.cmake OPTIONAL)
    if(NOT PACKAGE_VERSION_COMPATIBLE)
      include(${CMAKE_FIND_PACKAGE_REDIRECTS_DIR}/googletestConfigVersion.cmake OPTIONAL)
    endif()
    ]=])
endif()

```

```
endif()
]=])
endif()
```

Overriding Where To Find CMakeLists.txt

If the sub-project's **CMakeLists.txt** file is not at the top level of its source tree, the **SOURCE_SUBDIR** option can be used to tell **FetchContent** where to find it. The following example shows how to use that option, and it also sets a variable which is meaningful to the subproject before pulling it into the main build (set as an **INTERNAL** cache variable to avoid problems with policy *CMP0077*):

```
include(FetchContent)
FetchContent_Declare(
  protobuf
  GIT_REPOSITORY https://github.com/protocolbuffers/protobuf.git
  GIT_TAG         ae50d9b9902526efd6c7a1907d09739f959c6297 # v3.15.0
  SOURCE_SUBDIR  cmake
)
set(protobuf_BUILD_TESTS OFF CACHE INTERNAL "")
FetchContent_MakeAvailable(protobuf)
```

Complex Dependency Hierarchies

In more complex project hierarchies, the dependency relationships can be more complicated. Consider a hierarchy where **projA** is the top level project and it depends directly on projects **projB** and **projC**. Both **projB** and **projC** can be built standalone and they also both depend on another project **projD**. **projB** additionally depends on **projE**. This example assumes that all five projects are available on a company git server. The **CMakeLists.txt** of each project might have sections like the following:

projA

```
include(FetchContent)
FetchContent_Declare(
  projB
  GIT_REPOSITORY git@mycompany.com:git/projB.git
  GIT_TAG         4a89dc7e24ff212a7b5167bef7ab079d
)
FetchContent_Declare(
  projC
  GIT_REPOSITORY git@mycompany.com:git/projC.git
  GIT_TAG         4ad4016bd1d8d5412d135cf8ceea1bb9
)
FetchContent_Declare(
  projD
  GIT_REPOSITORY git@mycompany.com:git/projD.git
  GIT_TAG         origin/integrationBranch
)
FetchContent_Declare(
  projE
  GIT_REPOSITORY git@mycompany.com:git/projE.git
  GIT_TAG         v2.3-rc1
)

# Order is important, see notes in the discussion further below
FetchContent_MakeAvailable(projD projB projC)
```

projB

```

include(FetchContent)
FetchContent_Declare(
  projD
  GIT_REPOSITORY git@mycompany.com:git/projD.git
  GIT_TAG        20b415f9034bbd2a2e8216e9a5c9e632
)
FetchContent_Declare(
  projE
  GIT_REPOSITORY git@mycompany.com:git/projE.git
  GIT_TAG        68e20f674a48be38d60e129f600faf7d
)

FetchContent_MakeAvailable(projD projE)

projC

include(FetchContent)
FetchContent_Declare(
  projD
  GIT_REPOSITORY git@mycompany.com:git/projD.git
  GIT_TAG        7d9a17ad2c962aa13e2fbb8043fb6b8a
)

FetchContent_MakeAvailable(projD)

```

A few key points should be noted in the above:

- **projB** and **projC** define different content details for **projD**, but **projA** also defines a set of content details for **projD**. Because **projA** will define them first, the details from **projB** and **projC** will not be used. The override details defined by **projA** are not required to match either of those from **projB** or **projC**, but it is up to the higher level project to ensure that the details it does define still make sense for the child projects.
- In the **projA** call to *FetchContent_MakeAvailable()*, **projD** is listed ahead of **projB** and **projC**, so it will be populated before either **projB** or **projC**. It isn't required for **projA** to do this, doing so ensures that **projA** fully controls the environment in which **projD** is brought into the build (directory properties are particularly relevant).
- While **projA** defines content details for **projE**, it does not need to explicitly call **FetchContent_MakeAvailable(projE)** or **FetchContent_Populate(projD)** itself. Instead, it leaves that to the child **projB**. For higher level projects, it is often enough to just define the override content details and leave the actual population to the child projects. This saves repeating the same thing at each level of the project hierarchy unnecessarily, but it should only be done if directory properties set by dependencies are not expected to influence the population of the shared dependency (**projE** in this case).

Populating Content Without Adding It To The Build

Projects don't always need to add the populated content to the build. Sometimes the project just wants to make the downloaded content available at a predictable location. The next example ensures that a set of standard company toolchain files (and potentially even the toolchain binaries themselves) is available early enough to be used for that same build.

```

cmake_minimum_required(VERSION 3.14)

include(FetchContent)
FetchContent_Declare(
  mycom_toolchains
  URL https://intranet.mycompany.com//toolchains_1.3.2.tar.gz

```

```

)
FetchContent_MakeAvailable(mycom_toolchains)

project(CrossCompileExample)

```

The project could be configured to use one of the downloaded toolchains like so:

```
cmake -DCMAKE_TOOLCHAIN_FILE=_deps/mycom_toolchains-src/toolchain_arm.cmake /path
```

When CMake processes the **CMakeLists.txt** file, it will download and unpack the tarball into **_deps/my-company_toolchains-src** relative to the build directory. The **CMAKE_TOOLCHAIN_FILE** variable is not used until the *project()* command is reached, at which point CMake looks for the named toolchain file relative to the build directory. Because the tarball has already been downloaded and unpacked by then, the toolchain file will be in place, even the very first time that **cmake** is run in the build directory.

Populating Content In CMake Script Mode

This last example demonstrates how one might download and unpack a firmware tarball using CMake's *script mode*. The call to *FetchContent_Populate()* specifies all the content details and the unpacked firmware will be placed in a **firmware** directory below the current working directory.

getFirmware.cmake

```

# NOTE: Intended to be run in script mode with cmake -P
include(FetchContent)
FetchContent_Populate(
  firmware
  URL      https://mycompany.com/assets/firmware-1.23-arm.tar.gz
  URL_HASH MD5=68247684da89b608d466253762b0ff11
  SOURCE_DIR firmware
)

```

FindPackageHandleStandardArgs

This module provides functions intended to be used in *Find Modules* implementing *find_package(<PackageName>)* calls.

find_package_handle_standard_args

This command handles the **REQUIRED**, **QUIET** and version-related arguments of *find_package()*. It also sets the **<PackageName>_FOUND** variable. The package is considered found if all variables listed contain valid results, e.g. valid filepaths.

There are two signatures:

```

find_package_handle_standard_args(<PackageName>
  (DEFAULT_MSG|<custom-failure-message>)
  <required-var>...
)

find_package_handle_standard_args(<PackageName>
  [FOUND_VAR <result-var>]
  [REQUIRED_VARS <required-var>...]
  [VERSION_VAR <version-var>]
  [HANDLE_VERSION_RANGE]
  [HANDLE_COMPONENTS]
  [CONFIG_MODE]
  [NAME_MISMATCHED]
  [REASON_FAILURE_MESSAGE <reason-failure-message>]
)

```

```
[FAIL_MESSAGE <custom-failure-message>]
)
```

The **<PackageName>_FOUND** variable will be set to **TRUE** if all the variables **<required-var>...** are valid and any optional constraints are satisfied, and **FALSE** otherwise. A success or failure message may be displayed based on the results and on whether the **REQUIRED** and/or **QUIET** option was given to the *find_package()* call.

The options are:

(DEFAULT_MSG|<custom-failure-message>)

In the simple signature this specifies the failure message. Use **DEFAULT_MSG** to ask for a default message to be computed (recommended). Not valid in the full signature.

FOUND_VAR <result-var>

Deprecated since version 3.3.

Specifies either **<PackageName>_FOUND** or **<PACKAGENAME>_FOUND** as the result variable. This exists only for compatibility with older versions of CMake and is now ignored. Result variables of both names are now always set for compatibility also with or without this option.

REQUIRED_VARS <required-var>...

Specify the variables which are required for this package. These may be named in the generated failure message asking the user to set the missing variable values. Therefore these should typically be cache entries such as **FOO_LIBRARY** and not output variables like **FOO_LIBRARIES**.

Changed in version 3.18: If **HANDLE_COMPONENTS** is specified, this option can be omitted.

VERSION_VAR <version-var>

Specify the name of a variable that holds the version of the package that has been found. This version will be checked against the (potentially) specified required version given to the *find_package()* call, including its **EXACT** option. The default messages include information about the required version and the version which has been actually found, both if the version is ok or not.

HANDLE_VERSION_RANGE

Added in version 3.19.

Enable handling of a version range, if one is specified. Without this option, a developer warning will be displayed if a version range is specified.

HANDLE_COMPONENTS

Enable handling of package components. In this case, the command will report which components have been found and which are missing, and the **<PackageName>_FOUND** variable will be set to **FALSE** if any of the required components (i.e. not the ones listed after the **OPTIONAL_COMPONENTS** option of *find_package()*) are missing.

CONFIG_MODE

Specify that the calling find module is a wrapper around a call to **find_package(<PackageName> NO_MODULE)**. This implies a **VERSION_VAR** value of **<PackageName>_VERSION**. The command will automatically check whether the package configuration file was found.

REASON_FAILURE_MESSAGE <reason-failure-message>

Added in version 3.16.

Specify a custom message of the reason for the failure which will be appended to the default generated message.

FAIL_MESSAGE <custom-failure-message>

Specify a custom failure message instead of using the default generated message. Not recommended.

NAME_MISMATCHED

Added in version 3.17.

Indicate that the <PackageName> does not match **`\${CMAKE\_FIND\_PACKAGE\_NAME}`**. This is usually a mistake and raises a warning, but it may be intentional for usage of the command for components of a larger package.

Example for the simple signature:

```
find_package_handle_standard_args(LibXml2 DEFAULT_MSG
  LIBXML2_LIBRARY LIBXML2_INCLUDE_DIR)
```

The **LibXml2** package is considered to be found if both **LIBXML2_LIBRARY** and **LIBXML2_INCLUDE_DIR** are valid. Then also **LibXml2_FOUND** is set to **TRUE**. If it is not found and **REQUIRED** was used, it fails with a *message(FATAL_ERROR)*, independent whether **QUIET** was used or not. If it is found, success will be reported, including the content of the first **<required-var>**. On repeated CMake runs, the same message will not be printed again.

NOTE:

If <PackageName> does not match **CMAKE_FIND_PACKAGE_NAME** for the calling module, a warning that there is a mismatch is given. The **FPHSA_NAME_MISMATCHED** variable may be set to bypass the warning if using the old signature and the **NAME_MISMATCHED** argument using the new signature. To avoid forcing the caller to require newer versions of CMake for usage, the variable's value will be used if defined when the **NAME_MISMATCHED** argument is not passed for the new signature (but using both is an error)..

Example for the full signature:

```
find_package_handle_standard_args(LibArchive
  REQUIRED_VARS LibArchive_LIBRARY LibArchive_INCLUDE_DIR
  VERSION_VAR LibArchive_VERSION)
```

In this case, the **LibArchive** package is considered to be found if both **LibArchive_LIBRARY** and **LibArchive_INCLUDE_DIR** are valid. Also the version of **LibArchive** will be checked by using the version contained in **LibArchive_VERSION**. Since no **FAIL_MESSAGE** is given, the default messages will be printed.

Another example for the full signature:

```
find_package(Automoc4 QUIET NO_MODULE HINTS /opt/automoc4)
find_package_handle_standard_args(Automoc4 CONFIG_MODE)
```

In this case, a **FindAutomoc4.cmake** module wraps a call to **find_package(Automoc4 NO_MODULE)** and adds an additional search directory for **automoc4**. Then the call to

find_package_handle_standard_args produces a proper success/failure message.

find_package_check_version

Added in version 3.19.

Helper function which can be used to check if a **<version>** is valid against version-related arguments of *find_package()*.

```
find_package_check_version(<version> <result-var>
    [HANDLE_VERSION_RANGE]
    [RESULT_MESSAGE_VARIABLE <message-var>]
)
```

The **<result-var>** will hold a boolean value giving the result of the check.

The options are:

HANDLE_VERSION_RANGE

Enable handling of a version range, if one is specified. Without this option, a developer warning will be displayed if a version range is specified.

RESULT_MESSAGE_VARIABLE <message-var>

Specify a variable to get back a message describing the result of the check.

Example for the usage:

```
find_package_check_version(1.2.3 result HANDLE_VERSION_RANGE
    RESULT_MESSAGE_VARIABLE reason)
if(result)
    message(STATUS "${reason}")
else()
    message(FATAL_ERROR "${reason}")
endif()
```

FindPackageMessage

```
find_package_message(<name> "message for user" "find result details")
```

This function is intended to be used in FindXXX.cmake modules files. It will print a message once for each unique find result. This is useful for telling the user where a package was found. The first argument specifies the name (XXX) of the package. The second argument specifies the message to display. The third argument lists details about the find result so that if they change the message will be displayed again. The macro also obeys the QUIET argument to the find_package command.

Example:

```
if(X11_FOUND)
    find_package_message(X11 "Found X11: ${X11_X11_LIB}"
        "[${X11_X11_LIB}][${X11_INCLUDE_DIR}]"
    )
else()
    ...
endif()
```

FortranCInterface

Fortran/C Interface Detection

This module automatically detects the API by which C and Fortran languages interact.

Module Variables

Variables that indicate if the mangling is found:

FortranCInterface_GLOBAL_FOUND

Global subroutines and functions.

FortranCInterface_MODULE_FOUND

Module subroutines and functions (declared by "MODULE PROCEDURE").

This module also provides the following variables to specify the detected mangling, though a typical use case does not need to reference them and can use the *Module Functions* below.

FortranCInterface_GLOBAL_PREFIX

Prefix for a global symbol without an underscore.

FortranCInterface_GLOBAL_SUFFIX

Suffix for a global symbol without an underscore.

FortranCInterface_GLOBAL_CASE

The case for a global symbol without an underscore, either **UPPER** or **LOWER**.

FortranCInterface_GLOBAL__PREFIX

Prefix for a global symbol with an underscore.

FortranCInterface_GLOBAL__SUFFIX

Suffix for a global symbol with an underscore.

FortranCInterface_GLOBAL__CASE

The case for a global symbol with an underscore, either **UPPER** or **LOWER**.

FortranCInterface_MODULE_PREFIX

Prefix for a module symbol without an underscore.

FortranCInterface_MODULE_MIDDLE

Middle of a module symbol without an underscore that appears between the name of the module and the name of the symbol.

FortranCInterface_MODULE_SUFFIX

Suffix for a module symbol without an underscore.

FortranCInterface_MODULE_CASE

The case for a module symbol without an underscore, either **UPPER** or **LOWER**.

FortranCInterface_MODULE__PREFIX

Prefix for a module symbol with an underscore.

FortranCInterface_MODULE__MIDDLE

Middle of a module symbol with an underscore that appears between the name of the module and the name of the symbol.

FortranCInterface_MODULE__SUFFIX

Suffix for a module symbol with an underscore.

FortranCInterface_MODULE__CASE

The case for a module symbol with an underscore, either **UPPER** or **LOWER**.

Module Functions

FortranCInterface_HEADER

The **FortranCInterface_HEADER** function is provided to generate a C header file containing macros to mangle symbol names:

```
FortranCInterface_HEADER(<file>
                        [MACRO_NAMESPACE <macro-ns>]
                        [SYMBOL_NAMESPACE <ns>]
                        [SYMBOLS [<module>:]<function> ...])
```

It generates in **<file>** definitions of the following macros:

```
#define FortranCInterface_GLOBAL (name,NAME) ...
#define FortranCInterface_GLOBAL_(name,NAME) ...
#define FortranCInterface_MODULE (mod,name, MOD,NAME) ...
#define FortranCInterface_MODULE_(mod,name, MOD,NAME) ...
```

These macros mangle four categories of Fortran symbols, respectively:

- Global symbols without '_': **call mysub()**
- Global symbols with '_' : **call my_sub()**
- Module symbols without '_': **use mymod; call mysub()**
- Module symbols with '_' : **use mymod; call my_sub()**

If mangling for a category is not known, its macro is left undefined. All macros require raw names in both lower case and upper case.

The options are:

MACRO_NAMESPACE

Replace the default **FortranCInterface_** prefix with a given namespace **<macro-ns>**.

SYMBOLS

List symbols to mangle automatically with C preprocessor definitions:

```
<function>          ==> #define <ns><function> ...
<module>:<function> ==> #define <ns><module>_<function> ...
```

If the mangling for some symbol is not known then no preprocessor definition is created, and a warning is displayed.

SYMBOL_NAMESPACE

Prefix all preprocessor definitions generated by the **SYMBOLS** option with a given namespace **<ns>**.

FortranCInterface_VERIFY

The **FortranCInterface_VERIFY** function is provided to verify that the Fortran and C/C++ compilers work together:

```
FortranCInterface_VERIFY([CXX] [QUIET])
```

It tests whether a simple test executable using Fortran and C (and C++ when the CXX option is given) compiles and links successfully. The result is stored in the cache entry **FortranCInterface_VERIFIED_C** (or **FortranCInterface_VERIFIED_CXX** if CXX is given) as a boolean. If the check fails and **QUIET** is not given the function terminates with a fatal error message describing the problem. The purpose of this check is to stop a build early for incompatible compiler combinations. The test is built in the **Release** configuration.

Example Usage

```
include(FortranCInterface)
FortranCInterface_HEADER(FC.h MACRO_NAMESPACE "FC_")
```

This creates a "FC.h" header that defines mangling macros **FC_GLOBAL()**, **FC_GLOBAL_()**, **FC_MODULE()**, and **FC_MODULE_()**.

```
include(FortranCInterface)
FortranCInterface_HEADER(FCMangle.h
```

```
MACRO_NAMESPACE "FC_"
SYMBOL_NAMESPACE "FC_"
SYMBOLS mysub mymod:my_sub)
```

This creates a "FCMangle.h" header that defines the same **FC_***() mangling macros as the previous example plus preprocessor symbols **FC_mysub** and **FC_mymod_my_sub**.

Additional Manglings

FortranCInterface is aware of possible **GLOBAL** and **MODULE** manglings for many Fortran compilers, but it also provides an interface to specify new possible manglings. Set the variables:

```
FortranCInterface_GLOBAL_SYMBOLS
FortranCInterface_MODULE_SYMBOLS
```

before including FortranCInterface to specify manglings of the symbols **MySub**, **My_Sub**, **MyModule:MySub**, and **My_Module:My_Sub**. For example, the code:

```
set(FortranCInterface_GLOBAL_SYMBOLS mysub_ my_sub_ MYSUB_)
#          ^^^^^ ^^^^^ ^^^^^
set(FortranCInterface_MODULE_SYMBOLS
__mymodule_MOD_mysub __my_module_MOD_my_sub)
#          ^^^^^ ^^^^^ ^^^^^
include(FortranCInterface)
```

tells FortranCInterface to try given **GLOBAL** and **MODULE** manglings. (The carets point at raw symbol names for clarity in this example but are not needed.)

GenerateExportHeader

This module provides the **generate_export_header()** function to generate export macros for libraries.

Added in version 3.12: Added support for C projects. Previous versions supported C++ project only.

generate_export_header

The **generate_export_header()** function can be used to generate a file suitable for preprocessor inclusion which contains EXPORT macros to be used in library classes:

```
generate_export_header(LIBRARY_TARGET
    [BASE_NAME <base_name>]
    [EXPORT_MACRO_NAME <export_macro_name>]
    [EXPORT_FILE_NAME <export_file_name>]
    [DEPRECATED_MACRO_NAME <deprecated_macro_name>]
    [NO_EXPORT_MACRO_NAME <no_export_macro_name>]
    [INCLUDE_GUARD_NAME <include_guard_name>]
    [STATIC_DEFINE <static_define>]
    [NO_DEPRECATED_MACRO_NAME <no_deprecated_macro_name>]
    [DEFINE_NO_DEPRECATED]
    [PREFIX_NAME <prefix_name>]
    [CUSTOM_CONTENT_FROM_VARIABLE <variable>]
)
```

The target properties **CXX_VISIBILITY_PRESET** and **VISIBILITY_INLINES_HIDDEN** can be used to add the appropriate compile flags for targets. See the documentation of those target properties, and the convenience variables **CMAKE_CXX_VISIBILITY_PRESET** and **CMAKE_VISIBILITY_INLINES_HIDDEN**.

By default **generate_export_header()** generates macro names in a file name determined by the name of the

library. This means that in the simplest case, users of **GenerateExportHeader** will be equivalent to:

```
set(CMAKE_CXX_VISIBILITY_PRESET hidden)
set(CMAKE_VISIBILITY_INLINES_HIDDEN 1)
add_library(somelib someclass.cpp)
generate_export_header(somelib)
install(TARGETS somelib DESTINATION ${LIBRARY_INSTALL_DIR})
install(FILES
    someclass.h
    ${PROJECT_BINARY_DIR}/somelib_export.h DESTINATION ${INCLUDE_INSTALL_DIR}
)
```

And in the ABI header files:

```
#include "somelib_export.h"
class SOMELIB_EXPORT SomeClass {
    ...
};
```

The CMake fragment will generate a file in the `${CMAKE_CURRENT_BINARY_DIR}` called **somelib_export.h** containing the macros **SOMELIB_EXPORT**, **SOMELIB_NO_EXPORT**, **SOMELIB_DEPRECATED**, **SOMELIB_DEPRECATED_EXPORT** and **SOMELIB_DEPRECATED_NO_EXPORT**. They will be followed by content taken from the variable specified by the **CUSTOM_CONTENT_FROM_VARIABLE** option, if any. The resulting file should be installed with other headers in the library.

The **BASE_NAME** argument can be used to override the file name and the names used for the macros:

```
add_library(somelib someclass.cpp)
generate_export_header(somelib
    BASE_NAME other_name
)
```

Generates a file called **other_name_export.h** containing the macros **OTHER_NAME_EXPORT**, **OTHER_NAME_NO_EXPORT** and **OTHER_NAME_DEPRECATED** etc.

The **BASE_NAME** may be overridden by specifying other options in the function. For example:

```
add_library(somelib someclass.cpp)
generate_export_header(somelib
    EXPORT_MACRO_NAME OTHER_NAME_EXPORT
)
```

creates the macro **OTHER_NAME_EXPORT** instead of **SOMELIB_EXPORT**, but other macros and the generated file name is as default:

```
add_library(somelib someclass.cpp)
generate_export_header(somelib
    DEPRECATED_MACRO_NAME KDE_DEPRECATED
)
```

creates the macro **KDE_DEPRECATED** instead of **SOMELIB_DEPRECATED**.

If **LIBRARY_TARGET** is a static library, macros are defined without values.

If the same sources are used to create both a shared and a static library, the uppercased symbol **`${BASE_NAME}_STATIC_DEFINE`** should be used when building the static library:

```
add_library(shared_variant SHARED ${lib_SRCS})
add_library(static_variant ${lib_SRCS})
generate_export_header(shared_variant BASE_NAME libshared_and_static)
set_target_properties(static_variant PROPERTIES
    COMPILE_FLAGS -DLIBSHARED_AND_STATIC_STATIC_DEFINE)
```

This will cause the export macros to expand to nothing when building the static library.

If **`DEFINE_NO_DEPRECATED`** is specified, then a macro **`${BASE_NAME}_NO_DEPRECATED`** will be defined. This macro can be used to remove deprecated code from preprocessor output:

```
option(EXCLUDE_DEPRECATED "Exclude deprecated parts of the library" FALSE)
if (EXCLUDE_DEPRECATED)
    set(NO_BUILD_DEPRECATED DEFINE_NO_DEPRECATED)
endif()
generate_export_header(somelib ${NO_BUILD_DEPRECATED})
```

And then in `somelib`:

```
class SOMELIB_EXPORT SomeClass
{
public:
#ifdef SOMELIB_NO_DEPRECATED
    SOMELIB_DEPRECATED void oldMethod();
#endif
};

#ifdef SOMELIB_NO_DEPRECATED
void SomeClass::oldMethod() { }
#endif
```

If **`PREFIX_NAME`** is specified, the argument will be used as a prefix to all generated macros.

For example:

```
generate_export_header(somelib PREFIX_NAME VTK_)
```

Generates the macros **`VTK_SOMELIB_EXPORT`** etc.

Added in version 3.1: Library target can be an **`OBJECT`** library.

Added in version 3.7: Added the **`CUSTOM_CONTENT_FROM_VARIABLE`** option.

Added in version 3.11: Added the **`INCLUDE_GUARD_NAME`** option.

`add_compiler_export_flags`

```
add_compiler_export_flags([<output_variable>])
```

Deprecated since version 3.0: Set the target properties `CXX_VISIBILITY_PRESET` and `VISIBILITY_INLINES_HIDDEN` instead.

The `add_compiler_export_flags()` function adds `-fvisibility=hidden` to `CMAKE_CXX_FLAGS` if supported, and is a no-op on Windows which does not need extra compiler flags for exporting support. You may optionally pass a single argument to `add_compiler_export_flags()` that will be populated with the `CXX_FLAGS` required to enable visibility support for the compiler/architecture in use.

GNUInstallDirs

Define GNU standard installation directories

Provides install directory variables as defined by the *GNU Coding Standards*.

Result Variables

Inclusion of this module defines the following variables:

CMAKE_INSTALL_<dir>

Destination for files of a given type. This value may be passed to the **DESTINATION** options of `install()` commands for the corresponding file type. It should be a path relative to the installation prefix so that it can be converted to an absolute path in a relocatable way.

While absolute paths are allowed, they are not recommended as they do not work with the **cmake** `--install` command's `--prefix` option, or with the *cpack* installer generators. 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.

CMAKE_INSTALL_FULL_<dir>

The absolute path generated from the corresponding **CMAKE_INSTALL_<dir>** value. If the value is not already an absolute path, an absolute path is constructed typically by prepending the value of the `CMAKE_INSTALL_PREFIX` variable. However, there are some *special cases* as documented below.

These variables shouldn't be used in `install()` commands as they do not work with the **cmake** `--install` command's `--prefix` option, or with the *cpack* installer generators.

where <dir> is one of:

BINDIR

user executables (**bin**)

SBINDIR

system admin executables (**sbin**)

LIBEXECDIR

program executables (**libexec**)

SYSCONFDIR

read-only single-machine data (**etc**)

SHAREDSTATEDIR

modifiable architecture-independent data (**com**)

LOCALSTATEDIR

modifiable single-machine data (**var**)

RUNSTATEDIR

Added in version 3.9: run-time variable data (**LOCALSTATEDIR/run**)

LIBDIR

object code libraries (**lib** or **lib64**)

On Debian, this may be **lib**/**<multiarch-tuple>** when *CMAKE_INSTALL_PREFIX* is **/usr**.

INCLUDEDIR

C header files (**include**)

OLDINCLUDEDIR

C header files for non-gcc (**/usr/include**)

DATAROOTDIR

read-only architecture-independent data root (**share**)

DATADIR

read-only architecture-independent data (**DATAROOTDIR**)

INFODIR

info documentation (**DATAROOTDIR/info**)

LOCALEDIR

locale-dependent data (**DATAROOTDIR/locale**)

MANDIR

man documentation (**DATAROOTDIR/man**)

DOCDIR

documentation root (**DATAROOTDIR/doc/PROJECT_NAME**)

If the includer does not define a value the above-shown default will be used and the value will appear in the cache for editing by the user.

Special Cases

Added in version 3.4.

The following values of *CMAKE_INSTALL_PREFIX* are special:

/

For **<dir>** other than the **SYSCONFDIR**, **LOCALSTATEDIR** and **RUNSTATEDIR**, the value of **CMAKE_INSTALL_<dir>** is prefixed with **usr/** if it is not user-specified as an absolute path. For example, the **INCLUDEDIR** value **include** becomes **usr/include**. This is required by the *GNU Coding Standards*, which state:

When building the complete GNU system, the prefix will be empty and **/usr** will be a symbolic link to **/**.

/usr

For **<dir>** equal to **SYSCONFDIR**, **LOCALSTATEDIR** or **RUNSTATEDIR**, the **CMAKE_INSTALL_FULL_<dir>** is computed by prepending just **/** to the value of **CMAKE_INSTALL_<dir>** if it is not user-specified as an absolute path. For example, the **SYSCONFDIR** value **etc** becomes **/etc**. This is required by the *GNU Coding Standards*.

/opt/...

For **<dir>** equal to **SYSCONFDIR**, **LOCALSTATEDIR** or **RUNSTATEDIR**, the **CMAKE_INSTALL_FULL_<dir>** is computed by *appending* the prefix to the value of **CMAKE_INSTALL_<dir>** if it is not user-specified as an absolute path. For example, the **SYSCONFDIR** value **etc** becomes **/etc/opt/...** This is defined by the *F filesystem Hierarchy Standard*.

This behavior does not apply to paths under **/opt/homebrew/...**

Macros**GNUInstallDirs_get_absolute_install_dir**

```
GNUInstallDirs_get_absolute_install_dir(absvar var dirname)
```

Added in version 3.7.

Set the given variable **absvar** to the absolute path contained within the variable **var**. This is to allow the computation of an absolute path, accounting for all the special cases documented above. While this macro is used to compute the various **CMAKE_INSTALL_FULL_<dir>** variables, it is exposed publicly to allow users who create additional path variables to also compute absolute paths where necessary, using the same logic. **dirname** is the directory name to get, e.g. **BINDIR**.

Changed in version 3.20: Added the **<dirname>** parameter. Previous versions of CMake passed this value through the variable **\${dir}**.

GoogleTest

Added in version 3.9.

This module defines functions to help use the Google Test infrastructure. Two mechanisms for adding tests are provided. *gtest_add_tests()* has been around for some time, originally via **find_package(GTest)**. *gtest_discover_tests()* was introduced in CMake 3.10.

The (older) *gtest_add_tests()* scans source files to identify tests. This is usually effective, with some caveats, including in cross-compiling environments, and makes setting additional properties on tests more convenient. However, its handling of parameterized tests is less comprehensive, and it requires re-running CMake to detect changes to the list of tests.

The (newer) *gtest_discover_tests()* discovers tests by asking the compiled test executable to enumerate its tests. This is more robust and provides better handling of parameterized tests, and does not require CMake to be re-run when tests change. However, it may not work in a cross-compiling environment, and setting test properties is less convenient.

More details can be found in the documentation of the respective functions.

Both commands are intended to replace use of *add_test()* to register tests, and will create a separate CTest test for each Google Test test case. Note that this is in some cases less efficient, as common set-up and tear-down logic cannot be shared by multiple test cases executing in the same instance. However, it provides more fine-grained pass/fail information to CTest, which is usually considered as more beneficial. By default, the CTest test name is the same as the Google Test name (i.e. **suite.testcase**); see also **TEST_PREFIX** and **TEST_SUFFIX**.

gtest_add_tests

Automatically add tests with CTest by scanning source code for Google Test macros:

```
gtest_add_tests(TARGET target
                [SOURCES src1...]
                [EXTRA_ARGS args...]
                [WORKING_DIRECTORY dir]
                [TEST_PREFIX prefix]
                [TEST_SUFFIX suffix]
                [SKIP_DEPENDENCY]
                [TEST_LIST outVar]
```

)

gtest_add_tests attempts to identify tests by scanning source files. Although this is generally effective, it uses only a basic regular expression match, which can be defeated by atypical test declarations, and is unable to fully "split" parameterized tests. Additionally, it requires that CMake be re-run to discover any newly added, removed or renamed tests (by default, this means that CMake is re-run when any test source file is changed, but see **SKIP_DEPENDENCY**). However, it has the advantage of declaring tests at CMake time, which somewhat simplifies setting additional properties on tests, and always works in a cross-compiling environment.

The options are:

TARGET target

Specifies the Google Test executable, which must be a known CMake executable target. CMake will substitute the location of the built executable when running the test.

SOURCES src1...

When provided, only the listed files will be scanned for test cases. If this option is not given, the *SOURCES* property of the specified **target** will be used to obtain the list of sources.

EXTRA_ARGS args...

Any extra arguments to pass on the command line to each test case.

Changed in version 3.31: Empty values in **args...** are preserved, see *CMP0178*.

WORKING_DIRECTORY dir

Specifies the directory in which to run the discovered test cases. If this option is not provided, the current binary directory is used.

TEST_PREFIX prefix

Specifies a **prefix** to be prepended to the name of each discovered test case. This can be useful when the same source files are being used in multiple calls to **gtest_add_test()** but with different **EXTRA_ARGS**.

TEST_SUFFIX suffix

Similar to **TEST_PREFIX** except the **suffix** is appended to the name of every discovered test case. Both **TEST_PREFIX** and **TEST_SUFFIX** may be specified.

SKIP_DEPENDENCY

Normally, the function creates a dependency which will cause CMake to be re-run if any of the sources being scanned are changed. This is to ensure that the list of discovered tests is updated. If this behavior is not desired (as may be the case while actually writing the test cases), this option can be used to prevent the dependency from being added.

TEST_LIST outVar

The variable named by **outVar** will be populated in the calling scope with the list of discovered test cases. This allows the caller to do things like manipulate test properties of the discovered tests.

Changed in version 3.31: Empty values in the *TEST_LAUNCHER* and *CROSSCOMPILING_EMULATOR* target properties are preserved, see policy *CMP0178*.

Usage example:

```
include(GoogleTest)
add_executable(FooTest FooUnitTest.cxx)
```

```

gtest_add_tests(TARGET      FooTest
                TEST_SUFFIX  .noArgs
                TEST_LIST    noArgsTests
            )
gtest_add_tests(TARGET      FooTest
                EXTRA_ARGS  --someArg someValue
                TEST_SUFFIX  .withArgs
                TEST_LIST    withArgsTests
            )
set_tests_properties(${noArgsTests}  PROPERTIES TIMEOUT 10)
set_tests_properties(${withArgsTests} PROPERTIES TIMEOUT 20)

```

For backward compatibility, the following form is also supported:

```
gtest_add_tests(exe args files...)
```

- exe** The path to the test executable or the name of a CMake target.
- args** A ;-list of extra arguments to be passed to executable. The entire list must be passed as a single argument. Enclose it in quotes, or pass "" for no arguments.
- files...** A list of source files to search for tests and test fixtures. Alternatively, use **AUTO** to specify that **exe** is the name of a CMake executable target whose sources should be scanned.

```

include(GoogleTest)
set(FooTestArgs --foo 1 --bar 2)
add_executable(FooTest FooUnitTest.cxx)
gtest_add_tests(FooTest "${FooTestArgs}" AUTO)

```

gtest_discover_tests

Automatically add tests with CTest by querying the compiled test executable for available tests:

```

gtest_discover_tests(target
    [EXTRA_ARGS args...]
    [WORKING_DIRECTORY dir]
    [TEST_PREFIX prefix]
    [TEST_SUFFIX suffix]
    [TEST_FILTER expr]
    [NO_PRETTY_TYPES] [NO_PRETTY_VALUES]
    [PROPERTIES name1 value1...]
    [TEST_LIST var]
    [DISCOVERY_TIMEOUT seconds]
    [XML_OUTPUT_DIR dir]
    [DISCOVERY_MODE <POST_BUILD|PRE_TEST>]
    [DISCOVERY_EXTRA_ARGS args...]
)

```

Added in version 3.10.

gtest_discover_tests() sets up a post-build or pre-test command on the test executable that generates the list of tests by parsing the output from running the test executable with the **--gtest_list_tests** argument. Compared to the source parsing approach of *gtest_add_tests()*, this ensures that the full list of tests, including instantiations of parameterized tests, is obtained. Since test discovery occurs at build or test time, it is not necessary to re-run CMake when the list of tests changes. However, it requires that *CROSSCOMPILING_EMULATOR* is properly set in order to

function in a cross-compiling environment.

Additionally, setting properties on tests is somewhat less convenient, since the tests are not available at CMake time. Additional test properties may be assigned to the set of tests as a whole using the **PROPERTIES** option. If more fine-grained test control is needed, custom content may be provided through an external CTest script using the *TEST_INCLUDE_FILES* directory property. The set of discovered tests is made accessible to such a script via the *<target>_TESTS* variable (see the **TEST_LIST** option below for further discussion and limitations).

The options are:

target Specifies the Google Test executable, which must be a known CMake executable target. CMake will substitute the location of the built executable when running the test.

EXTRA_ARGS args...

Any extra arguments to pass on the command line to each test case.

Changed in version 3.31: Empty values in **args...** are preserved, see *CMP0178*.

WORKING_DIRECTORY dir

Specifies the directory in which to run the discovered test cases. If this option is not provided, the current binary directory is used.

TEST_PREFIX prefix

Specifies a **prefix** to be prepended to the name of each discovered test case. This can be useful when the same test executable is being used in multiple calls to **gtest_discover_tests()** but with different **EXTRA_ARGS**.

TEST_SUFFIX suffix

Similar to **TEST_PREFIX** except the **suffix** is appended to the name of every discovered test case. Both **TEST_PREFIX** and **TEST_SUFFIX** may be specified.

TEST_FILTER expr

Added in version 3.22.

Filter expression to pass as a **--gtest_filter** argument during test discovery. Note that the expression is a wildcard-based format that matches against the original test names as used by gtest. For type or value-parameterized tests, these names may be different to the potentially pretty-printed test names that **ctest** uses.

NO_PRETTY_TYPES

By default, the type index of type-parameterized tests is replaced by the actual type name in the CTest test name. If this behavior is undesirable (e.g. because the type names are unwieldy), this option will suppress this behavior.

NO_PRETTY_VALUES

By default, the value index of value-parameterized tests is replaced by the actual value in the CTest test name. If this behavior is undesirable (e.g. because the value strings are unwieldy), this option will suppress this behavior.

PROPERTIES name1 value1...

Specifies additional properties to be set on all tests discovered by this invocation of **gtest_discover_tests()**.

TEST_LIST var

Make the list of tests available in the variable **var**, rather than the default *<target>_TESTS*. This can be useful when the same test executable is being used in multiple calls to **gtest_discover_tests()**. Note that this variable is only available in CTest.

Due to a limitation of CMake's parsing rules, any test with a square bracket in its name will be omitted from the list of tests stored in this variable. Such tests will still be defined and executed by **ctest** as normal though.

DISCOVERY_TIMEOUT num

Added in version 3.10.3.

Specifies how long (in seconds) CMake will wait for the test to enumerate available tests. If the test takes longer than this, discovery (and your build) will fail. Most test executables will enumerate their tests very quickly, but under some exceptional circumstances, a test may require a longer timeout. The default is 5. See also the **TIMEOUT** option of *execute_process()*.

NOTE:

In CMake versions 3.10.1 and 3.10.2, this option was called **TIMEOUT**. This clashed with the **TIMEOUT** test property, which is one of the common properties that would be set with the **PROPERTIES** keyword, usually leading to legal but unintended behavior. The keyword was changed to **DISCOVERY_TIMEOUT** in CMake 3.10.3 to address this problem. The ambiguous behavior of the **TIMEOUT** keyword in 3.10.1 and 3.10.2 has not been preserved.

XML_OUTPUT_DIR dir

Added in version 3.18.

If specified, the parameter is passed along with **--gtest_output=xml:** to test executable. The actual file name is the same as the test target, including prefix and suffix. This should be used instead of **EXTRA_ARGS --gtest_output=xml** to avoid race conditions writing the XML result output when using parallel test execution.

DISCOVERY_MODE

Added in version 3.18.

Provides greater control over when **gtest_discover_tests()** performs test discovery. By default, **POST_BUILD** sets up a post-build command to perform test discovery at build time. In certain scenarios, like cross-compiling, this **POST_BUILD** behavior is not desirable. By contrast, **PRE_TEST** delays test discovery until just prior to test execution. This way test discovery occurs in the target environment where the test has a better chance at finding appropriate runtime dependencies.

DISCOVERY_MODE defaults to the value of the **CMAKE_GTEST_DISCOVER_TESTS_DISCOVERY_MODE** variable if it is not passed when calling **gtest_discover_tests()**. This provides a mechanism for globally selecting a preferred test discovery behavior without having to modify each call site.

DISCOVERY_EXTRA_ARGS args...

Added in version 3.31.

Any extra arguments to pass on the command line for the discovery command.

Added in version 3.29: The **TEST_LAUNCHER** target property is honored during test discovery and test execution.

Changed in version 3.31: Empty values in the `TEST_LAUNCHER` and `CROSSCOMPILING_EMULATOR` target properties are preserved, see policy `CMP0178`.

InstallRequiredSystemLibraries

Include this module to search for compiler-provided system runtime libraries and add install rules for them. Some optional variables may be set prior to including the module to adjust behavior:

CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS

Specify additional runtime libraries that may not be detected. After inclusion any detected libraries will be appended to this.

CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_SKIP

Set to TRUE to skip calling the `install(PROGRAMS)` command to allow the includer to specify its own install rule, using the value of `CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS` to get the list of libraries.

CMAKE_INSTALL_DEBUG_LIBRARIES

Set to TRUE to install the debug runtime libraries when available with MSVC tools.

CMAKE_INSTALL_DEBUG_LIBRARIES_ONLY

Set to TRUE to install only the debug runtime libraries with MSVC tools even if the release runtime libraries are also available.

CMAKE_INSTALL_UCRT_LIBRARIES

Added in version 3.6.

Set to TRUE to install the Windows Universal CRT libraries for app-local deployment (e.g. to Windows XP). This is meaningful only with MSVC from Visual Studio 2015 or higher.

Added in version 3.9: One may set a `CMAKE_WINDOWS_KITS_10_DIR` environment variable to an absolute path to tell CMake to look for Windows 10 SDKs in a custom location. The specified directory is expected to contain `Redist/ucrt/DLLs/*` directories.

CMAKE_INSTALL_MFC_LIBRARIES

Set to TRUE to install the MSVC MFC runtime libraries.

CMAKE_INSTALL_OPENMP_LIBRARIES

Set to TRUE to install the MSVC OpenMP runtime libraries

CMAKE_INSTALL_SYSTEM_RUNTIME_DESTINATION

Specify the `install(PROGRAMS)` command `DESTINATION` option. If not specified, the default is `bin` on Windows and `lib` elsewhere.

CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS_NO_WARNINGS

Set to TRUE to disable warnings about required library files that do not exist. (For example, Visual Studio Express editions may not provide the redistributable files.)

CMAKE_INSTALL_SYSTEM_RUNTIME_COMPONENT

Added in version 3.3.

Specify the `install(PROGRAMS)` command `COMPONENT` option. If not specified, no such option will be used.

Added in version 3.10: Support for installing Intel compiler runtimes.

ProcessorCount

ProcessorCount(var)

Determine the number of processors/cores and save value in \${var}

Sets the variable named \${var} to the number of physical cores available on the machine if the information can be determined. Otherwise it is set to 0. Currently this functionality is implemented for AIX, cygwin, FreeBSD, HPUX, Linux, macOS, QNX, Sun and Windows.

Changed in version 3.15: On Linux, returns the container CPU count instead of the host CPU count.

This function is guaranteed to return a positive integer (≥ 1) if it succeeds. It returns 0 if there's a problem determining the processor count.

More generally accurate physical CPU count can be obtained via *cmake_host_system_information()*:

```
cmake_host_system_information(RESULT N
                              QUERY NUMBER_OF_PHYSICAL_CORES)
```

Example use, in a ctest -S dashboard script:

```
include(ProcessorCount)
ProcessorCount(N)
if(NOT N EQUAL 0)
  set(CTEST_BUILD_FLAGS -j${N})
  set(ctest_test_args ${ctest_test_args} PARALLEL_LEVEL ${N})
endif()
```

This function is intended to offer an approximation of the value of the number of compute cores available on the current machine, such that you may use that value for parallel building and parallel testing. It is meant to help utilize as much of the machine as seems reasonable. Of course, knowledge of what else might be running on the machine simultaneously should be used when deciding whether to request a machine's full capacity all for yourself.

SelectLibraryConfigurations

select_library_configurations(basename)

This macro takes a library base name as an argument, and will choose good values for the variables

```
basename_LIBRARY
basename_LIBRARIES
basename_LIBRARY_DEBUG
basename_LIBRARY_RELEASE
```

depending on what has been found and set.

If only **basename_LIBRARY_RELEASE** is defined, **basename_LIBRARY** will be set to the release value, and **basename_LIBRARY_DEBUG** will be set to **basename_LIBRARY_DEBUG-NOTFOUND**. If only **basename_LIBRARY_DEBUG** is defined, then **basename_LIBRARY** will take the debug value, and **basename_LIBRARY_RELEASE** will be set to **basename_LIBRARY_RELEASE-NOTFOUND**.

If the generator supports configuration types, then **basename_LIBRARY** and **basename_LIBRARIES** will be set with debug and optimized flags specifying the library to be used for the given configuration. If no build type has been set or the generator in use does not support configuration types, then

basename_LIBRARY and **basename_LIBRARIES** will take only the release value, or the debug value if the release one is not set.

TestForANSIForScope

Check for ANSI for scope support

Check if the compiler restricts the scope of variables declared in a for-init-statement to the loop body.

CMAKE_NO_ANSI_FOR_SCOPE - holds result

TestForANSIStreamHeaders

Test for compiler support of ANSI stream headers iostream, etc.

check if the compiler supports the standard ANSI iostream header (without the .h)

CMAKE_NO_ANSI_STREAM_HEADERS - defined by the results

TestForSSTREAM

Test for compiler support of ANSI sstream header

check if the compiler supports the standard ANSI sstream header

CMAKE_NO_ANSI_STRING_STREAM - defined by the results

TestForSTDNamespace

Test for std:: namespace support

check if the compiler supports std:: on stl classes

CMAKE_NO_STD_NAMESPACE - defined by the results

UseEcos

This module defines variables and macros required to build eCos application.

This file contains the following macros:

ECOS_ADD_INCLUDE_DIRECTORIES()

add the eCos include dirs

ECOS_ADD_EXECUTABLE(name source1 ... sourceN)

create an eCos executable

ECOS_ADJUST_DIRECTORY(VAR source1 ... sourceN)

adjusts the path of the source files and puts the result into **VAR**

Macros for selecting the toolchain:

ECOS_USE_ARM_ELF_TOOLS()

enable the ARM ELF toolchain for the directory where it is called

ECOS_USE_I386_ELF_TOOLS()

enable the i386 ELF toolchain for the directory where it is called

ECOS_USE_PPC_EABI_TOOLS()

enable the PowerPC toolchain for the directory where it is called

It contains the following variables:

ECOS_DEFINITIONS

ECOSCONFIG_EXECUTABLE

ECOS_CONFIG_FILE

defaults to `ecos.ecc`, if your eCos configuration file has a different name, adjust this variable for internal use only:

ECOS_ADD_TARGET_LIB**UseJava**

This file provides support for **Java**. It is assumed that `FindJava` has already been loaded. See `FindJava` for information on how to load Java into your CMake project.

Synopsis*Creating and Installing JARS*

```
add_jar (<target_name> [SOURCES] <source1> [<source2>...] ...)
```

```
install_jar (<target_name> DESTINATION <destination> [COMPONENT <component>])
```

```
install_jni_symlink (<target_name> DESTINATION <destination> [COMPONENT <component>])
```

Header Generation

```
create_javah ((TARGET <target> | GENERATED_FILES <VAR>) CLASSES <class>... ...)
```

Exporting JAR Targets

```
install_jar_exports (TARGETS <jars>... FILE <filename> DESTINATION <destination>)
```

```
export_jars (TARGETS <jars>... [NAMESPACE <namespace>] FILE <filename>)
```

Finding JARs

```
find_jar (<VAR> NAMES <name1> [<name2>...] [PATHS <path1> [<path2>... ENV <var>])
```

Creating Java Documentation

```
create_javadoc (<VAR> (PACKAGES <pkg1> [<pkg2>...] | FILES <file1> [<file2>...])
```

Creating And Installing JARs**add_jar**

Creates a jar file containing java objects and, optionally, resources:

```
add_jar(<target_name>
  [SOURCES] <source1> [<source2>...] [<resource1>...]
  [RESOURCES NAMESPACE <ns1> <resource1>... [NAMESPACE <nsX> <resourceX>...]]
  [INCLUDE_JARS <jar1> [<jar2>...]]
  [ENTRY_POINT <entry>]
  [VERSION <version>]
  [MANIFEST <manifest>]
  [OUTPUT_NAME <name>]
  [OUTPUT_DIR <dir>]
  [GENERATE_NATIVE_HEADERS <target>
    [DESTINATION (<dir>|INSTALL <dir> [BUILD <build>...]]
  )
)
```

This command creates a `<target_name>.jar`. It compiles the given `<source>` files and adds the given `<resource>` files to the jar file. Source files can be java files or listing files (prefixed by `@`). If only resource files are given then just a jar file is created.

SOURCES

Compiles the specified source files and adds the result in the jar file.

Added in version 3.4: Support for resource files, prefixed by `@`.

RESOURCES

Added in version 3.21.

Adds the named **<resource>** files to the jar by stripping the source file path and placing the file beneath **<ns>** within the jar.

For example:

```
RESOURCES NAMESPACE "/com/my/namespace" "a/path/to/resource.txt"
```

results in a resource accessible via **/com/my/namespace/resource.txt** within the jar.

Resources may be added without adjusting the namespace by adding them to the list of **SOURCES** (original behavior), in this case, resource paths must be relative to **CMAKE_CURRENT_SOURCE_DIR**. Adding resources without using the **RESOURCES** parameter in out of source builds will almost certainly result in confusion.

NOTE:

Adding resources via the **SOURCES** parameter relies upon a hard-coded list of file extensions which are tested to determine whether they compile (e.g. File.java). **SOURCES** files which match the extensions are compiled. Files which do not match are treated as resources. To include uncompiled resources matching those file extensions use the **RESOURCES** parameter.

INCLUDE_JARS

The list of jars are added to the classpath when compiling the java sources and also to the dependencies of the target. **INCLUDE_JARS** also accepts other target names created by **add_jar()**. For backwards compatibility, jar files listed as sources are ignored (as they have been since the first version of this module).

ENTRY_POINT

Defines an entry point in the jar file.

VERSION

Adds a version to the target output name.

The following example will create a jar file with the name **shibboleet-1.2.0.jar** and will create a symlink **shibboleet.jar** pointing to the jar with the version information.

```
add_jar(shibboleet shibbotleet.java VERSION 1.2.0)
```

MANIFEST

Defines a custom manifest for the jar.

OUTPUT_NAME

Specify a different output name for the target.

OUTPUT_DIR

Sets the directory where the jar file will be generated. If not specified, **CMAKE_CURRENT_BINARY_DIR** is used as the output directory.

GENERATE_NATIVE_HEADERS

Added in version 3.11.

Generates native header files for methods declared as native. These files provide the connective glue that allow your Java and C code to interact. An **INTERFACE** target will be created for an easy usage of generated files. Sub-option **DESTINATION** can be used to

specify the output directory for generated header files.

This option requires, at least, version 1.8 of the JDK.

For an optimum usage of this option, it is recommended to include module JNI before any call to **add_jar()**. The produced target for native headers can then be used to compile C/C++ sources with the *target_link_libraries()* command.

```
find_package(JNI)
add_jar(foo foo.java GENERATE_NATIVE_HEADERS foo-native)
add_library(bar bar.cpp)
target_link_libraries(bar PRIVATE foo-native)
```

Added in version 3.20: **DESTINATION** sub-option now supports the possibility to specify different output directories for **BUILD** and **INSTALL** steps. If **BUILD** directory is not specified, a default directory will be used.

To export the interface target generated by **GENERATE_NATIVE_HEADERS** option, sub-option **INSTALL** of **DESTINATION** is required:

```
add_jar(foo foo.java GENERATE_NATIVE_HEADERS foo-native
          DESTINATION INSTALL include)
install(TARGETS foo-native EXPORT native)
install(DIRECTORY "$<TARGET_PROPERTY:foo-native,NATIVE_HEADERS_DIRECTORY>"
          DESTINATION include)
install(EXPORT native DESTINATION /to/export NAMESPACE foo)
```

Some variables can be set to customize the behavior of **add_jar()** as well as the java compiler:

CMAKE_JAVA_COMPILE_FLAGS

Specify additional flags to java compiler.

CMAKE_JAVA_INCLUDE_PATH

Specify additional paths to the class path.

CMAKE_JNI_TARGET

If the target is a JNI library, sets this boolean variable to **TRUE** to enable creation of a JNI symbolic link (see also *install_jni_symlink()*).

CMAKE_JAR_CLASSES_PREFIX

If multiple jars should be produced from the same java source filetree, to prevent the accumulation of duplicate class files in subsequent jars, set/reset **CMAKE_JAR_CLASSES_PREFIX** prior to calling the **add_jar()**:

```
set(CMAKE_JAR_CLASSES_PREFIX com/redhat/foo)
add_jar(foo foo.java)

set(CMAKE_JAR_CLASSES_PREFIX com/redhat/bar)
add_jar(bar bar.java)
```

The **add_jar()** function sets the following target properties on **<target_name>**:

INSTALL_FILES

The files which should be installed. This is used by *install_jar()*.

JNI_SYMLINK

The JNI symlink which should be installed. This is used by *install_jni_symlink()*.

JAR_FILE

The location of the jar file so that you can include it.

CLASSDIR

The directory where the class files can be found. For example to use them with **javah**.

NATIVE_HEADERS_DIRECTORY

Added in version 3.20.

The directory where native headers are generated. Defined when option **GENERATE_NATIVE_HEADERS** is specified.

install_jar

This command installs the jar file to the given destination:

```
install_jar(<target_name> <destination>)
install_jar(<target_name> DESTINATION <destination> [COMPONENT <component>]
```

This command installs the **<target_name>** file to the given **<destination>**. It should be called in the same scope as *add_jar()* or it will fail.

Added in version 3.4: The second signature with **DESTINATION** and **COMPONENT** options.

DESTINATION

Specify the directory on disk to which a file will be installed.

COMPONENT

Specify an installation component name with which the install rule is associated, such as "runtime" or "development".

The **install_jar()** command sets the following target properties on **<target_name>**:

INSTALL_DESTINATION

Holds the **<destination>** as described above, and is used by *install_jar_exports()*.

install_jni_symlink

Installs JNI symlinks for target generated by *add_jar()*:

```
install_jni_symlink(<target_name> <destination>)
install_jni_symlink(<target_name> DESTINATION <destination> [COMPONENT <comp
```

This command installs the **<target_name>** JNI symlinks to the given **<destination>**. It should be called in the same scope as *add_jar()* or it will fail.

Added in version 3.4: The second signature with **DESTINATION** and **COMPONENT** options.

DESTINATION

Specify the directory on disk to which a file will be installed.

COMPONENT

Specify an installation component name with which the install rule is associated, such as "runtime" or "development".

Utilize the following commands to create a JNI symbolic link:

```

set(CMAKE_JNI_TARGET TRUE)
add_jar(shibboleet shibboleet.java VERSION 1.2.0)
install_jar(shibboleet ${LIB_INSTALL_DIR}/shibboleet)
install_jni_symlink(shibboleet ${JAVA_LIB_INSTALL_DIR})

```

Header Generation

create_javah

Added in version 3.4.

Generates C header files for java classes:

```

create_javah(TARGET <target> | GENERATED_FILES <VAR>
             CLASSES <class>...
             [CLASSPATH <classpath>...]
             [DEPENDS <depend>...]
             [OUTPUT_NAME <path>|OUTPUT_DIR <path>]
             )

```

Deprecated since version 3.11: This command will no longer be supported starting with version 10 of the JDK due to the *suppression of javah tool*. The `add_jar(GENERATE_NATIVE_HEADERS)` command should be used instead.

Create C header files from java classes. These files provide the connective glue that allow your Java and C code to interact.

There are two main signatures for **create_javah()**. The first signature returns generated files through variable specified by the **GENERATED_FILES** option. For example:

```

create_javah(GENERATED_FILES files_headers
             CLASSES org.cmake.HelloWorld
             CLASSPATH hello.jar
             )

```

The second signature for **create_javah()** creates a target which encapsulates header files generation. E.g.

```

create_javah(TARGET target_headers
             CLASSES org.cmake.HelloWorld
             CLASSPATH hello.jar
             )

```

Both signatures share same options.

CLASSES

Specifies Java classes used to generate headers.

CLASSPATH

Specifies various paths to look up classes. Here **.class** files, jar files or targets created by command `add_jar` can be used.

DEPENDS

Targets on which the javah target depends.

OUTPUT_NAME

Concatenates the resulting header files for all the classes listed by option **CLASSES** into **<path>**. Same behavior as option **-o** of **javah** tool.

OUTPUT_DIR

Sets the directory where the header files will be generated. Same behavior as option **-d** of **javah** tool. If not specified, **CMAKE_CURRENT_BINARY_DIR** is used as the output directory.

Exporting JAR Targets**install_jar_exports**

Added in version 3.7.

Installs a target export file:

```
install_jar_exports(TARGETS <jars>...
                   [NAMESPACE <namespace>]
                   FILE <filename>
                   DESTINATION <destination> [COMPONENT <component>])
```

This command installs a target export file **<filename>** for the named jar targets to the given **<destination>** directory. Its function is similar to that of **install(EXPORT)**.

TARGETS

List of targets created by **add_jar()** command.

NAMESPACE

Added in version 3.9.

The **<namespace>** value will be prepend to the target names as they are written to the import file.

FILE Specify name of the export file.

DESTINATION

Specify the directory on disk to which a file will be installed.

COMPONENT

Specify an installation component name with which the install rule is associated, such as "runtime" or "development".

export_jars

Added in version 3.7.

Writes a target export file:

```
export_jars(TARGETS <jars>...
            [NAMESPACE <namespace>]
            FILE <filename>)
```

This command writes a target export file **<filename>** for the named **<jars>** targets. Its function is similar to that of **export()**.

TARGETS

List of targets created by **add_jar()** command.

NAMESPACE

Added in version 3.9.

The **<namespace>** value will be prepend to the target names as they are written to the import file.

FILE Specify name of the export file.

Finding JARs**find_jar**

Finds the specified jar file:

```
find_jar(<VAR>
  <name> | NAMES <name1> [<name2>...]
  [PATHS <path1> [<path2>... ENV <var>]]
  [VERSIONS <version1> [<version2>]]
  [DOC "cache documentation string"]
)
```

This command is used to find a full path to the named jar. A cache entry named by **<VAR>** is created to store the result of this command. If the full path to a jar is found the result is stored in the variable and the search will not repeated unless the variable is cleared. If nothing is found, the result will be **<VAR>-NOTFOUND**, and the search will be attempted again next time **find_jar()** is invoked with the same variable.

NAMES

Specify one or more possible names for the jar file.

PATHS

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

VERSIONS

Specify jar versions.

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

Creating Java Documentation**create_javadoc**

Creates java documentation based on files and packages:

```
create_javadoc(<VAR>
  (PACKAGES <pkg1> [<pkg2>...] | FILES <file1> [<file2>...])
  [SOURCEPATH <sourcepath>]
  [CLASSPATH <classpath>]
  [INSTALLPATH <install path>]
  [DOCTITLE <the documentation title>]
  [WINDOWTITLE <the title of the document>]
  [AUTHOR (TRUE|FALSE)]
  [USE (TRUE|FALSE)]
  [VERSION (TRUE|FALSE)]
)
```

The **create_javadoc()** command can be used to create java documentation. There are two main signatures for **create_javadoc()**.

The first signature works with package names on a path with source files:

```

create_javadoc(my_example_doc
    PACKAGES com.example.foo com.example.bar
    SOURCEPATH "${CMAKE_CURRENT_SOURCE_DIR}"
    CLASSPATH ${CMAKE_JAVA_INCLUDE_PATH}
    WINDOWTITLE "My example"
    DOCTITLE "<h1>My example</h1>"
    AUTHOR TRUE
    USE TRUE
    VERSION TRUE
)

```

The second signature for **create_javadoc()** works on a given list of files:

```

create_javadoc(my_example_doc
    FILES java/A.java java/B.java
    CLASSPATH ${CMAKE_JAVA_INCLUDE_PATH}
    WINDOWTITLE "My example"
    DOCTITLE "<h1>My example</h1>"
    AUTHOR TRUE
    USE TRUE
    VERSION TRUE
)

```

Both signatures share most of the options. For more details please read the javadoc manpage.

PACKAGES

Specify java packages.

FILES Specify java source files. If relative paths are specified, they are relative to *CMAKE_CURRENT_SOURCE_DIR*.

SOURCEPATH

Specify the directory where to look for packages. By default, *CMAKE_CURRENT_SOURCE_DIR* directory is used.

CLASSPATH

Specify where to find user class files. Same behavior as option **-classpath** of **javadoc** tool.

INSTALLPATH

Specify where to install the java documentation. If you specified, the documentation will be installed to **\${CMAKE_INSTALL_PREFIX}/share/javadoc/<VAR>**.

DOCTITLE

Specify the title to place near the top of the overview summary file. Same behavior as option **-doctitle** of **javadoc** tool.

WINDOWTITLE

Specify the title to be placed in the HTML **<title>** tag. Same behavior as option **-windowtitle** of **javadoc** tool.

AUTHOR

When value **TRUE** is specified, includes the **@author** text in the generated docs. Same behavior as option **-author** of **javadoc** tool.

USE

When value **TRUE** is specified, creates class and package usage pages. Includes one Use page for each documented class and package. Same behavior as option **-use** of **javadoc** tool.

VERSION

When value **TRUE** is specified, includes the version text in the generated docs. Same behavior as option **-version** of **javadoc** tool.

UseSWIG

This file provides support for **SWIG**. It is assumed that *FindSWIG* module has already been loaded.

CMake Commands

The following command is defined for use with **SWIG**:

swig_add_library

Added in version 3.8.

Define swig module with given name and specified language:

```
swig_add_library(<name>
    [TYPE <SHARED|MODULE|STATIC|USE_BUILD_SHARED_LIBS>]
    LANGUAGE <language>
    [NO_PROXY]
    [OUTPUT_DIR <directory>]
    [OUTFILE_DIR <directory>]
    SOURCES <file>...
)
```

Targets created with the **swig_add_library** command have the same capabilities as targets created with the *add_library()* command, so those targets can be used with any command expecting a target (e.g. *target_link_libraries()*).

Changed in version 3.13: This command creates a target with the specified **<name>** when policy *CMP0078* is set to **NEW**. Otherwise, the legacy behavior will choose a different target name and store it in the **SWIG_MODULE_<name>_REAL_NAME** variable.

Changed in version 3.15: Alternate library name (set with the *OUTPUT_NAME* property, for example) will be passed on to **Python** and **CSharp** wrapper libraries.

Changed in version 3.21: Generated library use standard naming conventions for **CSharp** language when policy *CMP0122* is set to **NEW**. Otherwise, the legacy behavior is applied.

NOTE:

For multi-config generators, this module does not support configuration-specific files generated by **SWIG**. All build configurations must result in the same generated source file.

NOTE:

For *Makefile Generators*, if, for some sources, the **USE_SWIG_DEPENDENCIES** property is **FALSE**, **swig_add_library** does not track file dependencies, so depending on the **<name>_swig_compilation** custom target is required for targets which require the **swig**-generated files to exist. Other generators may depend on the source files that would be generated by **SWIG**.

TYPE **SHARED**, **MODULE** and **STATIC** have the same semantic as for the *add_library()* command. If **USE_BUILD_SHARED_LIBS** is specified, the library type will be **STATIC** or **SHARED** based on whether the current value of the *BUILD_SHARED_LIBS* variable is **ON**. If no type is specified, **MODULE** will be used.

LANGUAGE

Specify the target language.

Added in version 3.1: Go and Lua language support.

Added in version 3.2: R language support.

Added in version 3.18: Fortran language support.

NO_PROXY

Added in version 3.12.

Prevent the generation of the wrapper layer (swig **-noproxy** option).

OUTPUT_DIR

Added in version 3.12.

Specify where to write the language specific files (swig **-outdir** option). If not given, the **CMAKE_SWIG_OUTDIR** variable will be used. If neither is specified, the default depends on the value of the **UseSWIG_MODULE_VERSION** variable as follows:

- If **UseSWIG_MODULE_VERSION** is 1 or is undefined, output is written to the **CMAKE_CURRENT_BINARY_DIR** directory.
- If **UseSWIG_MODULE_VERSION** is 2, a dedicated directory will be used. The path of this directory can be retrieved from the **SWIG_SUPPORT_FILES_DIRECTORY** target property.

OUTFILE_DIR

Added in version 3.12.

Specify an output directory name where the generated source file will be placed (swig **-o** option). If not specified, the **SWIG_OUTFILE_DIR** variable will be used. If neither is specified, **OUTPUT_DIR** or **CMAKE_SWIG_OUTDIR** is used instead.

SOURCES

List of sources for the library. Files with extension **.i** will be identified as sources for the **SWIG** tool. Other files will be handled in the standard way.

Added in version 3.14: This behavior can be overridden by specifying the variable **SWIG_SOURCE_FILE_EXTENSIONS**.

NOTE:

If **UseSWIG_MODULE_VERSION** is set to 2, it is **strongly** recommended to use a dedicated directory unique to the target when either the **OUTPUT_DIR** option or the **CMAKE_SWIG_OUTDIR** variable are specified. The output directory contents are erased as part of the target build, so to prevent interference between targets or losing other important files, each target should have its own dedicated output directory.

Properties on Source Files

Source file properties on module files **must** be set before the invocation of the **swig_add_library** command to specify special behavior of SWIG and ensure generated files will receive the required settings.

CPLUSPLUS

Call SWIG in c++ mode. For example:

```
set_property(SOURCE mymod.i PROPERTY CPLUSPLUS ON)
swig_add_library(mymod LANGUAGE python SOURCES mymod.i)
```

SWIG_FLAGS

Deprecated since version 3.12: Replaced with the fine-grained properties that follow.

Pass custom flags to the SWIG executable.

INCLUDE_DIRECTORIES, COMPILE_DEFINITIONS and COMPILE_OPTIONS

Added in version 3.12.

Add custom flags to SWIG compiler and have same semantic as properties *INCLUDE_DIRECTORIES*, *COMPILE_DEFINITIONS* and *COMPILE_OPTIONS*.

USE_TARGET_INCLUDE_DIRECTORIES

Added in version 3.13.

If set to **TRUE**, contents of target property *INCLUDE_DIRECTORIES* will be forwarded to **SWIG** compiler. If set to **FALSE** target property *INCLUDE_DIRECTORIES* will be ignored. If not set, target property **SWIG_USE_TARGET_INCLUDE_DIRECTORIES** will be considered.

GENERATED_INCLUDE_DIRECTORIES, GENERATED_COMPILE_DEFINITIONS and GENERATED_COMPILE_OPTIONS

Added in version 3.12.

Add custom flags to the C/C++ generated source. They will fill, respectively, properties *INCLUDE_DIRECTORIES*, *COMPILE_DEFINITIONS* and *COMPILE_OPTIONS* of generated C/C++ file.

DEPENDS

Added in version 3.12.

Specify additional dependencies to the source file.

USE_SWIG_DEPENDENCIES

Added in version 3.20.

If set to **TRUE**, implicit dependencies are generated by the **swig** tool itself. This property is only meaningful for *Makefile*, *Ninja*, *Xcode*, and *Visual Studio* generators. Default value is **FALSE**.

Added in version 3.21: Added the support of *Xcode* generator.

Added in version 3.22: Added the support of *Visual Studio Generators*.

SWIG_MODULE_NAME

Specify the actual import name of the module in the target language. This is required if it cannot be scanned automatically from source or different from the module file basename. For example:

```
set_property(SOURCE mymod.i PROPERTY SWIG_MODULE_NAME mymod_realname)
```

Changed in version 3.14: If policy *CMP0086* is set to **NEW**, **-module <module_name>** is passed to **SWIG** compiler.

OUTPUT_DIR

Added in version 3.19.

Specify where to write the language specific files (swig **-outdir** option) for the considered source file. If not specified, the other ways to define the output directory applies (see **OUTPUT_DIR** option of **swig_add_library()** command).

OUTFILE_DIR

Added in version 3.19.

Specify an output directory where the generated source file will be placed (swig **-o** option) for the considered source file. If not specified, **OUTPUT_DIR** source property will be used. If neither are specified, the other ways to define output file directory applies (see **OUTFILE_DIR** option of **swig_add_library()** command).

Properties on Targets

Target library properties can be set to apply same configuration to all SWIG input files.

SWIG_INCLUDE_DIRECTORIES, SWIG_COMPILE_DEFINITIONS and SWIG_COMPILE_OPTIONS

Added in version 3.12.

These properties will be applied to all SWIG input files and have same semantic as target properties *INCLUDE_DIRECTORIES*, *COMPILE_DEFINITIONS* and *COMPILE_OPTIONS*.

```
set (UseSWIG_TARGET_NAME_PREFERENCE STANDARD)
swig_add_library(mymod LANGUAGE python SOURCES mymod.i)
set_property(TARGET mymod PROPERTY SWIG_COMPILE_DEFINITIONS MY_DEF1 MY_DEF2)
set_property(TARGET mymod PROPERTY SWIG_COMPILE_OPTIONS -bla -blb)
```

SWIG_USE_TARGET_INCLUDE_DIRECTORIES

Added in version 3.13.

If set to **TRUE**, contents of target property *INCLUDE_DIRECTORIES* will be forwarded to **SWIG** compiler. If set to **FALSE** or not defined, target property *INCLUDE_DIRECTORIES* will be ignored. This behavior can be overridden by specifying source property **USE_TARGET_INCLUDE_DIRECTORIES**.

SWIG_GENERATED_INCLUDE_DIRECTORIES, SWIG_GENERATED_COMPILE_DEFINITIONS and SWIG_GENERATED_COMPILE_OPTIONS

Added in version 3.12.

These properties will populate, respectively, properties *INCLUDE_DIRECTORIES*, *COMPILE_DEFINITIONS* and *COMPILE_FLAGS* of all generated C/C++ files.

SWIG_DEPENDS

Added in version 3.12.

Add dependencies to all SWIG input files.

Read-only Target Properties

The following target properties are output properties and can be used to get information about support files generated by **SWIG** interface compilation.

SWIG_SUPPORT_FILES

Added in version 3.12.

This output property list of wrapper files generated during SWIG compilation.

```
set (UseSWIG_TARGET_NAME_PREFERENCE STANDARD)
swig_add_library(mymod LANGUAGE python SOURCES mymod.i)
get_property(support_files TARGET mymod PROPERTY SWIG_SUPPORT_FILES)
```

NOTE:

Only most principal support files are listed. In case some advanced features of **SWIG** are used (for example **%template**), associated support files may not be listed. Prefer to use the **SWIG_SUPPORT_FILES_DIRECTORY** property to handle support files.

SWIG_SUPPORT_FILES_DIRECTORY

Added in version 3.12.

This output property specifies the directory where support files will be generated.

NOTE:

When source property **OUTPUT_DIR** is defined, multiple directories can be specified as part of **SWIG_SUPPORT_FILES_DIRECTORY**.

CMake Variables

Some variables can be set to customize the behavior of **swig_add_library** as well as **SWIG**:

UseSWIG_MODULE_VERSION

Added in version 3.12.

Specify different behaviors for **UseSWIG** module.

- Set to 1 or undefined: Legacy behavior is applied.
- Set to 2: A new strategy is applied regarding support files: the output directory of support files is erased before **SWIG** interface compilation.

CMAKE_SWIG_FLAGS

Add flags to all swig calls.

CMAKE_SWIG_OUTDIR

Specify where to write the language specific files (swig **-outdir** option).

SWIG_OUTFILE_DIR

Added in version 3.8.

Specify an output directory name where the generated source file will be placed. If not specified, **CMAKE_SWIG_OUTDIR** is used.

SWIG_MODULE_<name>_EXTRA_DEPS

Specify extra dependencies for the generated module for **<name>**.

SWIG_SOURCE_FILE_EXTENSIONS

Added in version 3.14.

Specify a list of source file extensions to override the default behavior of considering only **.i** files as sources for the **SWIG** tool. For example:

```
set(SWIG_SOURCE_FILE_EXTENSIONS ".i" ".swg")
```

SWIG_USE_SWIG_DEPENDENCIES

Added in version 3.20.

If set to **TRUE**, implicit dependencies are generated by the **swig** tool itself. This variable is only meaningful for *Makefile*, *Ninja*, *Xcode*, and *Visual Studio* generators. Default value is **FALSE**.

Source file property **USE_SWIG_DEPENDENCIES**, if not defined, will be initialized with the value of this variable.

Added in version 3.21: Added the support of *Xcode* generator.

Added in version 3.22: Added the support of *Visual Studio Generators*.

Deprecated Commands**swig_link_libraries**

Deprecated since version 3.13: Use *target_link_libraries()* with the standard target name, or with **\${SWIG_MODULE_<name>_REAL_NAME}** for legacy target naming.

Link libraries to swig module:

```
swig_link_libraries(<name> <item>...)
```

This command has same capabilities as *target_link_libraries()* command.

NOTE:

When policy *CMP0078* is set to **NEW**, *swig_add_library()* creates a standard target with the specified **<name>** and *target_link_libraries()* must be used instead of this command.

With the legacy behavior (when *CMP0078* is set to **OLD** and the **UseSWIG_TARGET_NAME_PREFERENCE** variable is set to **"LEGACY"**, or in CMake versions prior to 3.12), it is preferable to use **target_link_libraries(\${SWIG_MODULE_<name>_REAL_NAME} ...)** instead of this command.

UsewxWidgets

Convenience include for using wxWidgets library.

Determines if wxWidgets was FOUND and sets the appropriate libs, incdirs, flags, etc. **INCLUDE_DIRECTORIES** and **LINK_DIRECTORIES** are called.

USAGE

```
# Note that for MinGW users the order of libs is important!
find_package(wxWidgets REQUIRED net gl core base)
```

```
include(${wxWidgets_USE_FILE})
# and for each of your dependent executable/library targets:
target_link_libraries(<YourTarget> ${wxWidgets_LIBRARIES})
```

DEPRECATED

LINK_LIBRARIES is not called in favor of adding dependencies per target.

AUTHOR

Jan Woetzel <jw -at- mip.informatik.uni-kiel.de>

FIND MODULES

These modules search for third-party software. They are normally called through the *find_package()* command.

FindALSA

Find Advanced Linux Sound Architecture (ALSA)

Find the alsa libraries (**asound**)

Imported Targets

Added in version 3.12.

This module defines *IMPORTED* target **ALSA::ALSA**, if ALSA has been found.

Result Variables

This module defines the following variables:

ALSA_FOUND

True if ALSA_INCLUDE_DIR & ALSA_LIBRARY are found

ALSA_LIBRARIES

List of libraries when using ALSA.

ALSA_INCLUDE_DIRS

Where to find the ALSA headers.

Cache variables

The following cache variables may also be set:

ALSA_INCLUDE_DIR

the ALSA include directory

ALSA_LIBRARY

the absolute path of the asound library

FindArmadillo

Find the Armadillo C++ library. Armadillo is a library for linear algebra & scientific computing.

Added in version 3.18: Support for linking wrapped libraries directly (**ARMA_DONT_USE_WRAPPER**).

Using Armadillo:

```
find_package(Armadillo REQUIRED)
include_directories(${ARMADILLO_INCLUDE_DIRS})
add_executable(foo foo.cc)
target_link_libraries(foo ${ARMADILLO_LIBRARIES})
```

This module sets the following variables:

```
ARMADILLO_FOUND - set to true if the library is found
ARMADILLO_INCLUDE_DIRS - list of required include directories
ARMADILLO_LIBRARIES - list of libraries to be linked
ARMADILLO_VERSION_MAJOR - major version number
ARMADILLO_VERSION_MINOR - minor version number
ARMADILLO_VERSION_PATCH - patch version number
ARMADILLO_VERSION_STRING - version number as a string (ex: "1.0.4")
ARMADILLO_VERSION_NAME - name of the version (ex: "Antipodean Antileech")
```

FindASPELL

Try to find ASPELL

Once done this will define

```
ASPELL_FOUND - system has ASPELL
ASPELL_EXECUTABLE - the ASPELL executable
ASPELL_INCLUDE_DIR - the ASPELL include directory
ASPELL_LIBRARIES - The libraries needed to use ASPELL
ASPELL_DEFINITIONS - Compiler switches required for using ASPELL
```

FindAVIFile

Locate *AVIFILE* library and include paths.

AVIFILE is a set of libraries for i386 machines to use various AVI codecs. Support is limited beyond Linux. Windows provides native AVI support, and so doesn't need this library. This module defines

```
AVIFILE_INCLUDE_DIR, where to find avifile.h , etc.
AVIFILE_LIBRARIES, the libraries to link against
AVIFILE_DEFINITIONS, definitions to use when compiling
AVIFILE_FOUND, If false, don't try to use AVIFILE
```

FindBacktrace

Find provider for *backtrace(3)*.

Checks if OS supports **backtrace(3)** via either **libc** or custom library. This module defines the following variables:

Backtrace_HEADER

The header file needed for **backtrace(3)**. Cached. Could be forcibly set by user.

Backtrace_INCLUDE_DIRS

The include directories needed to use **backtrace(3)** header.

Backtrace_LIBRARIES

The libraries (linker flags) needed to use **backtrace(3)**, if any.

Backtrace_FOUND

Is set if and only if **backtrace(3)** support detected.

The following cache variables are also available to set or use:

Backtrace_LIBRARY

The external library providing backtrace, if any.

Backtrace_INCLUDE_DIR

The directory holding the **backtrace(3)** header.

Typical usage is to generate of header file using *configure_file()* with the contents like the following:

```
#cmakedefine01 Backtrace_FOUND
#if Backtrace_FOUND
# include <${Backtrace_HEADER}>
#endif
```

And then reference that generated header file in actual source.

Imported Targets

Added in version 3.30.

This module defines the following *IMPORTED* targets:

Backtrace::Backtrace

An interface library providing usage requirements for the found components.

FindBISON

Find **bison** executable and provide a macro to generate custom build rules.

The module defines the following variables:

BISON_FOUND

True if the program was found.

BISON_EXECUTABLE

The path to the **bison** program.

BISON_VERSION

The version of **bison**.

The minimum required version of **bison** can be specified using the standard CMake syntax, e.g. *find_package(BISON 2.1.3)*.

If **bison** is found, the module defines the macro:

bison_target

```
bison_target(<Name> <YaccInput> <CodeOutput>
             [OPTIONS <options>...]
             [COMPILE_FLAGS <string>]
             [DEFINES_FILE <file>]
             [VERBOSE [<file>]]
             [REPORT_FILE <file>]
             )
```

which will create a custom rule to generate a parser. **<YaccInput>** is the path to a yacc file. **<CodeOutput>** is the name of the source file generated by bison. A header file can also be generated, and contains the token list.

Changed in version 3.14: When *CMP0088* is set to **NEW**, **bison** runs in the *CMAKE_CURRENT_BINARY_DIR* directory.

The options are:

OPTIONS <options>...

Added in version 4.0.

A *semicolon-separated* list of options added to the **bison** command line.

COMPILE_FLAGS <string>

Deprecated since version 4.0.

Space-separated bison options added to the **bison** command line. A *;-list* will not work. This option is deprecated in favor of **OPTIONS** <options>....

DEFINES_FILE <file>

Added in version 3.4.

Specify a non-default header <file> to be generated by **bison**.

VERBOSE [<file>]

Tell **bison** to write a report file of the grammar and parser.

Deprecated since version 3.7: If <file> is given, it specifies path the report file is copied to. [<file>] is left for backward compatibility of this module. Use **VERBOSE REPORT_FILE** <file>.

REPORT_FILE <file>

Added in version 3.7.

Specify a non-default report <file>, if generated.

The macro defines the following variables:

BISON_<Name>_DEFINED

True if the macro ran successfully.

BISON_<Name>_INPUT

The input source file, an alias for <YaccInput>.

BISON_<Name>_OUTPUT_SOURCE

The source file generated by **bison**.

BISON_<Name>_OUTPUT_HEADER

The header file generated by **bison**.

BISON_<Name>_OUTPUTS

All files generated by **bison** including the source, the header and the report.

BISON_<Name>_OPTIONS

Added in version 4.0.

Options used in the **bison** command line.

BISON_<Name>_COMPILE_FLAGS

Deprecated since version 4.0.

Options used in the **bison** command line. This variable is deprecated in favor of **BISON_<Name>_OPTIONS** variable.

Examples

```
find_package(BISON)
bison_target(MyParser parser.y ${CMAKE_CURRENT_BINARY_DIR}/parser.cpp
```

```

        DEFINES_FILE ${CMAKE_CURRENT_BINARY_DIR}/parser.h)
    add_executable(Foo main.cpp ${BISON_MyParser_OUTPUTS})

```

Adding additional command-line options to the **bison** executable can be passed as a list. For example, adding the **-Wall** option to report all warnings, and **--no-lines (-l)** to not generate **#line** directives.

```

    find_package(BISON)

    if(BISON_FOUND)
        bison_target(MyParser parser.y parser.cpp OPTIONS -Wall --no-lines)
    endif()

```

Generator expressions can be used in **OPTIONS <options....** For example, to add the **--debug (-t)** option only for the **Debug** build type:

```

    find_package(BISON)

    if(BISON_FOUND)
        bison_target(MyParser parser.y parser.cpp OPTIONS $<$<CONFIG:Debug>:-t>)
    endif()

```

FindBLAS

Find Basic Linear Algebra Subprograms (BLAS) library

Version modified for Gentoo Linux. If a valid PkgConfig configuration is found, this overrides and cancels all further checks.

This module finds an installed Fortran library that implements the *BLAS linear-algebra interface*.

At least one of the **C**, **CXX**, or **Fortran** languages must be enabled.

Input Variables

The following variables may be set to influence this module's behavior:

BLA_STATIC

if **ON** use static linkage

BLA_VENDOR

Set to one of the *BLAS/LAPACK Vendors* to search for BLAS only from the specified vendor. If not set, all vendors are considered.

BLA_F95

if **ON** tries to find the BLAS95 interfaces

BLA_PREFER_PKGCONFIG

Added in version 3.11.

if set **pkg-config** will be used to search for a BLAS library first and if one is found that is preferred

BLA_PKGCONFIG_BLAS

Added in version 3.25.

If set, the **pkg-config** method will look for this module name instead of just **blas**.

BLA_SIZEOF_INTEGER

Added in version 3.22.

Specify the BLAS/LAPACK library integer size:

- 4** Search for a BLAS/LAPACK with 32-bit integer interfaces.
- 8** Search for a BLAS/LAPACK with 64-bit integer interfaces.
- ANY** Search for any BLAS/LAPACK. Most likely, a BLAS/LAPACK with 32-bit integer interfaces will be found.

Imported Targets

This module defines the following *IMPORTED* targets:

BLAS::BLAS

Added in version 3.18.

The libraries to use for BLAS, if found.

Result Variables

This module defines the following variables:

BLAS_FOUND

library implementing the BLAS interface is found

BLAS_LINKER_FLAGS

uncached list of required linker flags (excluding **-l** and **-L**).

BLAS_LIBRARIES

uncached list of libraries (using full path name) to link against to use BLAS (may be empty if compiler implicitly links BLAS)

BLAS95_LIBRARIES

uncached list of libraries (using full path name) to link against to use BLAS95 interface

BLAS95_FOUND

library implementing the BLAS95 interface is found

BLAS/LAPACK Vendors**Generic**

Generic reference implementation

ACML, ACML_MP, ACML_GPU

AMD Core Math Library

AOCL, AOCL_mt

Added in version 3.27.

AMD Optimizing CPU Libraries

Apple, NAS

Apple BLAS (Accelerate), and Apple NAS (vecLib)

Arm, Arm_mp, Arm_ilp64, Arm_ilp64_mp

Added in version 3.18.

Arm Performance Libraries

ATLAS

Automatically Tuned Linear Algebra Software

CXML, DXML

Compaq/Digital Extended Math Library

EML, EML_mt

Added in version 3.20.

Elbrus Math Library

FLAME

Added in version 3.11.

BLIS Framework

FlexiBLAS

Added in version 3.19.

Fujitsu_SSL2, Fujitsu_SSL2BLAMP, Fujitsu_SSL2SVE, Fujitsu_SSL2BLAMPSVE

Added in version 3.20.

Fujitsu SSL2 serial and parallel blas/lapack with SVE instructions

Goto GotoBLAS**IBMESSL, IBMESSL_SMP**

IBM Engineering and Scientific Subroutine Library

Intel Intel MKL 32 bit and 64 bit obsolete versions**Intel10_32**

Intel MKL v10 32 bit, threaded code

Intel10_64lp

Intel MKL v10+ 64 bit, threaded code, lp64 model

Intel10_64lp_seq

Intel MKL v10+ 64 bit, sequential code, lp64 model

Intel10_64ilp

Added in version 3.13.

Intel MKL v10+ 64 bit, threaded code, ilp64 model

Intel10_64ilp_seq

Added in version 3.13.

Intel MKL v10+ 64 bit, sequential code, ilp64 model

Intel10_64_dyn

Added in version 3.17.

Intel MKL v10+ 64 bit, single dynamic library

libblastrampoline

Added in version 3.30.

A BLAS/LAPACK demuxing library using PLT trampolines

NVHPC

Added in version 3.21.

NVIDIA HPC SDK

OpenBLAS

Added in version 3.6.

PhiPACK

Portable High Performance ANSI C (PHiPAC)

SCSL, SCSL_mp

Scientific Computing Software Library

SGIMATH

SGI Scientific Mathematical Library

SunPerf

Sun Performance Library

Intel MKL

To use the Intel MKL implementation of BLAS, a project must enable at least one of the **C** or **CXX** languages. Set **BLA_VENDOR** to an Intel MKL variant either on the command-line as **-DBLA_VENDOR=Intel10_64lp** or in project code:

```
set(BLA_VENDOR Intel10_64lp)
find_package(BLAS)
```

In order to build a project using Intel MKL, and end user must first establish an Intel MKL environment:

Intel oneAPI

Source the full Intel environment script:

```
. /opt/intel/oneapi/setvars.sh
```

Or, source the MKL component environment script:

```
. /opt/intel/oneapi/mkl/latest/env/vars.sh
```

Intel Classic

Source the full Intel environment script:

```
. /opt/intel/bin/compilervars.sh intel64
```

Or, source the MKL component environment script:

```
. /opt/intel/mkl/bin/mklvars.sh intel64
```

The above environment scripts set the **MKLROOT** environment variable to the top of the MKL installation. They also add the location of the runtime libraries to the dynamic library loader environment variable for your platform (e.g. **LD_LIBRARY_PATH**). This is necessary for programs linked against MKL to run.

NOTE:

As of Intel oneAPI 2021.2, loading only the MKL component does not make all of its dependencies available. In particular, the **iompi5** library must be available separately, or provided by also loading the compiler component environment:

```
. /opt/intel/oneapi/compiler/latest/env/vars.sh
```

FindBullet

Try to find the Bullet physics engine

This module defines the following variables

BULLET_FOUND - Was bullet found
BULLET_INCLUDE_DIRS - the Bullet include directories
BULLET_LIBRARIES - Link to this, by default it includes
all bullet components (Dynamics,
Collision, LinearMath, & SoftBody)

This module accepts the following variables

BULLET_ROOT - Can be set to bullet install path or Windows build path

FindBZip2

Try to find BZip2

Imported Targets

Added in version 3.12.

This module defines *IMPORTED* target **BZip2::BZip2**, if BZip2 has been found.

Result Variables

This module defines the following variables:

BZIP2_FOUND

system has BZip2

BZIP2_INCLUDE_DIRS

Added in version 3.12: the BZip2 include directories

BZIP2_LIBRARIES

Link these to use BZip2

BZIP2_NEED_PREFIX

this is set if the functions are prefixed with **BZ2_**

BZIP2_VERSION

Added in version 3.26: the version of BZip2 found.

See also legacy variable **BZIP2_VERSION_STRING**.

Cache variables

The following cache variables may also be set:

BZIP2_INCLUDE_DIR

the directory containing the BZip2 headers

BZIP2_LIBRARY_RELEASE

the path to the BZip2 library for release configurations

BZIP2_LIBRARY_DEBUG

the path to the BZip2 library for debug configurations

Legacy Variables

The following variables are provided for backward compatibility:

BZIP2_VERSION_STRING

the version of BZip2 found.

Changed in version 3.26: Superseded by **BZIP2_VERSION**.

FindCABLE

Find CABLE

This module finds if CABLE is installed and determines where the include files and libraries are. This code sets the following variables:

```
CABLE           the path to the cable executable
CABLE_TCL_LIBRARY the path to the Tcl wrapper library
CABLE_INCLUDE_DIR the path to the include directory
```

To build Tcl wrappers, you should add shared library and link it to `${CABLE_TCL_LIBRARY}`. You should also add `${CABLE_INCLUDE_DIR}` as an include directory.

FindCoin3D

Find Coin3D (Open Inventor)

Coin3D is an implementation of the Open Inventor API. It provides data structures and algorithms for 3D visualization.

This module defines the following variables

```
COIN3D_FOUND      - system has Coin3D - Open Inventor
COIN3D_INCLUDE_DIRS - where the Inventor include directory can be found
COIN3D_LIBRARIES  - Link to this to use Coin3D
```

FindCUDAToolkit

Added in version 3.17.

This script locates the NVIDIA CUDA toolkit and the associated libraries, but does not require the **CUDA** language be enabled for a given project. This module does not search for the NVIDIA CUDA Samples.

Added in version 3.19: QNX support.

Search Behavior

The CUDA Toolkit search behavior uses the following order:

1. If the **CUDA** language has been enabled we will use the directory containing the compiler as the first search location for **nvcc**.
2. If the variable `CMAKE_CUDA_COMPILER` or the environment variable `CUDACXX` is defined, it will be used as the path to the **nvcc** executable.
3. If the **CUDAToolkit_ROOT** cmake configuration variable (e.g., `-DCUDA-Toolkit_ROOT=/some/path`) or environment variable is defined, it will be searched. If both an environment variable **and** a configuration variable are specified, the *configuration* variable takes precedence.

The directory specified here must be such that the executable **nvcc** or the appropriate **version.txt** or **version.json** file can be found underneath the specified directory.

4. If the `CUDA_PATH` environment variable is defined, it will be searched for **nvcc**.
5. The user's path is searched for **nvcc** using `find_program()`. If this is found, no subsequent search attempts are performed. Users are responsible for ensuring that the first **nvcc** to show up in the path is the desired path in the event that multiple CUDA Toolkits are installed.
6. On Unix systems, if the symbolic link `/usr/local/cuda` exists, this is used. No subsequent search attempts are performed. No default symbolic link location exists for the Windows platform.
7. The platform specific default install locations are searched. If exactly one candidate is found, this is used. The default CUDA Toolkit install locations searched are:

Platform	Search Pattern
macOS	<code>/Developer/NVIDIA/CUDA-X.Y</code>
Other Unix	<code>/usr/local/cuda-X.Y</code>
Windows	<code>C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\vX.Y</code>

Where **X.Y** would be a specific version of the CUDA Toolkit, such as `/usr/local/cuda-9.0` or `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0`

NOTE:

When multiple CUDA Toolkits are installed in the default location of a system (e.g., both `/usr/local/cuda-9.0` and `/usr/local/cuda-10.0` exist but the `/usr/local/cuda` symbolic link does **not** exist), this package is marked as **not** found.

There are too many factors involved in making an automatic decision in the presence of multiple CUDA Toolkits being installed. In this situation, users are encouraged to either (1) set **CUDA_Toolkit_ROOT** or (2) ensure that the correct **nvcc** executable shows up in **\$PATH** for `find_program()` to find.

Arguments

[<version>]

The [<version>] argument requests a version with which the package found should be compatible. See `find_package version format` for more details.

Options

REQUIRED

If specified, configuration will error if a suitable CUDA Toolkit is not found.

QUIET

If specified, the search for a suitable CUDA Toolkit will not produce any messages.

EXACT

If specified, the CUDA Toolkit is considered found only if the exact **VERSION** specified is recovered.

Imported Targets

An *imported target* named **CUDA::toolkit** is provided.

This module defines *IMPORTED* targets for each of the following libraries that are part of the CUDA-Toolkit:

- *CUDA Runtime Library*
- *CUDA Driver Library*

- *cuBLAS*
- *cuDLA*
- *cuFile*
- *cuFFT*
- *cuRAND*
- *cuSOLVER*
- *cuSPARSE*
- *cuPTI*
- *NPP*
- *nvBLAS*
- *nvGRAPH*
- *nvJPEG*
- *nvidia-ML*
- *nvPTX Compiler*
- *nvRTC*
- *nvJitLink*
- *nvFatBin*
- *nvToolsExt*
- *nvtx3*
- *OpenCL*
- *cuLIBOS*

CUDA Runtime Library

The CUDA Runtime library (*cudart*) are what most applications will typically need to link against to make any calls such as *cudaMalloc*, and *cudaFree*.

Targets Created:

- **CUDA::cudart**
- **CUDA::cudart_static**

CUDA Driver Library

The CUDA Driver library (*cuda*) are used by applications that use calls such as *cuMemAlloc*, and *cuMemFree*.

Targets Created:

- **CUDA::cuda_driver**

cuBLAS

The *cuBLAS* library.

Targets Created:

- **CUDA::cublas**
- **CUDA::cublas_static**
- **CUDA::cublasLt** starting in CUDA 10.1

- **CUDA::cublasLt_static** starting in CUDA 10.1

cuDLA

Added in version 3.27.

The NVIDIA Tegra Deep Learning Accelerator *cuDLA* library.

Targets Created:

- **CUDA::cudla** starting in CUDA 11.6

cuFile

Added in version 3.25.

The NVIDIA GPUDirect Storage *cuFile* library.

Targets Created:

- **CUDA::cuFile** starting in CUDA 11.4
- **CUDA::cuFile_static** starting in CUDA 11.4
- **CUDA::cuFile_rdma** starting in CUDA 11.4
- **CUDA::cuFile_rdma_static** starting in CUDA 11.4

cuFFT

The *cuFFT* library.

Targets Created:

- **CUDA::cufft**
- **CUDA::cufftw**
- **CUDA::cufft_static**
- **CUDA::cufft_static_nocallback** starting in CUDA 9.2, requires CMake 3.23+
- **CUDA::cufftw_static**

cuRAND

The *cuRAND* library.

Targets Created:

- **CUDA::curand**
- **CUDA::curand_static**

cuSOLVER

The *cuSOLVER* library.

Targets Created:

- **CUDA::cusolver**
- **CUDA::cusolver_static**

cuSPARSE

The *cuSPARSE* library.

Targets Created:

- **CUDA::cusparse**
- **CUDA::cusparse_static**

cupti

The *NVIDIA CUDA Profiling Tools Interface*.

Targets Created:

- **CUDA::cupti**
- **CUDA::cupti_static**

Added in version 3.27:

- **CUDA::nvperf_host** starting in CUDA 10.2
- **CUDA::nvperf_host_static** starting in CUDA 10.2
- **CUDA::nvperf_target** starting in CUDA 10.2
- **CUDA::pcsamplutil** starting in CUDA 11.3

NPP

The *NPP* libraries.

Targets Created:

- *nppc*:
 - **CUDA::nppc**
 - **CUDA::nppc_static**
- *nppial*: Arithmetic and logical operation functions in *nppi_arithmetic_and_logical_operations.h*
 - **CUDA::nppial**
 - **CUDA::nppial_static**
- *nppicc*: Color conversion and sampling functions in *nppi_color_conversion.h*
 - **CUDA::nppicc**
 - **CUDA::nppicc_static**
- *nppicom*: JPEG compression and decompression functions in *nppi_compression_functions.h* Removed starting in CUDA 11.0, use *nvJPEG* instead.
 - **CUDA::nppicom**
 - **CUDA::nppicom_static**
- *nppidei*: Data exchange and initialization functions in *nppi_data_exchange_and_initialization.h*
 - **CUDA::nppidei**
 - **CUDA::nppidei_static**
- *nppif*: Filtering and computer vision functions in *nppi_filter_functions.h*
 - **CUDA::nppif**
 - **CUDA::nppif_static**
- *nppig*: Geometry transformation functions found in *nppi_geometry_transforms.h*
 - **CUDA::nppig**

- **CUDA::nppig_static**
- *nppim*: Morphological operation functions found in *nppi_morphological_operations.h*
 - **CUDA::nppim**
 - **CUDA::nppim_static**
- *nppist*: Statistics and linear transform in *nppi_statistics_functions.h* and *nppi_linear_transforms.h*
 - **CUDA::nppist**
 - **CUDA::nppist_static**
- *nppisu*: Memory support functions in *nppi_support_functions.h*
 - **CUDA::nppisu**
 - **CUDA::nppisu_static**
- *nppitc*: Threshold and compare operation functions in *nppi_threshold_and_compare_operations.h*
 - **CUDA::nppitc**
 - **CUDA::nppitc_static**
- *npps*:
 - **CUDA::npps**
 - **CUDA::npps_static**

nvBLAS

The *nvBLAS* libraries. This is a shared library only.

Targets Created:

- **CUDA::nvblas**

nvGRAPH

The *nvGRAPH* library. Removed starting in CUDA 11.0

Targets Created:

- **CUDA::nvgraph**
- **CUDA::nvgraph_static**

nvJPEG

The *nvJPEG* library. Introduced in CUDA 10.

Targets Created:

- **CUDA::nvjpeg**
- **CUDA::nvjpeg_static**

nvPTX Compiler

Added in version 3.25.

The *nvPTX* (PTX Compilation) library. The PTX Compiler APIs are a set of APIs which can be used to compile a PTX program into GPU assembly code. Introduced in CUDA 11.1 This is a static library only.

Targets Created:

- **CUDA::nvptxcompiler_static** starting in CUDA 11.1

nvRTC

The *nvRTC* (Runtime Compilation) library.

Targets Created:

- **CUDA::nvrtc**

Added in version 3.26:

- **CUDA::nvrtc_builtins**
- **CUDA::nvrtc_static** starting in CUDA 11.5
- **CUDA::nvrtc_builtins_static** starting in CUDA 11.5

nvJitLink

The *nvJitLink* (Runtime LTO Linking) library.

Targets Created:

- **CUDA::nvJitLink** starting in CUDA 12.0
- **CUDA::nvJitLink_static** starting in CUDA 12.0

nvFatBin

Added in version 3.30.

The *nvFatBin* (Runtime fatbin creation) library.

Targets Created:

- **CUDA::nvfatbin** starting in CUDA 12.4
- **CUDA::nvfatbin_static** starting in CUDA 12.4

nvidia-ML

The *NVIDIA Management Library*.

Targets Created:

- **CUDA::nvml**
- **CUDA::nvml_static** starting in CUDA 12.4

Added in version 3.31: Added **CUDA::nvml_static**.

nvToolsExt

Deprecated since version 3.25: With CUDA 10.0+, use *nvtx3*.

The *NVIDIA Tools Extension*. This is a shared library only.

Targets Created:

- **CUDA::nvToolsExt**

nvtx3

Added in version 3.25.

The header-only *NVIDIA Tools Extension Library*. Introduced in CUDA 10.0.

Targets created:

- **CUDA::nvtx3**

OpenCL

The *NVIDIA OpenCL Library*. This is a shared library only.

Targets Created:

- **CUDA::OpenCL**

cuLIBOS

The cuLIBOS library is a backend thread abstraction layer library which is static only. The **CUDA::cublas_static**, **CUDA::cusparse_static**, **CUDA::cufft_static**, **CUDA::curand_static**, and (when implemented) NPP libraries all automatically have this dependency linked.

Target Created:

- **CUDA::culibos**

Note: direct usage of this target by consumers should not be necessary.

Result variables

CUDAToolkit_FOUND

A boolean specifying whether or not the CUDA Toolkit was found.

CUDAToolkit_VERSION

The exact version of the CUDA Toolkit found (as reported by **nvcc --version**, **version.txt**, or **version.json**).

CUDAToolkit_VERSION_MAJOR

The major version of the CUDA Toolkit.

CUDAToolkit_VERSION_MINOR

The minor version of the CUDA Toolkit.

CUDAToolkit_VERSION_PATCH

The patch version of the CUDA Toolkit.

CUDAToolkit_BIN_DIR

The path to the CUDA Toolkit library directory that contains the CUDA executable **nvcc**.

CUDAToolkit_INCLUDE_DIRS

List of paths to all the CUDA Toolkit folders containing header files required to compile a project linking against CUDA.

CUDAToolkit_LIBRARY_DIR

The path to the CUDA Toolkit library directory that contains the CUDA Runtime library **cuda**.

CUDAToolkit_LIBRARY_ROOT

Added in version 3.18.

The path to the CUDA Toolkit directory containing the **nvvm** directory and either **version.txt** or **version.json**.

CUDAToolkit_TARGET_DIR

The path to the CUDA Toolkit directory including the target architecture when cross-compiling. When not cross-compiling this will be equivalent to the parent directory of **CUDAToolkit_BIN_DIR**.

CUDAToolkit_NVCC_EXECUTABLE

The path to the NVIDIA CUDA compiler **nvcc**. Note that this path may **not** be the same as **CMAKE_CUDA_COMPILER**. **nvcc** must be found to determine the CUDA Toolkit version as

well as determining other features of the Toolkit. This variable is set for the convenience of modules that depend on this one.

FindCups

Find the Common UNIX Printing System (CUPS).

Set **CUPS_REQUIRE_IPP_DELETE_ATTRIBUTE** to **TRUE** if you need a version which features this function (i.e. at least **1.1.19**)

Imported Targets

Added in version 3.15.

This module defines *IMPORTED* target **Cups::Cups**, if Cups has been found.

Result variables

This module will set the following variables in your project:

CUPS_FOUND

true if CUPS headers and libraries were found

CUPS_INCLUDE_DIRS

the directory containing the Cups headers

CUPS_LIBRARIES

the libraries to link against to use CUPS.

CUPS_VERSION_STRING

the version of CUPS found (since CMake 2.8.8)

Cache variables

The following cache variables may also be set:

CUPS_INCLUDE_DIR

the directory containing the Cups headers

FindCURL

Find the native CURL headers and libraries.

Added in version 3.14: This module accept optional COMPONENTS to check supported features and protocols:

```

PROTOCOLS:  ICT FILE FTP FTPS GOPHER HTTP HTTPS IMAP IMAPS LDAP LDAPS POP3
             POP3S RTMP RTSP SCP SFTP SMB SMBS SMTP SMTPS TELNET TFTP
FEATURES:   SSL IPv6 UnixSockets libz AsynchDNS IDN GSS-API PSL SPNEGO
             Kerberos NTLM NTLM_WB TLS-SRP HTTP2 HTTPS-proxy

```

Imported Targets

Added in version 3.12.

This module defines *IMPORTED* target **CURL::libcurl**, if curl has been found.

Result Variables

This module defines the following variables:

CURL_FOUND

"True" if **curl** found.

CURL_INCLUDE_DIRS

where to find **curl/curl.h**, etc.

CURL_LIBRARIES

List of libraries when using **curl**.

CURL_VERSION

Added in version 4.0.

The version of **curl** found. This supersedes **CURL_VERSION_STRING**.

CURL_VERSION_STRING

The version of **curl** found. Superseded by **CURL_VERSION**.

Added in version 3.13: Debug and Release variants are found separately.

CURL CMake

Added in version 3.17.

If CURL was built using the CMake buildsystem then it provides its own **CURLConfig.cmake** file for use with the *find_package()* command's config mode. This module looks for this file and, if found, returns its results with no further action.

Set **CURL_NO_CURL_CMAKE** to **ON** to disable this search.

Hints**CURL_USE_STATIC_LIBS**

Added in version 3.28.

Set to **TRUE** to use static libraries.

This is meaningful only when CURL is not found via its CMake Package Configuration file.

FindCurses

Find the curses or ncurses include file and library.

Result Variables

This module defines the following variables:

CURSES_FOUND

True if Curses is found.

CURSES_INCLUDE_DIRS

The include directories needed to use Curses.

CURSES_LIBRARIES

The libraries needed to use Curses.

CURSES_CFLAGS

Added in version 3.16.

Parameters which ought be given to C/C++ compilers when using Curses.

CURSES_HAVE_CURSES_H

True if curses.h is available.

CURSES_HAVE_NCURSES_H

True if ncurses.h is available.

CURSES_HAVE_NCURSES_NCURSES_H

True if `ncurses/ncurses.h` is available.

CURSES_HAVE_NCURSES_CURSES_H

True if `ncurses/curses.h` is available.

Set **CURSES_NEED_NCURSES** to **TRUE** before the **find_package(Curses)** call if NCurses functionality is required.

Added in version 3.10: Set **CURSES_NEED_WIDE** to **TRUE** before the **find_package(Curses)** call if unicode functionality is required.

Backward Compatibility

The following variable are provided for backward compatibility:

CURSES_INCLUDE_DIR

Path to Curses include. Use **CURSES_INCLUDE_DIRS** instead.

CURSES_LIBRARY

Path to Curses library. Use **CURSES_LIBRARIES** instead.

FindCVS

Find the Concurrent Versions System (CVS).

The module defines the following variables:

CVS_EXECUTABLE - path to cvs command line client
CVS_FOUND - true if the command line client was found

Example usage:

```
find_package(CVS)
if(CVS_FOUND)
  message("CVS found: ${CVS_EXECUTABLE}")
endif()
```

FindCxxTest

Find CxxTest unit testing framework.

Find the *CxxTest* suite and declare a helper macro for creating unit tests and integrating them with CTest.

Input Variables**CXXTEST_USE_PYTHON**

Deprecated since version 1.3.

Only used in the case both Python & Perl are detected on the system to control which CxxTest code generator is used. Valid only for CxxTest version 3.

In older versions of this Find Module, this variable controlled if the Python test generator was used instead of the Perl one, regardless of which scripting language the user had installed.

CXXTEST_TESTGEN_ARGS

Added in version 2.8.3.

Specify a list of options to pass to the CxxTest code generator. If not defined, **--err or--printer** is passed.

Result Variables**CXXTEST_FOUND**

True if the CxxTest framework was found

CXXTEST_INCLUDE_DIRS

Where to find the CxxTest include directory

CXXTEST_PERL_TESTGEN_EXECUTABLE

The perl-based test generator

CXXTEST_PYTHON_TESTGEN_EXECUTABLE

The python-based test generator

CXXTEST_TESTGEN_EXECUTABLE

Added in version 2.8.3.

The test generator that is actually used (chosen using user preferences and interpreters found in the system)

CXXTEST_TESTGEN_INTERPRETER

Added in version 2.8.3.

The full path to the Perl or Python executable on the system, on platforms where the script cannot be executed using its shebang line.

Module Commands**cxctest_add_test**

Create a CxxTest runner and adds it to the CTest testing suite:

```
CXXTEST_ADD_TEST(<test_name> <gen_source_file>
                 <input_files_to_testgen>...)
```

Parameters:

test_name

The name of the test

gen_source_file

The generated source filename to be generated by CxxTest

input_files_to_testgen

The list of header files containing the CxxTest::TestSuite's to be included in this runner

Example Usage

The following example, if CxxTest is found, will:

- Invoke the testgen executable to autogenerate foo_test.cc in the binary tree from "foo_test.h" in the current source directory.
- Create an executable and test called unittest_foo.

```
find_package(CxxTest)
if(CXXTEST_FOUND)
  include_directories(${CXXTEST_INCLUDE_DIR})
  enable_testing()
  CXXTEST_ADD_TEST(unittest_foo foo_test.cc
                  ${CMAKE_CURRENT_SOURCE_DIR}/foo_test.h)
  target_link_libraries(unittest_foo foo) # as needed
endif()
```

foo_test.h contains:

```
#include <cxxtest/TestSuite.h>
class MyTestSuite : public CxxTest::TestSuite
{
public:
    void testAddition( void )
    {
        TS_ASSERT( 1 + 1 > 1 );
        TS_ASSERT_EQUALS( 1 + 1, 2 );
    }
};
```

FindCygwin

Find Cygwin, a POSIX-compatible environment that runs natively on Microsoft Windows

FindDCMTK

Find DICOM ToolKit (DCMTK) libraries and applications

The module defines the following variables:

```
DCMTK_INCLUDE_DIRS - Directories to include to use DCMTK
DCMTK_LIBRARIES    - Files to link against to use DCMTK
DCMTK_FOUND        - If false, don't try to use DCMTK
DCMTK_DIR          - (optional) Source directory for DCMTK
```

Compatibility

This module is able to find a version of DCMTK that does or does not export a **DCMTKConfig.cmake** file. It applies a two step process:

- Step 1: Attempt to find DCMTK version providing a **DCMTKConfig.cmake** file.
- Step 2: If step 1 failed, rely on **FindDCMTK.cmake** to set **DCMTK_*** variables details below.

Recent DCMTK provides a **DCMTKConfig.cmake** package configuration file. To exclusively use the package configuration file (recommended when possible), pass the *NO_MODULE* option to *find_package()*. For example, *find_package(DCMTK NO_MODULE)*. This requires official DCMTK snapshot 3.6.1_20140617 or newer.

Until all clients update to the more recent DCMTK, build systems will need to support different versions of DCMTK.

On any given system, the following combinations of DCMTK versions could be considered:

	SYSTEM DCMTK	LOCAL DCMTK	Supported ?
Case A	NA	[] DCMTKConfig	YES
Case B	NA	[X] DCMTKConfig	YES
Case C	[] DCMTKConfig	NA	YES
Case D	[X] DCMTKConfig	NA	YES
Case E	[] DCMTKConfig	[] DCMTKConfig	YES (*)
Case F	[X] DCMTKConfig	[] DCMTKConfig	NO
Case G	[] DCMTKConfig	[X] DCMTKConfig	YES
Case H	[X] DCMTKConfig	[X] DCMTKConfig	YES

(*) See Troubleshooting section.

Legend:

NA: Means that no System or Local DCMTK is available

[] DCMTKConfig ..: Means that the version of DCMTK does NOT export a DCMTKConfig.cmake file.

[X] DCMTKConfig ..: Means that the version of DCMTK exports a DCMTKConfig.cmake file.

Troubleshooting

What to do if my project finds a different version of DCMTK?

Remove DCMTK entry from the CMake cache per *find_package()* documentation.

FindDevIL

This module locates the Developer's Image Library, *DevIL*.

Imported Targets

Added in version 3.21.

This module defines the *IMPORTED* targets:

DevIL::IL

Defined if the system has DevIL.

DevIL::ILU

Defined if the system has DevIL Utilities.

DevIL::ILUT

Defined if the system has DevIL Utility Toolkit.

Result Variables

This module sets:

IL_LIBRARIES

The name of the IL library. These include the full path to the core DevIL library. This one has to be linked into the application.

ILU_LIBRARIES

The name of the ILU library. Again, the full path. This library is for filters and effects, not actual loading. It doesn't have to be linked if the functionality it provides is not used.

ILUT_LIBRARIES

The name of the ILUT library. Full path. This part of the library interfaces with OpenGL. It is not strictly needed in applications.

IL_INCLUDE_DIR

where to find the il.h, ilu.h and ilut.h files.

DevIL_FOUND

This is set to TRUE if all the above variables were set. This will be set to false if ILU or ILUT are not found, even if they are not needed. In most systems, if one library is found all the others are as well. That's the way the DevIL developers release it.

DevIL_ILUT_FOUND

Added in version 3.21.

This is set to TRUE if the ILUT library is found.

FindDoxygen

Doxygen is a documentation generation tool (see <https://www.doxygen.nl>). This module looks for Doxygen and some optional tools it supports:

dot *Graphviz dot* utility used to render various graphs.

mscgen

Message Chart Generator utility used by Doxygen's `\msc` and `\mscfile` commands.

dia *Dia* the diagram editor used by Doxygen's `\diafile` command.

Added in version 3.9: These tools are available as components in the `find_package()` command. For example:

```
# Require dot, treat the other components as optional
find_package(Doxygen
              REQUIRED dot
              OPTIONAL_COMPONENTS mscgen dia)
```

The following variables are defined by this module:

DOXYGEN_FOUND

True if the **doxygen** executable was found.

DOXYGEN_VERSION

The version reported by **doxygen --version**.

Imported Targets

Added in version 3.9.

The module defines **IMPORTED** targets for Doxygen and each component found. These can be used as part of custom commands, etc. and should be preferred over old-style (and now deprecated) variables like **DOXYGEN_EXECUTABLE**. The following import targets are defined if their corresponding executable could be found (the component import targets will only be defined if that component was requested):

- **Doxygen::doxygen**
- **Doxygen::dot**
- **Doxygen::mscgen**
- **Doxygen::dia**

Functions

doxygen_add_docs

Added in version 3.9.

This function is intended as a convenience for adding a target for generating documentation with Doxygen. It aims to provide sensible defaults so that projects can generally just provide the input files and directories and that will be sufficient to give sensible results. The function supports the ability to customize the Doxygen configuration used to build the documentation.

```
doxygen_add_docs(targetName
                 [filesOrDirs...]
                 [ALL]
                 [USE_STAMP_FILE]
                 [WORKING_DIRECTORY dir]
                 [COMMENT comment]
                 [CONFIG_FILE filename])
```

The function constructs a **Doxyfile** and defines a custom target that runs Doxygen on that

generated file. The listed files and directories are used as the **INPUT** of the generated **Doxyfile** and they can contain wildcards. Any files that are listed explicitly will also be added as **SOURCES** of the custom target so they will show up in an IDE project's source list.

So that relative input paths work as expected, by default the working directory of the Doxygen command will be the current source directory (i.e. `CMAKE_CURRENT_SOURCE_DIR`). This can be overridden with the **WORKING_DIRECTORY** option to change the directory used as the relative base point. Note also that Doxygen's default behavior is to strip the working directory from relative paths in the generated documentation (see the **STRIP_FROM_PATH** Doxygen config option for details).

If provided, the optional **comment** will be passed as the **COMMENT** for the `add_custom_target()` command used to create the custom target internally.

Added in version 3.27: If **CONFIG_FILE** is set, the given file provided with full-path will be used as doxygen configuration file

Added in version 3.12: If **ALL** is set, the target will be added to the default build target.

Added in version 3.16: If **USE_STAMP_FILE** is set, the custom command defined by this function will create a stamp file with the name `<targetName>.stamp` in the current binary directory whenever doxygen is re-run. With this option present, all items in `<filesOrDirs>` must be files (i.e. no directories, symlinks or wildcards) and each of the files must exist at the time `doxygen_add_docs()` is called. An error will be raised if any of the items listed is missing or is not a file when **USE_STAMP_FILE** is given. A dependency will be created on each of the files so that doxygen will only be re-run if one of the files is updated. Without the **USE_STAMP_FILE** option, doxygen will always be re-run if the `<targetName>` target is built regardless of whether anything listed in `<filesOrDirs>` has changed.

The contents of the generated **Doxyfile** can be customized by setting CMake variables before calling `doxygen_add_docs()`. Any variable with a name of the form **DOXYGEN_<tag>** will have its value substituted for the corresponding `<tag>` configuration option in the **Doxyfile**. See the - Doxygen documentation for the full list of supported configuration options.

Some of Doxygen's defaults are overridden to provide more appropriate behavior for a CMake project. Each of the following will be explicitly set unless the variable already has a value before `doxygen_add_docs()` is called (with some exceptions noted):

DOXYGEN_HAVE_DOT

Set to **YES** if the **dot** component was requested and it was found, **NO** otherwise. Any existing value of **DOXYGEN_HAVE_DOT** is ignored.

DOXYGEN_DOT_MULTI_TARGETS

Set to **YES** by this module (note that this requires a **dot** version newer than 1.8.10). This option is only meaningful if **DOXYGEN_HAVE_DOT** is also set to **YES**.

DOXYGEN_GENERATE_LATEX

Set to **NO** by this module.

DOXYGEN_WARN_FORMAT

For Visual Studio based generators, this is set to the form recognized by the Visual Studio IDE: **\$file(\$line) : \$text**. For all other generators, Doxygen's default value is not overridden.

DOXYGEN_PROJECT_NAME

Populated with the name of the current project (i.e. *PROJECT_NAME*).

DOXYGEN_PROJECT_NUMBER

Populated with the version of the current project (i.e. *PROJECT_VERSION*).

DOXYGEN_PROJECT_BRIEF

Populated with the description of the current project (i.e. *PROJECT_DESCRIPTION*).

DOXYGEN_INPUT

Projects should not set this variable. It will be populated with the set of files and directories passed to **doxygen_add_docs()**, thereby providing consistent behavior with the other built-in commands like *add_executable()*, *add_library()* and *add_custom_target()*. If a variable named **DOXYGEN_INPUT** is set by the project, it will be ignored and a warning will be issued.

DOXYGEN_RECURSIVE

Set to **YES** by this module.

DOXYGEN_EXCLUDE_PATTERNS

If the set of inputs includes directories, this variable will specify patterns used to exclude files from them. The following patterns are added by **doxygen_add_docs()** to ensure CMake-specific files and directories are not included in the input. If the project sets **DOXYGEN_EXCLUDE_PATTERNS**, those contents are merged with these additional patterns rather than replacing them:

```
*/.git/*
*/.svn/*
*/.hg/*
*/CMakeFiles/*
*/_CPack_Packages/*
DartConfiguration.tcl
CMakeLists.txt
CMakeCache.txt
```

DOXYGEN_OUTPUT_DIRECTORY

Set to *CMAKE_CURRENT_BINARY_DIR* by this module. Note that if the project provides its own value for this and it is a relative path, it will be converted to an absolute path relative to the current binary directory. This is necessary because doxygen will normally be run from a directory within the source tree so that relative source paths work as expected. If this directory does not exist, it will be recursively created prior to executing the doxygen commands.

To change any of these defaults or override any other Doxygen config option, set relevant variables before calling **doxygen_add_docs()**. For example:

```
set(DOXYGEN_GENERATE_HTML NO)
set(DOXYGEN_GENERATE_MAN YES)

doxygen_add_docs(
  doxygen
  ${PROJECT_SOURCE_DIR}
  COMMENT "Generate man pages"
)
```

A number of Doxygen config options accept lists of values, but Doxygen requires them to be separated by whitespace. CMake variables hold lists as a string with items separated by semi-colons, so a conversion needs to be performed. The **doxygen_add_docs()** command specifically checks the following Doxygen

config options and will convert their associated CMake variable's contents into the required form if set. CMake variables are named **DOXYGEN_<name>** for the Doxygen settings specified here.

- **ABBREVIATE_BRIEF**
- **ALIASES**
- **CITE_BIB_FILES**
- **DIAFILE_DIRS**
- **DOTFILE_DIRS**
- **DOT_FONTPATH**
- **ENABLED_SECTIONS**
- **EXAMPLE_PATH**
- **EXAMPLE_PATTERNS**
- **EXCLUDE**
- **EXCLUDE_PATTERNS**
- **EXCLUDE_SYMBOLS**
- **EXPAND_AS_DEFINED**
- **EXTENSION_MAPPING**
- **EXTRA_PACKAGES**
- **EXTRA_SEARCH_MAPPINGS**
- **FILE_PATTERNS**
- **FILTER_PATTERNS**
- **FILTER_SOURCE_PATTERNS**
- **HTML_EXTRA_FILES**
- **HTML_EXTRA_STYLESHEET**
- **IGNORE_PREFIX**
- **IMAGE_PATH**
- **INCLUDE_FILE_PATTERNS**
- **INCLUDE_PATH**
- **INPUT**
- **LATEX_EXTRA_FILES**
- **LATEX_EXTRA_STYLESHEET**
- **MATHJAX_EXTENSIONS**
- **MSCFILE_DIRS**
- **PLANTUML_INCLUDE_PATH**
- **PREDEFINED**
- **QHP_CUST_FILTER_ATTRS**
- **QHP_SECT_FILTER_ATTRS**
- **STRIP_FROM_INC_PATH**
- **STRIP_FROM_PATH**

- **TAGFILES**
- **TCL_SUBST**

The following single value Doxygen options will be quoted automatically if they contain at least one space:

- **CHM_FILE**
- **DIA_PATH**
- **DOCBOOK_OUTPUT**
- **DOCSET_FEEDNAME**
- **DOCSET_PUBLISHER_NAME**
- **DOT_FONTNAME**
- **DOT_PATH**
- **EXTERNAL_SEARCH_ID**
- **FILE_VERSION_FILTER**
- **GENERATE_TAGFILE**
- **HHC_LOCATION**
- **HTML_FOOTER**
- **HTML_HEADER**
- **HTML_OUTPUT**
- **HTML_STYLESHEET**
- **INPUT_FILTER**
- **LATEX_FOOTER**
- **LATEX_HEADER**
- **LATEX_OUTPUT**
- **LAYOUT_FILE**
- **MAN_OUTPUT**
- **MAN_SUBDIR**
- **MATHJAX_CODEFILE**
- **MSCGEN_PATH**
- **OUTPUT_DIRECTORY**
- **PERL_PATH**
- **PLANTUML_JAR_PATH**
- **PROJECT_BRIEF**
- **PROJECT_LOGO**
- **PROJECT_NAME**
- **QCH_FILE**
- **QHG_LOCATION**
- **QHP_CUST_FILTER_NAME**
- **QHP_VIRTUAL_FOLDER**

- **RTF_EXTENSIONS_FILE**
- **RTF_OUTPUT**
- **RTF_STYLESHEET_FILE**
- **SEARCHDATA_FILE**
- **USE_MDFILE_AS_MAINPAGE**
- **WARN_FORMAT**
- **WARN_LOGFILE**
- **XML_OUTPUT**

Added in version 3.11: There are situations where it may be undesirable for a particular config option to be automatically quoted by **doxygen_add_docs()**, such as **ALIASES** which may need to include its own embedded quoting. The **DOXYGEN_VERBATIM_VARS** variable can be used to specify a list of Doxygen variables (including the leading **DOXYGEN_** prefix) which should not be quoted. The project is then responsible for ensuring that those variables' values make sense when placed directly in the Doxygen input file. In the case of list variables, list items are still separated by spaces, it is only the automatic quoting that is skipped. For example, the following allows **doxygen_add_docs()** to apply quoting to **DOXYGEN_PROJECT_BRIEF**, but not each item in the **DOXYGEN_ALIASES** list (*bracket syntax* can also be used to make working with embedded quotes easier):

```
set(DOXYGEN_PROJECT_BRIEF "String with spaces")
set(DOXYGEN_ALIASES
    [[somealias="@some_command param"]]
    "anotherAlias=@foobar"
)
set(DOXYGEN_VERBATIM_VARS DOXYGEN_ALIASES)
```

The resultant **Doxyfile** will contain the following lines:

```
PROJECT_BRIEF = "String with spaces"
ALIASES       = somealias="@some_command param" anotherAlias=@foobar
```

Deprecated Result Variables

Deprecated since version 3.9.

For compatibility with previous versions of CMake, the following variables are also defined but they are deprecated and should no longer be used:

DOXYGEN_EXECUTABLE

The path to the **doxygen** command. If projects need to refer to the **doxygen** executable directly, they should use the **Doxygen::doxygen** import target instead.

DOXYGEN_DOT_FOUND

True if the **dot** executable was found.

DOXYGEN_DOT_EXECUTABLE

The path to the **dot** command. If projects need to refer to the **dot** executable directly, they should use the **Doxygen::dot** import target instead.

DOXYGEN_DOT_PATH

The path to the directory containing the **dot** executable as reported in **DOXYGEN_DOT_EXECUTABLE**. The path may have forward slashes even on Windows and is not suitable for direct substitution into a **Doxyfile.in** template. If you need this value, get the **IMPORTED_LOCATION** property of the **Doxygen::dot** target and use *get_filename_component()* to extract the directory

part of that path. You may also want to consider using *file(TO_NATIVE_PATH)* to prepare the path for a Doxygen configuration file.

Deprecated Hint Variables

Deprecated since version 3.9.

DOXYGEN_SKIP_DOT

This variable has no effect for the component form of **find_package**. In backward compatibility mode (i.e. without components list) it prevents the finder module from searching for Graphviz's **dot** utility.

FindEnvModules

Added in version 3.15.

Locate an environment module implementation and make commands available to CMake scripts to use them. This is compatible with both Lua-based Lmod and TCL-based EnvironmentModules.

This module is intended for the use case of setting up the compiler and library environment within a *CTest Script* (**ctest -S**). It can also be used in a *CMake Script* (**cmake -P**).

NOTE:

The loaded environment will not survive past the end of the calling process. Do not use this module in project code (**CMakeLists.txt** files) to load a compiler environment; it will not be available during the build. Instead load the environment manually before running CMake or using the generated build system.

Example Usage

```
set(CTEST_BUILD_NAME "CrayLinux-CrayPE-Cray-dynamic")
set(CTEST_BUILD_CONFIGURATION Release)
set(CTEST_BUILD_FLAGS "-k -j8")
set(CTEST_CMAKE_GENERATOR "Unix Makefiles")

...

find_package(EnvModules REQUIRED)

env_module(purge)
env_module(load modules)
env_module(load craype)
env_module(load PrgEnv-cray)
env_module(load craype-knl)
env_module(load cray-mpich)
env_module(load cray-libsci)

set(ENV{CRAYPE_LINK_TYPE} dynamic)

...
```

Result Variables

This module will set the following variables in your project:

EnvModules_FOUND

True if a compatible environment modules framework was found.

Cache Variables

The following cache variable will be set:

EnvModules_COMMAND

The low level module command to use. Currently supported implementations are the Lua based Lmod and TCL based EnvironmentModules.

Environment Variables**ENV{MODULESHOME}**

Usually set by the module environment implementation, used as a hint to locate the module command to execute.

Provided Functions

This defines the following CMake functions for interacting with environment modules:

env_module

Execute an arbitrary module command:

```
env_module(cmd arg1 ... argN)
env_module(
  COMMAND cmd arg1 ... argN
  [OUTPUT_VARIABLE <out-var>]
  [RESULT_VARIABLE <ret-var>]
)
```

The options are:

cmd arg1 ... argN

The module sub-command and arguments to execute as if they were passed directly to the module command in your shell environment.

OUTPUT_VARIABLE <out-var>

The standard output from executing the module command.

RESULT_VARIABLE <ret-var>

The return code from executing the module command.

env_module_swap

Swap one module for another:

```
env_module_swap(out_mod in_mod
  [OUTPUT_VARIABLE <out-var>]
  [RESULT_VARIABLE <ret-var>]
)
```

This is functionally equivalent to the **module swap out_mod in_mod** shell command. The options are:

OUTPUT_VARIABLE <out-var>

The standard output from executing the module command.

RESULT_VARIABLE <ret-var>

The return code from executing the module command.

env_module_list

Retrieve the list of currently loaded modules:

```
env_module_list(<out-var>)
```

This is functionally equivalent to the **module list** shell command. The result is stored in **<out-var>** as a properly formatted CMake *semicolon-separated list* variable.

env_module_avail

Retrieve the list of available modules:

```
env_module_avail([<mod-prefix>] <out-var>)
```

This is functionally equivalent to the **module avail** **<mod-prefix>** shell command. The result is stored in **<out-var>** as a properly formatted CMake *semicolon-separated list* variable.

FindEXPAT

Find the native Expat headers and library. Expat is a stream-oriented XML parser library written in C.

Imported Targets

Added in version 3.10.

This module defines the following *IMPORTED* targets:

EXPAT::EXPAT

The Expat **expat** library, if found.

Result Variables

This module will set the following variables in your project:

EXPAT_INCLUDE_DIRS

where to find expat.h, etc.

EXPAT_LIBRARIES

the libraries to link against to use Expat.

EXPAT_FOUND

true if the Expat headers and libraries were found.

Hints**EXPAT_USE_STATIC_LIBS**

Added in version 3.28.

Set to **TRUE** to use static libraries.

Added in version 3.31: Implemented on non-Windows platforms.

FindFLEX

Find Fast Lexical Analyzer (Flex) executable and provide a macro to generate custom build rules.

The module defines the following variables:

FLEX_FOUND

True if **flex** executable is found.

FLEX_EXECUTABLE

The path to the **flex** executable.

FLEX_VERSION

The version of **flex**.

FLEX_LIBRARIES

The **flex** libraries.

FLEX_INCLUDE_DIRS

The path to the **flex** headers.

The minimum required version of **flex** can be specified using the standard CMake syntax, e.g.

find_package(FLEX 2.5.13).

If **flex** is found on the system, the module defines the macro:

flex_target

```
flex_target(<Name> <FlexInput> <FlexOutput>
           [OPTIONS <options>...]
           [COMPILE_FLAGS <string>]
           [DEFINES_FILE <string>]
           )
```

which creates a custom command to generate the **<FlexOutput>** file from the **<FlexInput>** file. **<Name>** is an alias used to get details of this custom command.

The options are:

OPTIONS <options>...

Added in version 4.0.

*A semicolon-separated list of flex options added to the **flex** command line.*

COMPILE_FLAGS <string>

Deprecated since version 4.0.

Space-separated flex options added to the **flex** command line. A *;-list* will not work. This option is deprecated in favor of **OPTIONS <options>...**

DEFINES_FILE <string>

Added in version 3.5.

If flex is configured to output a header file, this option may be used to specify its name.

Changed in version 3.17: When *CMP0098* is set to **NEW**, **flex** runs in the *CMAKE_CURRENT_BINARY_DIR* directory.

The macro defines the following variables:

FLEX_<Name>_DEFINED

True if the macro ran successfully.

FLEX_<Name>_OUTPUTS

The source file generated by the custom rule, an alias for **<FlexOutput>**.

FLEX_<Name>_INPUT

The flex source file, an alias for **<FlexInput>**.

FLEX_<Name>_OUTPUT_HEADER

The header flex output, if any.

FLEX_<Name>_OPTIONS

Added in version 4.0.

Options used in the **flex** command line.

Flex scanners often use tokens defined by Bison: the code generated by Flex depends of the header generated by Bison. This module also defines a macro:

add_flex_bison_dependency

```
add_flex_bison_dependency(<FlexTarget> <BisonTarget>)
```

which adds the required dependency between a scanner and a parser where **<FlexTarget>** and **<BisonTarget>** are the first parameters of respectively **flex_target** and **bison_target** macros.

Examples

```
find_package(BISON)
find_package(FLEX)

bison_target(MyParser parser.y ${CMAKE_CURRENT_BINARY_DIR}/parser.cpp)
flex_target(MyScanner lexer.l ${CMAKE_CURRENT_BINARY_DIR}/lexer.cpp)
add_flex_bison_dependency(MyScanner MyParser)

include_directories(${CMAKE_CURRENT_BINARY_DIR})
add_executable(Foo
    Foo.cc
    ${BISON_MyParser_OUTPUTS}
    ${FLEX_MyScanner_OUTPUTS}
)
target_link_libraries(Foo ${FLEX_LIBRARIES})
```

Adding additional command-line options to the **flex** executable can be passed as a list. For example, adding the **--warn** option to report warnings, and the **--noline** (**-L**) to not generate **#line** directives.

```
find_package(FLEX)

if(FLEX_FOUND)
    flex_target(MyScanner lexer.l lexer.cpp OPTIONS --warn --noline)
endif()
```

Generator expressions can be used in **OPTIONS** **<options....** For example, to add the **--debug** (**-d**) option only for the **Debug** build type:

```
find_package(FLEX)

if(FLEX_FOUND)
    flex_target(MyScanner lexer.l lexer.cpp OPTIONS $<$<CONFIG:Debug>:--debug>)
endif()
```

FindFLTK

Find the Fast Light Toolkit (FLTK) library

Input Variables

By default this module will search for all of the FLTK components and add them to the **FLTK_LIBRARIES** variable. You can limit the components which get placed in **FLTK_LIBRARIES** by defining one or more of the following three options:

FLTK_SKIP_OPENGL

Set to true to disable searching for the FLTK GL library

FLTK_SKIP_FORMS

Set to true to disable searching for the FLTK Forms library

FLTK_SKIP_IMAGES

Set to true to disable searching for the FLTK Images library

FLTK is composed also by a binary tool. You can set the following option:

FLTK_SKIP_FLUID

Set to true to not look for the FLUID binary

Result Variables

The following variables will be defined:

FLTK_FOUND

True if all components not skipped were found

FLTK_INCLUDE_DIR

Path to the include directory for FLTK header files

FLTK_LIBRARIES

List of the FLTK libraries found

FLTK_FLUID_EXECUTABLE

Path to the FLUID binary tool

FLTK_WRAP_UI

True if FLUID is found, used to enable the FLTK_WRAP_UI command

Cache Variables

The following cache variables are also available to set or use:

FLTK_BASE_LIBRARY_RELEASE

The FLTK base library (optimized)

FLTK_BASE_LIBRARY_DEBUG

The FLTK base library (debug)

FLTK_GL_LIBRARY_RELEASE

The FLTK GL library (optimized)

FLTK_GL_LIBRARY_DEBUG

The FLTK GL library (debug)

FLTK_FORMS_LIBRARY_RELEASE

The FLTK Forms library (optimized)

FLTK_FORMS_LIBRARY_DEBUG

The FLTK Forms library (debug)

FLTK_IMAGES_LIBRARY_RELEASE

The FLTK Images protobuf library (optimized)

FLTK_IMAGES_LIBRARY_DEBUG

The FLTK Images library (debug)

Added in version 3.11: Debug and Release variants are found separately and use per-configuration variables.

FindFLTK2

Find the native FLTK 2.0 includes and library

The following settings are defined

FLTK2_FLUID_EXECUTABLE, where to find the Fluid tool
FLTK2_WRAP_UI, This enables the FLTK2_WRAP_UI command
FLTK2_INCLUDE_DIR, where to find include files

FLTK2_LIBRARIES, list of fltk2 libraries
 FLTK2_FOUND, Don't use FLTK2 if false.

The following settings should not be used in general.

FLTK2_BASE_LIBRARY = the full path to fltk2.lib
 FLTK2_GL_LIBRARY = the full path to fltk2_gl.lib
 FLTK2_IMAGES_LIBRARY = the full path to fltk2_images.lib

FindFontconfig

Added in version 3.14.

Find Fontconfig headers and library.

Imported Targets

Fontconfig::Fontconfig

The Fontconfig library, if found.

Result Variables

This will define the following variables in your project:

Fontconfig_FOUND

true if (the requested version of) Fontconfig is available.

Fontconfig_VERSION

the version of Fontconfig.

Fontconfig_LIBRARIES

the libraries to link against to use Fontconfig.

Fontconfig_INCLUDE_DIRS

where to find the Fontconfig headers.

Fontconfig_COMPILE_OPTIONS

this should be passed to target_compile_options(), if the target is not used for linking

FindFreeType

Find the FreeType font renderer includes and library.

Imported Targets

Added in version 3.10.

This module defines the following *IMPORTED* target:

FreeType::FreeType

The FreeType **freetype** library, if found

Result Variables

This module will set the following variables in your project:

FREETYPE_FOUND

true if the FreeType headers and libraries were found

FREETYPE_INCLUDE_DIRS

directories containing the FreeType headers. This is the concatenation of the variables:

FREETYPE_INCLUDE_DIR_ft2build

directory holding the main FreeType API configuration header

FREETYPE_INCLUDE_DIR_freetype2

directory holding FreeType public headers

FREETYPE_LIBRARIES

the library to link against

FREETYPE_VERSION_STRING

the version of freetype found

Added in version 3.7: Debug and Release variants are found separately.

Hints

The user may set the environment variable **FREETYPE_DIR** to the root directory of a Freetype installation.

FindGCCXML

Find the GCC-XML front-end executable.

This module will define the following variables:

GCCXML – the GCC-XML front-end executable.

FindGDAL

Find Geospatial Data Abstraction Library (GDAL).

Deprecated since version 4.0: GDAL 3.5 and above provide a **GDALConfig.cmake** package configuration file. Call **find_package(GDAL CONFIG)** to find it directly and avoid using this find module. For further details, see *GDAL's documentation on CMake integration*.

Imported Targets

Added in version 3.14.

This module defines *IMPORTED* target **GDAL::GDAL** if GDAL has been found.

Result Variables

This module will set the following variables in your project:

GDAL_FOUND

True if GDAL is found.

GDAL_INCLUDE_DIRS

Include directories for GDAL headers.

GDAL_LIBRARIES

Libraries to link to GDAL.

GDAL_VERSION

Added in version 3.14: The version of GDAL found.

Cache variables

The following cache variables may also be set:

GDAL_LIBRARY

The libgdal library file.

GDAL_INCLUDE_DIR

The directory containing **gdal.h**.

Hints

Set **GDAL_DIR** or **GDAL_ROOT** in the environment to specify the GDAL installation prefix.

The following variables may be set to modify the search strategy:

FindGDAL_SKIP_GDAL_CONFIG

If set, **gdal-config** will not be used. This can be useful if there are GDAL libraries built with auto-tools (which provide the tool) and CMake (which do not) in the same environment.

GDAL_ADDITIONAL_LIBRARY_VERSIONS

Extra versions of library names to search for.

FindGettext

Find GNU gettext tools.

This module looks for the GNU gettext tools.

Result Variables

This module defines the following variables:

GETTEXT_FOUND

True if gettext has been found.

GETTEXT_VERSION_STRING

The version of gettext found.

GETTEXT_MSGMERGE_EXECUTABLE

The full path to the msgmerge tool.

GETTEXT_MSGFMT_EXECUTABLE

The full path to the msgfmt tool.

Functions

This module provides several function.

gettext_create_translations

```
gettext_create_translations(<mofile> [ALL] <file>...)
```

This will create a target "translations" which will convert the given input .po files into the binary output .mo file. Options:

ALL The translations will be created when building the default target.

gettext_process_pot_file

```
gettext_process_pot_file(<potfile> [ALL]
                        [INSTALL_DESTINATION <destdir>]
                        LANGUAGES <lang>...)
```

This function creates a custom target "potfile" which processes the given .pot file to .mo files. Options:

ALL The .pot file will be processed when building the default target.

INSTALL_DESTINATION

Install the results into the given directory (**share/locale/** by default). The language subdirectory will be taken into account.

gettext_process_po_files

```
gettext_process_po_files(<lang> [ALL]
                        [INSTALL_DESTINATION <dir>]
                        PO_FILES <pofile>...)
```

This function creates a custom target "pofiles", which processes the given .po files to .mo files for the given language. Options:

ALL The .po files will be processed when building the default target.

INSTALL_DESTINATION

Install the results into the given directory (**share/locale/** by default). The language subdirectory will be taken into account .

Added in version 3.2: If you wish to use the Gettext runtime library (libintl), use *FindIntl*.

FindGIF

This finds the Graphics Interchange Format (GIF) library (**giflib**)

Imported Targets

This module defines the following *IMPORTED* target:

GIF::GIF

The **giflib** library, if found.

Result variables

This module will set the following variables in your project:

GIF_FOUND

If false, do not try to use GIF.

GIF_INCLUDE_DIRS

where to find gif_lib.h, etc.

GIF_LIBRARIES

the libraries needed to use GIF.

GIF_VERSION

3, 4 or a full version string (eg 5.1.4) for versions >= 4.1.6.

Cache variables

The following cache variables may also be set:

GIF_INCLUDE_DIR

where to find the GIF headers.

GIF_LIBRARY

where to find the GIF library.

Hints

GIF_DIR is an environment variable that would correspond to the `./configure --prefix=$GIF_DIR`.

FindGit

The module defines the following variables:

GIT_EXECUTABLE

Path to Git command-line client.

Git_FOUND, GIT_FOUND

True if the Git command-line client was found.

GIT_VERSION_STRING

The version of Git found.

Added in version 3.14: The module defines the following **IMPORTED** targets (when *CMAKE_ROLE* is **PROJECT**):

Git::Git

Executable of the Git command-line client.

Example usage:

```

    find_package(Git)
    if(Git_FOUND)
        message("Git found: ${GIT_EXECUTABLE}")
    endif()

```

FindGLEW

Find the OpenGL Extension Wrangler Library (GLEW)

Input Variables

The following variables may be set to influence this module's behavior:

GLEW_USE_STATIC_LIBS

to find and create *IMPORTED* target for static linkage.

GLEW_VERBOSE

to output a detailed log of this module.

Imported Targets

Added in version 3.1.

This module defines the following *Imported Targets*:

GLEW::glew

The GLEW shared library.

GLEW::glew_s

The GLEW static library, if **GLEW_USE_STATIC_LIBS** is set to **TRUE**.

GLEW::GLEW

Duplicates either **GLEW::glew** or **GLEW::glew_s** based on availability.

Result Variables

This module defines the following variables:

GLEW_INCLUDE_DIRS

include directories for GLEW

GLEW_LIBRARIES

libraries to link against GLEW

GLEW_SHARED_LIBRARIES

libraries to link against shared GLEW

GLEW_STATIC_LIBRARIES

libraries to link against static GLEW

GLEW_FOUND

true if GLEW has been found and can be used

GLEW_VERSION

GLEW version

GLEW_VERSION_MAJOR

GLEW major version

GLEW_VERSION_MINOR

GLEW minor version

GLEW_VERSION_MICRO

GLEW micro version

Added in version 3.7: Debug and Release variants are found separately.

FindGLUT

Find OpenGL Utility Toolkit (GLUT) library and include files.

Imported Targets

Added in version 3.1.

This module defines the *IMPORTED* targets:

GLUT::GLUT

Defined if the system has GLUT.

Result Variables

This module defines the following variables:

GLUT_FOUND

True if **glut** was found.

GLUT_INCLUDE_DIRS

Added in version 3.23.

Where to find GL/glut.h, etc.

GLUT_LIBRARIES

List of libraries for using **glut**.

Cache Variables

This module may set the following variables depending on platform. These variables may optionally be set to help this module find the correct files, but clients should not use these as results:

GLUT_INCLUDE_DIR

The full path to the directory containing **GL/glut.h**, not including **GL/**.

GLUT_glut_LIBRARY

The full path to the glut library.

GLUT_Xmu_LIBRARY

The full path to the Xmu library.

GLUT_Xi_LIBRARY

The full path to the Xi Library.

Obsolete Variables

The following variables may also be provided, for backwards compatibility:

GLUT_INCLUDE_DIR

This is one of above *Cache Variables*, but prior to CMake 3.23 was also a result variable. Prefer to use **GLUT_INCLUDE_DIRS** instead in CMake 3.23 and above.

FindGnuplot

This module looks for gnuplot.

Result Variables**GNUPLOT_FOUND**

System has Gnuplot.

GNUPLOT_EXECUTABLE

The Gnuplot executable.

GNUPLOT_VERSION_STRING

The version of Gnuplot found.

NOTE:

Version string detection will not work for old versions like 3.7.1.

FindGnuTLS

Find the GNU Transport Layer Security library (gnutls)

Imported Targets

Added in version 3.16.

This module defines *IMPORTED* target **GnuTLS::GnuTLS**, if gnutls has been found.

Result Variables**GNUTLS_FOUND**

System has gnutls

GNUTLS_INCLUDE_DIR

The gnutls include directory

GNUTLS_LIBRARIES

The libraries needed to use gnutls

GNUTLS_DEFINITIONS

Compiler switches required for using gnutls

GNUTLS_VERSION

version of gnutls.

FindGSL

Added in version 3.2.

Find the native GNU Scientific Library (GSL) includes and libraries.

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License.

Imported Targets

If GSL is found, this module defines the following *IMPORTED* targets:

GSL::gsl - The main GSL library.
GSL::gslcblas - The CBLAS support library used by GSL.

Result Variables

This module will set the following variables in your project:

GSL_FOUND - True if GSL found on the local system
GSL_INCLUDE_DIRS - Location of GSL header files.
GSL_LIBRARIES - The GSL libraries.
GSL_VERSION - The version of the discovered GSL install.

Hints

Set **GSL_ROOT_DIR** to a directory that contains a GSL installation.

This script expects to find libraries at **\$GSL_ROOT_DIR/lib** and the GSL headers at **\$GSL_ROOT_DIR/include/gsl**. The library directory may optionally provide Release and Debug folders. If available, the libraries named **gsld**, **gslblasd** or **cblasd** are recognized as debug libraries. For Unix-like systems, this script will use **\$GSL_ROOT_DIR/bin/gsl-config** (if found) to aid in the discovery of GSL.

Cache Variables

This module may set the following variables depending on platform and type of GSL installation discovered. These variables may optionally be set to help this module find the correct files:

GSL_CBLAS_LIBRARY - Location of the GSL CBLAS library.

GSL_CBLAS_LIBRARY_DEBUG – Location of the debug GSL CBLAS library (if any).
GSL_CONFIG_EXECUTABLE – Location of the ``gsl-config`` script (if any).
GSL_LIBRARY – Location of the GSL library.
GSL_LIBRARY_DEBUG – Location of the debug GSL library (if any).

FindGTest

Locate the Google C++ Testing Framework.

Added in version 3.20: Upstream **GTestConfig.cmake** is used if possible.

Imported Targets

Added in version 3.20: This module defines the following *IMPORTED* targets:

GTest::gtest

The Google Test **gtest** library, if found; adds Thread::Thread automatically

GTest::gtest_main

The Google Test **gtest_main** library, if found

Added in version 3.23.

GTest::gmock

The Google Mock **gmock** library, if found; adds Thread::Thread automatically

GTest::gmock_main

The Google Mock **gmock_main** library, if found

Deprecated since version 3.20: For backwards compatibility, this module defines additionally the following deprecated *IMPORTED* targets (available since 3.5):

GTest::GTest

The Google Test **gtest** library, if found; adds Thread::Thread automatically

GTest::Main

The Google Test **gtest_main** library, if found

Result variables

This module will set the following variables in your project:

GTest_FOUND

Found the Google Testing framework

GTEST_INCLUDE_DIRS

the directory containing the Google Test headers

The library variables below are set as normal variables. These contain debug/optimized keywords when a debugging library is found.

GTEST_LIBRARIES

The Google Test **gtest** library; note it also requires linking with an appropriate thread library

GTEST_MAIN_LIBRARIES

The Google Test **gtest_main** library

GTEST_BOTH_LIBRARIES

Both **gtest** and **gtest_main**

Cache variables

The following cache variables may also be set:

GTEST_ROOT

The root directory of the Google Test installation (may also be set as an environment variable)

GTEST_MSVC_SEARCH

If compiling with MSVC, this variable can be set to **MT** or **MD** (the default) to enable searching a GTest build tree

Example usage

```
enable_testing()
find_package(GTest REQUIRED)

add_executable(foo foo.cc)
target_link_libraries(foo GTest::gtest GTest::gtest_main)

add_test(AllTestsInFoo foo)
```

Deeper integration with CTest

See *GoogleTest* for information on the *gtest_add_tests()* and *gtest_discover_tests()* commands.

Changed in version 3.9: Previous CMake versions defined *gtest_add_tests()* macro in this module.

FindGTK

Find GTK, glib and GTKGLArea

```
GTK_INCLUDE_DIR    - Directories to include to use GTK
GTK_LIBRARIES      - Files to link against to use GTK
GTK_FOUND          - GTK was found
GTK_GL_FOUND       - GTK's GL features were found
```

FindGTK2

Find the GTK2 widget libraries and several of its other optional components like **gtkmm**, **glade**, and **glademm**.

Specify one or more of the following components as you call this find module. See example below.

- **gtk**
- **gtkmm**
- **glade**
- **glademm**

Imported Targets

This module defines the following *IMPORTED* targets (subject to component selection):

```
GTK2::atk, GTK2::atkmm, GTK2::cairo, GTK2::caiomm, GTK2::gdk_pixbuf, GTK2::gdk,
GTK2::gdkmm, GTK2::gio, GTK2::giomm, GTK2::glade, GTK2::glademm, GTK2::glib,
GTK2::glibmm, GTK2::gmodule, GTK2::gobject, GTK2::gthread, GTK2::gtk, GTK2::gtkmm,
GTK2::harfbuzz, GTK2::pango, GTK2::pangocairo, GTK2::pangoft2, GTK2::pangomm,
GTK2::pangoxft, GTK2::sigc.
```

Added in version 3.16.7: Added the **GTK2::harfbuzz** target.

Result Variables

The following variables will be defined for your use

GTK2_FOUND

Were all of your specified components found?

GTK2_INCLUDE_DIRS

All include directories

GTK2_LIBRARIES

All libraries

GTK2_TARGETS

Added in version 3.5: All imported targets

GTK2_DEFINITIONS

Additional compiler flags

GTK2_VERSION

The version of GTK2 found (x.y.z)

GTK2_MAJOR_VERSION

The major version of GTK2

GTK2_MINOR_VERSION

The minor version of GTK2

GTK2_PATCH_VERSION

The patch version of GTK2

Added in version 3.5: When **GTK2_USE_IMPORTED_TARGETS** is set to **TRUE**, **GTK2_LIBRARIES** will list imported targets instead of library paths.

Input Variables

Optional variables you can define prior to calling this module:

GTK2_DEBUG

Enables verbose debugging of the module

GTK2_ADDITIONAL_SUFFIXES

Allows defining additional directories to search for include files

Example Usage

Call *find_package()* once. Here are some examples to pick from:

Require GTK 2.6 or later:

```
find_package(GTK2 2.6 REQUIRED gtk)
```

Require GTK 2.10 or later and Glade:

```
find_package(GTK2 2.10 REQUIRED gtk glade)
```

Search for GTK/GTKMM 2.8 or later:

```
find_package(GTK2 2.8 COMPONENTS gtk gtkmm)
```

Use the results:

```
if(GTK2_FOUND)
  include_directories(${GTK2_INCLUDE_DIRS})
  add_executable(mygui mygui.cc)
  target_link_libraries(mygui ${GTK2_LIBRARIES})
```

```
endif()
```

FindHDF5

Find Hierarchical Data Format (HDF5), a library for reading and writing self describing array data.

This module invokes the **HDF5** wrapper compiler that should be installed alongside **HDF5**. Depending upon the **HDF5** Configuration, the wrapper compiler is called either **h5cc** or **h5pcc**. If this succeeds, the module will then call the compiler with the `show` argument to see what flags are used when compiling an **HDF5** client application.

The module will optionally accept the **COMPONENTS** argument. If no **COMPONENTS** are specified, then the find module will default to finding only the **HDF5** C library. If one or more **COMPONENTS** are specified, the module will attempt to find the language bindings for the specified components. The valid components are **C**, **CXX**, **Fortran**, **HL**. **HL** refers to the "high-level" HDF5 functions for C and Fortran. If the **COMPONENTS** argument is not given, the module will attempt to find only the C bindings. For example, to use Fortran HDF5 and HDF5-HL functions, do: **find_package(HDF5 COMPONENTS Fortran HL)**.

This module will read the variable **HDF5_USE_STATIC_LIBRARIES** to determine whether or not to prefer a static link to a dynamic link for **HDF5** and all of it's dependencies. To use this feature, make sure that the **HDF5_USE_STATIC_LIBRARIES** variable is set before the call to **find_package**.

Added in version 3.10: Support for **HDF5_USE_STATIC_LIBRARIES** on Windows.

Both the serial and parallel **HDF5** wrappers are considered and the first directory to contain either one will be used. In the event that both appear in the same directory the serial version is preferentially selected. This behavior can be reversed by setting the variable **HDF5_PREFER_PARALLEL** to **TRUE**.

In addition to finding the includes and libraries required to compile an **HDF5** client application, this module also makes an effort to find tools that come with the **HDF5** distribution that may be useful for regression testing.

Result Variables

This module will set the following variables in your project:

HDF5_FOUND

HDF5 was found on the system

HDF5_VERSION

Added in version 3.3: HDF5 library version

HDF5_INCLUDE_DIRS

Location of the HDF5 header files

HDF5_DEFINITIONS

Required compiler definitions for HDF5

HDF5_LIBRARIES

Required libraries for all requested bindings

HDF5_HL_LIBRARIES

Required libraries for the HDF5 high level API for all bindings, if the **HL** component is enabled

Available components are: **C** **CXX** **Fortran** and **HL**. For each enabled language binding, a corresponding **HDF5_\${LANG}_LIBRARIES** variable, and potentially **HDF5_\${LANG}_DEFINITIONS**, will be defined. If the **HL** component is enabled, then an **HDF5_\${LANG}_HL_LIBRARIES** will also be defined. With all components enabled, the following variables will be defined:

HDF5_C_DEFINITIONS

Required compiler definitions for HDF5 C bindings

HDF5_CXX_DEFINITIONS

Required compiler definitions for HDF5 C++ bindings

HDF5_Fortran_DEFINITIONS

Required compiler definitions for HDF5 Fortran bindings

HDF5_C_INCLUDE_DIRS

Required include directories for HDF5 C bindings

HDF5_CXX_INCLUDE_DIRS

Required include directories for HDF5 C++ bindings

HDF5_Fortran_INCLUDE_DIRS

Required include directories for HDF5 Fortran bindings

HDF5_C_LIBRARIES

Required libraries for the HDF5 C bindings

HDF5_CXX_LIBRARIES

Required libraries for the HDF5 C++ bindings

HDF5_Fortran_LIBRARIES

Required libraries for the HDF5 Fortran bindings

HDF5_C_HL_LIBRARIES

Required libraries for the high level C bindings

HDF5_CXX_HL_LIBRARIES

Required libraries for the high level C++ bindings

HDF5_Fortran_HL_LIBRARIES

Required libraries for the high level Fortran bindings.

HDF5_IS_PARALLEL

HDF5 library has parallel IO support

HDF5_C_COMPILER_EXECUTABLE

path to the HDF5 C wrapper compiler

HDF5_CXX_COMPILER_EXECUTABLE

path to the HDF5 C++ wrapper compiler

HDF5_Fortran_COMPILER_EXECUTABLE

path to the HDF5 Fortran wrapper compiler

HDF5_C_COMPILER_EXECUTABLE_NO_INTERROGATE

path to the primary C compiler which is also the HDF5 wrapper

HDF5_CXX_COMPILER_EXECUTABLE_NO_INTERROGATE

path to the primary C++ compiler which is also the HDF5 wrapper

HDF5_Fortran_COMPILER_EXECUTABLE_NO_INTERROGATE

path to the primary Fortran compiler which is also the HDF5 wrapper

HDF5_DIFF_EXECUTABLE

path to the HDF5 dataset comparison tool

With all components enabled, the following targets will be defined:

HDF5::HDF5

All detected **HDF5_LIBRARIES**.

hdf5::hdf5

C library.

hdf5::hdf5_cpp

C++ library.

hdf5::hdf5_fortran

Fortran library.

hdf5::hdf5_hl

High-level C library.

hdf5::hdf5_hl_cpp

High-level C++ library.

hdf5::hdf5_hl_fortran

High-level Fortran library.

hdf5::h5diff**h5diff** executable.**Hints**

The following variables can be set to guide the search for HDF5 libraries and includes:

HDF5_PREFER_PARALLEL

Added in version 3.4.

set **true** to prefer parallel HDF5 (by default, serial is preferred)

HDF5_FIND_DEBUG

Added in version 3.9.

Set **true** to get extra debugging output.

HDF5_NO_FIND_PACKAGE_CONFIG_FILE

Added in version 3.8.

Set **true** to skip trying to find **hdf5-config.cmake**.

FindHg

Extract information from a mercurial working copy.

The module defines the following variables:

HG_EXECUTABLE - path to mercurial command line client (hg)

HG_FOUND - true if the command line client was found

HG_VERSION_STRING - the version of mercurial found

Added in version 3.1: If the command line client executable is found the following macro is defined:

HG_WC_INFO(<dir> <var-prefix>)

Hg_WC_INFO extracts information of a mercurial working copy at a given location. This macro defines the following variables:

<var-prefix>_WC_CHANGESSET - current changeset

<var-prefix>_WC_REVISION - current revision

Example usage:

```
find_package(Hg)
if(HG_FOUND)
  message("hg found: ${HG_EXECUTABLE}")
  HG_WC_INFO(${PROJECT_SOURCE_DIR} Project)
  message("Current revision is ${Project_WC_REVISION}")
  message("Current changeset is ${Project_WC_CHANGESET}")
endif()
```

FindHSPELL

Try to find Hebrew spell-checker (Hspell) and morphology engine.

Once done this will define

```
HSPELL_FOUND - system has Hspell
HSPELL_INCLUDE_DIR - the Hspell include directory
HSPELL_LIBRARIES - The libraries needed to use Hspell
HSPELL_DEFINITIONS - Compiler switches required for using Hspell

HSPELL_VERSION_STRING - The version of Hspell found (x.y)
HSPELL_MAJOR_VERSION - the major version of Hspell
HSPELL_MINOR_VERSION - The minor version of Hspell
```

FindHTMLHelp

This module looks for Microsoft HTML Help Compiler

It defines:

```
HTML_HELP_COMPILER      : full path to the Compiler (hhc.exe)
HTML_HELP_INCLUDE_PATH  : include path to the API (htmlhelp.h)
HTML_HELP_LIBRARY       : full path to the library (htmlhelp.lib)
```

FindIce

Added in version 3.1.

Find the ZeroC Internet Communication Engine (ICE) programs, libraries and datafiles.

This module supports multiple components. Components can include any of: **Freeze**, **Glacier2**, **Ice**, **IceBox**, **IceDB**, **IceDiscovery**, **IceGrid**, **IceLocatorDiscovery**, **IcePatch**, **IceSSL**, **IceStorm**, **IceUtil**, **IceXML**, or **Slice**.

Ice 3.7 and later also include C++11-specific components: **Glacier2++11**, **Ice++11**, **IceBox++11**, **IceDiscovery++11**, **IceGrid**, **IceLocatorDiscovery++11**, **IceSSL++11**, **IceStorm++11**

Note that the set of supported components is Ice version-specific.

Added in version 3.4: Imported targets for components and most **EXECUTABLE** variables.

Added in version 3.7: Debug and Release variants are found separately.

Added in version 3.10: Ice 3.7 support, including new components, programs and the Nuget package.

This module reports information about the Ice installation in several variables. General variables:

```
Ice_VERSION - Ice release version
Ice_FOUND - true if the main programs and libraries were found
Ice_LIBRARIES - component libraries to be linked
Ice_INCLUDE_DIRS - the directories containing the Ice headers
Ice_SLICE_DIRS - the directories containing the Ice slice interface
                  definitions
```

Imported targets:

```
Ice::<C>
```

Where <C> is the name of an Ice component, for example **Ice::Glacier2** or **Ice++11**.

Ice slice programs are reported in:

```
Ice_SLICE2CONFLUENCE_EXECUTABLE - path to slice2confluence executable
Ice_SLICE2CPP_EXECUTABLE - path to slice2cpp executable
Ice_SLICE2CS_EXECUTABLE - path to slice2cs executable
Ice_SLICE2FREEZEJ_EXECUTABLE - path to slice2freezej executable
Ice_SLICE2FREEZE_EXECUTABLE - path to slice2freeze executable
Ice_SLICE2HTML_EXECUTABLE - path to slice2html executable
Ice_SLICE2JAVA_EXECUTABLE - path to slice2java executable
Ice_SLICE2JS_EXECUTABLE - path to slice2js executable
Ice_SLICE2MATLAB_EXECUTABLE - path to slice2matlab executable
Ice_SLICE2OBJC_EXECUTABLE - path to slice2objc executable
Ice_SLICE2PHP_EXECUTABLE - path to slice2php executable
Ice_SLICE2PY_EXECUTABLE - path to slice2py executable
Ice_SLICE2RB_EXECUTABLE - path to slice2rb executable
```

Added in version 3.14: Variables for **slice2confluence** and **slice2matlab**.

Ice programs are reported in:

```
Ice_GLACIER2ROUTER_EXECUTABLE - path to glacier2router executable
Ice_ICEBOX_EXECUTABLE - path to icebox executable
Ice_ICEBOXXX11_EXECUTABLE - path to icebox++11 executable
Ice_ICEBOXADMIN_EXECUTABLE - path to iceboxadmin executable
Ice_ICEBOXD_EXECUTABLE - path to iceboxd executable
Ice_ICEBOXNET_EXECUTABLE - path to iceboxnet executable
Ice_ICEBRIDGE_EXECUTABLE - path to icebridge executable
Ice_ICEGRIDADMIN_EXECUTABLE - path to icegridadmin executable
Ice_ICEGRIDDB_EXECUTABLE - path to icegriddb executable
Ice_ICEGRIDNODE_EXECUTABLE - path to icegridnode executable
Ice_ICEGRIDNODED_EXECUTABLE - path to icegridnoded executable
Ice_ICEGRIDREGISTRY_EXECUTABLE - path to icegridregistry executable
Ice_ICEGRIDREGISTRYD_EXECUTABLE - path to icegridregistryd executable
Ice_ICEPATCH2CALC_EXECUTABLE - path to icepatch2calc executable
Ice_ICEPATCH2CLIENT_EXECUTABLE - path to icepatch2client executable
Ice_ICEPATCH2SERVER_EXECUTABLE - path to icepatch2server executable
Ice_ICESERVICEINSTALL_EXECUTABLE - path to iceserviceinstall executable
Ice_ICESTORMADMIN_EXECUTABLE - path to icestormadmin executable
```

Ice_ICESTORMDB_EXECUTABLE – path to icestormdb executable
 Ice_ICESTORMMIGRATE_EXECUTABLE – path to icestormmigrate executable

Ice db programs (Windows only; standard system versions on all other platforms) are reported in:

Ice_DB_ARCHIVE_EXECUTABLE – path to db_archive executable
 Ice_DB_CHECKPOINT_EXECUTABLE – path to db_checkpoint executable
 Ice_DB_DEADLOCK_EXECUTABLE – path to db_deadlock executable
 Ice_DB_DUMP_EXECUTABLE – path to db_dump executable
 Ice_DB_HOTBACKUP_EXECUTABLE – path to db_hotbackup executable
 Ice_DB_LOAD_EXECUTABLE – path to db_load executable
 Ice_DB_LOG_VERIFY_EXECUTABLE – path to db_log_verify executable
 Ice_DB_PRINTLOG_EXECUTABLE – path to db_printlog executable
 Ice_DB_RECOVER_EXECUTABLE – path to db_recover executable
 Ice_DB_STAT_EXECUTABLE – path to db_stat executable
 Ice_DB_TUNER_EXECUTABLE – path to db_tuner executable
 Ice_DB_UPGRADE_EXECUTABLE – path to db_upgrade executable
 Ice_DB_VERIFY_EXECUTABLE – path to db_verify executable
 Ice_DUMPDB_EXECUTABLE – path to dumpdb executable
 Ice_TRANSFORMDB_EXECUTABLE – path to transformdb executable

Ice component libraries are reported in:

Ice_<C>_FOUND – ON if component was found
 Ice_<C>_LIBRARIES – libraries for component

Note that <C> is the uppercased name of the component.

This module reads hints about search results from:

Ice_HOME – the root of the Ice installation

The environment variable **ICE_HOME** may also be used; the Ice_HOME variable takes precedence.

NOTE:

On Windows, Ice 3.7.0 and later provide libraries via the NuGet package manager. Appropriate NuGet packages will be searched for using **CMAKE_PREFIX_PATH**, or alternatively **Ice_HOME** may be set to the location of a specific NuGet package to restrict the search.

The following cache variables may also be set:

Ice_<P>_EXECUTABLE – the path to executable <P>
 Ice_INCLUDE_DIR – the directory containing the Ice headers
 Ice_SLICE_DIR – the directory containing the Ice slice interface definitions
 Ice_<C>_LIBRARY – the library for component <C>

NOTE:

In most cases none of the above variables will require setting, unless multiple Ice versions are available and a specific version is required. On Windows, the most recent version of Ice will be found through the registry. On Unix, the programs, headers and libraries will usually be in standard locations, but Ice_SLICE_DIRS might not be automatically detected (commonly known locations are searched). All the other variables are defaulted using Ice_HOME, if set. It's possible to set Ice_HOME and selectively specify alternative locations for the other components; this might be required for e.g. newer

versions of Visual Studio if the heuristics are not sufficient to identify the correct programs and libraries for the specific Visual Studio version.

Other variables one may set to control this module are:

Ice_DEBUG – Set to ON to enable debug output from FindIce.

FindIconv

Added in version 3.11.

This module finds the **iconv()** POSIX.1 functions on the system. These functions might be provided in the regular C library or externally in the form of an additional library.

The following variables are provided to indicate iconv support:

Iconv_FOUND

Variable indicating if the iconv support was found.

Iconv_INCLUDE_DIRS

The directories containing the iconv headers.

Iconv_LIBRARIES

The iconv libraries to be linked.

Iconv_VERSION

Added in version 3.21.

The version of iconv found (x.y)

Iconv_VERSION_MAJOR

Added in version 3.21.

The major version of iconv

Iconv_VERSION_MINOR

Added in version 3.21.

The minor version of iconv

Iconv_IS_BUILT_IN

A variable indicating whether iconv support is stemming from the C library or not. Even if the C library provides *iconv()*, the presence of an external *libiconv* implementation might lead to this being false.

Additionally, the following *IMPORTED* target is being provided:

Iconv::Iconv

Imported target for using iconv.

The following cache variables may also be set:

Iconv_INCLUDE_DIR

The directory containing the iconv headers.

Iconv_LIBRARY

The iconv library (if not implicitly given in the C library).

NOTE:

On POSIX platforms, iconv might be part of the C library and the cache variables **Iconv_INCLUDE_DIR** and **Iconv_LIBRARY** might be empty.

NOTE:

Some libiconv implementations don't embed the version number in their header files. In this case the variables **Iconv_VERSION*** will be empty.

FindIcotoool

Find icotoool

This module looks for icotoool. Convert and create Win32 icon and cursor files. This module defines the following values:

ICOTOOL_EXECUTABLE: the full path to the icotoool tool.
 ICOTOOL_FOUND: True if icotoool has been found.
 ICOTOOL_VERSION_STRING: the version of icotoool found.

FindICU

Added in version 3.7.

Find the International Components for Unicode (ICU) libraries and programs.

This module supports multiple components. Components can include any of: **data**, **i18n**, **io**, **le**, **lx**, **test**, **tu** and **uc**.

Note that on Windows **data** is named **dt** and **i18n** is named **in**; any of the names may be used, and the appropriate platform-specific library name will be automatically selected.

Added in version 3.11: Added support for static libraries on Windows.

This module reports information about the ICU installation in several variables. General variables:

ICU_FOUND - true if the main programs and libraries were found
 ICU_INCLUDE_DIRS - the directories containing the ICU headers
 ICU_LIBRARIES - component libraries to be linked
 ICU_VERSION - ICU release version

Imported targets:

ICU: :<C>

Where <C> is the name of an ICU component, for example **ICU::i18n**; <C> is lower-case.

ICU programs are reported in:

ICU_GENCNVAL_EXECUTABLE - path to gencnval executable
 ICU_ICUINFO_EXECUTABLE - path to icuinfo executable
 ICU_GENBRK_EXECUTABLE - path to genbrk executable
 ICU_ICU-CONFIG_EXECUTABLE - path to icu-config executable
 ICU_GENRB_EXECUTABLE - path to genrb executable
 ICU_GENDICT_EXECUTABLE - path to gendict executable
 ICU_DERB_EXECUTABLE - path to derb executable
 ICU_PKGDATA_EXECUTABLE - path to pkgdata executable

ICU_UCONV_EXECUTABLE – path to uconv executable
 ICU_GENCFU_EXECUTABLE – path to gencfu executable
 ICU_MAKECONV_EXECUTABLE – path to makeconv executable
 ICU_GENNORM2_EXECUTABLE – path to gennorm2 executable
 ICU_GENCCODE_EXECUTABLE – path to genccode executable
 ICU_GENSPREP_EXECUTABLE – path to gensprep executable
 ICU_ICUPKG_EXECUTABLE – path to icupkg executable
 ICU_GENCMN_EXECUTABLE – path to gencmn executable

ICU component libraries are reported in:

ICU_<C>_FOUND – ON if component was found; ``<C>`` is upper-case.
 ICU_<C>_LIBRARIES – libraries for component; ``<C>`` is upper-case.

ICU datafiles are reported in:

ICU_MAKEFILE_INC – Makefile.inc
 ICU_PKGDATA_INC – pkgdata.inc

This module reads hints about search results from:

ICU_ROOT – the root of the ICU installation

The environment variable **ICU_ROOT** may also be used; the ICU_ROOT variable takes precedence.

The following cache variables may also be set:

ICU_<P>_EXECUTABLE – the path to executable <P>; ``<P>`` is upper-case.
 ICU_INCLUDE_DIR – the directory containing the ICU headers
 ICU_<C>_LIBRARY – the library for component <C>; ``<C>`` is upper-case.

NOTE:

In most cases none of the above variables will require setting, unless multiple ICU versions are available and a specific version is required.

FindImageMagick

Find ImageMagick, software suite for displaying, converting and manipulating raster images.

Added in version 3.9: Added support for ImageMagick 7.

This module will search for a set of ImageMagick tools specified as components in the *find_package()* call. Typical components include, but are not limited to (future versions of ImageMagick might have additional components not listed here):

- **animate**
- **compare**
- **composite**
- **conjure**
- **convert**
- **display**

- **identify**
- **import**
- **mogrify**
- **montage**
- **stream**

If no component is specified in the *find_package()* call, then it only searches for the ImageMagick executable directory.

There are also components for the following ImageMagick APIs:

- **Magick++**: ImageMagick C++ API, if found.
- **MagickWand**: ImageMagick MagickWand C API, if found.
- **MagickCore**: ImageMagick MagickCore low-level C API, if found.

Imported Targets

Added in version 3.26.

This module defines the following *IMPORTED* targets:

ImageMagick::Magick++

ImageMagick C++ API, if found.

ImageMagick::MagickWand

ImageMagick MagickWand C API, if found.

ImageMagick::MagickCore

ImageMagick MagickCore low-level C API, if found.

Result Variables

ImageMagick_FOUND

TRUE if all components are found.

ImageMagick_EXECUTABLE_DIR

Full path to executables directory.

ImageMagick_INCLUDE_DIRS

Full paths to all include dirs.

ImageMagick_LIBRARIES

Full paths to all libraries.

ImageMagick_COMPILE_OPTIONS

Compile options of all libraries.

ImageMagick_VERSION_STRING

The version of ImageMagick found (since CMake 2.8.8). Will not work for old versions like 5.2.3.

ImageMagick_<component>_FOUND

TRUE if <component> is found.

ImageMagick_<component>_EXECUTABLE

Full path to <component> executable.

ImageMagick_<component>_INCLUDE_DIRS

Full path to <component> include dirs.

ImageMagick_<component>_COMPILE_OPTIONS

Added in version 3.26.

Compile options of <component>.

ImageMagick_<component>_LIBRARIES

Added in version 3.31.

Full path to <component> libraries.

Example Usage

```
find_package(ImageMagick COMPONENTS Magick++)
target_link_libraries(example PRIVATE ImageMagick::Magick++)
```

FindIntl

Added in version 3.2.

Find the Gettext libintl headers and libraries.

This module reports information about the Gettext libintl installation in several variables.

Intl_FOUND

True if libintl is found.

Intl_INCLUDE_DIRS

The directory containing the libintl headers.

Intl_LIBRARIES

The intl libraries to be linked.

Intl_VERSION

Added in version 3.21.

The version of intl found (x.y.z)

Intl_VERSION_MAJOR

Added in version 3.21.

The major version of intl

Intl_VERSION_MINOR

Added in version 3.21.

The minor version of intl

Intl_VERSION_PATCH

Added in version 3.21.

The patch version of intl

Added in version 3.20: This module defines *IMPORTED* target **Intl::Intl**.

The following cache variables may also be set:

Intl_INCLUDE_DIR

The directory containing the libintl headers

Intl_LIBRARY

The libintl library (if any)

Intl_IS_BUILT_IN

Added in version 3.20.

whether **intl** is a part of the C library.

NOTE:

On some platforms, such as Linux with GNU libc, the gettext functions are present in the C standard library and libintl is not required. **Intl_LIBRARIES** will be empty in this case.

NOTE:

Some libintl implementations don't embed the version number in their header files. In this case the variables **Intl_VERSION*** will be empty.

NOTE:

If you wish to use the Gettext tools (**msgmerge**, **msgfmt**, etc.), use *FindGettext*.

FindJasper

Find the Jasper JPEG2000 library.

Imported Targets**Jasper::Jasper**

The jasper library, if found.

Result Variables

This module defines the following variables:

JASPER_FOUND

system has Jasper

JASPER_INCLUDE_DIRS

Added in version 3.22.

the Jasper include directory

JASPER_LIBRARIES

the libraries needed to use Jasper

JASPER_VERSION_STRING

the version of Jasper found

Cache variables

The following cache variables may also be set:

JASPER_INCLUDE_DIR

where to find jasper/jasper.h, etc.

JASPER_LIBRARY_RELEASE

where to find the Jasper library (optimized).

JASPER_LIBRARY_DEBUG

where to find the Jasper library (debug).

FindJava

Find Java

This module finds if Java is installed and determines where the include files and libraries are. The caller may set variable **JAVA_HOME** to specify a Java installation prefix explicitly.

See also the *FindJNI* module to find Java Native Interface (JNI).

Added in version 3.10: Added support for Java 9+ version parsing.

Specify one or more of the following components as you call this find module. See example below.

```
Runtime      = Java Runtime Environment used to execute Java byte-compiled applica
Development = Development tools (java, javac, javah, jar and javadoc), includes R
IdlJ         = Interface Description Language (IDL) to Java compiler
JarSigner    = Signer and verifier tool for Java Archive (JAR) files
```

This module sets the following result variables:

```
Java_JAVA_EXECUTABLE      = the full path to the Java runtime
Java_JAVAC_EXECUTABLE     = the full path to the Java compiler
Java_JAVAH_EXECUTABLE     = the full path to the Java header generator
Java_JAVADOC_EXECUTABLE   = the full path to the Java documentation generator
Java_IDLJ_EXECUTABLE      = the full path to the Java idl compiler
Java_JAR_EXECUTABLE       = the full path to the Java archiver
Java_JARSIGNER_EXECUTABLE = the full path to the Java jar signer
Java_VERSION_STRING       = Version of java found, eg. 1.6.0_12
Java_VERSION_MAJOR        = The major version of the package found.
Java_VERSION_MINOR        = The minor version of the package found.
Java_VERSION_PATCH        = The patch version of the package found.
Java_VERSION_TWEAK        = The tweak version of the package found (after '_')
Java_VERSION              = This is set to: $major[.$minor[.$patch[.$tweak]]]
```

Added in version 3.4: Added the **Java_IDLJ_EXECUTABLE** and **Java_JARSIGNER_EXECUTABLE** variables.

The minimum required version of Java can be specified using the *find_package()* syntax, e.g.

```
find_package(Java 1.8)
```

NOTE: **\${Java_VERSION}** and **\${Java_VERSION_STRING}** are not guaranteed to be identical. For example some java version may return: **Java_VERSION_STRING** = **1.8.0_17** and **Java_VERSION** = **1.8.0.17**

another example is the Java OEM, with: **Java_VERSION_STRING** = **1.8.0-oem** and **Java_VERSION** = **1.8.0**

For these components the following variables are set:

```
Java_FOUND              - TRUE if all components are found.
Java_<component>_FOUND - TRUE if <component> is found.
```

Example Usages:

```
find_package(Java)
find_package(Java 1.8 REQUIRED)
find_package(Java COMPONENTS Runtime)
find_package(Java COMPONENTS Development)
```

FindJNI

Find Java Native Interface (JNI) headers and libraries.

JNI enables Java code running in a Java Virtual Machine (JVM) or Dalvik Virtual Machine (DVM) on Android to call and be called by native applications and libraries written in other languages such as C and C++.

This module finds if Java is installed and determines where the include files and libraries are. It also determines what the name of the library is. The caller may set variable **JAVA_HOME** to specify a Java installation prefix explicitly.

Added in version 3.24: Added imported targets, components **AWT**, **JVM**, and Android NDK support. If no components are specified, the module defaults to an empty components list while targeting Android, and all available components otherwise.

When using Android NDK, the corresponding package version is reported and a specific release can be requested. At Android API level 31 and above, the additional **NativeHelper** component can be requested. **NativeHelper** is also exposed as an implicit dependency of the **JVM** component (only if this does not cause a conflict) which provides a uniform access to JVM functions.

Imported Targets

Added in version 3.24.

JNI::JNI

Main JNI target, defined only if **jni.h** was found.

JNI::AWT

Java AWT Native Interface (JAWT) library, defined only if component **AWT** was found.

JNI::JVM

Java Virtual Machine (JVM) library, defined only if component **JVM** was found.

JNI::NativeHelper

When targeting Android API level 31 and above, the import target will provide access to **libnativehelper.so** that exposes JVM functions such as **JNI_CreateJavaVM**.

Result Variables

This module sets the following result variables:

JNI_INCLUDE_DIRS

The include directories to use.

JNI_LIBRARIES

The libraries to use (JAWT and JVM).

JNI_FOUND

TRUE if JNI headers and libraries were found.

JNI_<component>_FOUND

Added in version 3.24.

TRUE if <component> was found.

JNI_VERSION

Full Android NDK package version (including suffixes such as **-beta3** and **-rc1**) or undefined otherwise.

JNI_VERSION_MAJOR

Added in version 3.24.

Android NDK major version or undefined otherwise.

JNI_VERSION_MINOR

Added in version 3.24.

Android NDK minor version or undefined otherwise.

JNI_VERSION_PATCH

Added in version 3.24.

Android NDK patch version or undefined otherwise.

Cache Variables

The following cache variables are also available to set or use:

JAVA_AWT_LIBRARY

The path to the Java AWT Native Interface (JAWT) library.

JAVA_JVM_LIBRARY

The path to the Java Virtual Machine (JVM) library.

JAVA_INCLUDE_PATH

The include path to **jni.h**.

JAVA_INCLUDE_PATH2

The include path to machine-dependent headers **jni_md.h** and **jniport.h**. The variable is defined only if **jni.h** depends on one of these headers. In contrast, Android NDK **jni.h** can be typically used standalone.

JAVA_AWT_INCLUDE_PATH

The include path to **jawt.h**.

FindJPEG

Find the Joint Photographic Experts Group (JPEG) library (**libjpeg**)

Imported Targets

Added in version 3.12.

This module defines the following *IMPORTED* targets:

JPEG::JPEG

The JPEG library, if found.

Result variables

This module will set the following variables in your project:

JPEG_FOUND

If false, do not try to use JPEG.

JPEG_INCLUDE_DIRS

where to find jpeglib.h, etc.

JPEG_LIBRARIES

the libraries needed to use JPEG.

JPEG_VERSION

Added in version 3.12: the version of the JPEG library found

Cache variables

The following cache variables may also be set:

JPEG_INCLUDE_DIR

where to find jpeglib.h, etc.

JPEG_LIBRARY_RELEASE

where to find the JPEG library (optimized).

JPEG_LIBRARY_DEBUG

where to find the JPEG library (debug).

Added in version 3.12: Debug and Release variant are found separately.

Obsolete variables**JPEG_LIBRARY**

where to find the JPEG library.

FindKDE3

Find the KDE3 include and library dirs, KDE preprocessors and define a some macros

This module defines the following variables:

KDE3_DEFINITIONS

compiler definitions required for compiling KDE software

KDE3_INCLUDE_DIR

the KDE include directory

KDE3_INCLUDE_DIRS

the KDE and the Qt include directory, for use with include_directories()

KDE3_LIB_DIR

the directory where the KDE libraries are installed, for use with link_directories()

QT_AND_KDECORE_LIBS

this contains both the Qt and the kdec core library

KDE3_DCOPIDL_EXECUTABLE

the dco pidl executable

KDE3_DCOPIDL2CPP_EXECUTABLE

the dco pidl2cpp executable

KDE3_KCFG_COMPILER_EXECUTABLE

the kconfig_compiler executable

KDE3_FOUND

set to TRUE if all of the above has been found

The following user adjustable options are provided:

KDE3_BUILD_TESTS

enable this to build KDE testcases

It also adds the following macros (from **KDE3Macros.cmake**) **SRCS_VAR** is always the variable which contains the list of source files for your application or library.

KDE3_AUTOMOC(file1 ... fileN)

Call this if you want to have automatic moc file handling. This means if you include "foo.moc" in the source file foo.cpp a moc file for the header foo.h will be created automatically. You can set the property **SKIP_AUTOMAKE** using **set_source_files_properties()** to exclude some files in the list from being processed.

KDE3_ADD_MOC_FILES(**SRCS_VAR** file1 ... fileN)

If you don't use the **KDE3_AUTOMOC()** macro, for the files listed here moc files will be created (named "foo.moc.cpp")

KDE3_ADD_DCOP_SKELS(**SRCS_VAR** header1.h ... headerN.h)

Use this to generate DCOP skeletons from the listed headers.

KDE3_ADD_DCOP_STUBS(**SRCS_VAR** header1.h ... headerN.h)

Use this to generate DCOP stubs from the listed headers.

KDE3_ADD_UI_FILES(**SRCS_VAR** file1.ui ... fileN.ui)

Use this to add the Qt designer ui files to your application/library.

KDE3_ADD_KCFG_FILES(**SRCS_VAR** file1.kcfgc ... fileN.kcfgc)

Use this to add KDE kconfig compiler files to your application/library.

KDE3_INSTALL_LIBTOOL_FILE(target)

This will create and install a simple libtool file for the given target.

KDE3_ADD_EXECUTABLE(name file1 ... fileN)

Currently identical to **add_executable()**, may provide some advanced features in the future.

KDE3_ADD_KPART(name [WITH_PREFIX] file1 ... fileN)

Create a KDE plugin (KPart, kioslave, etc.) from the given source files. If **WITH_PREFIX** is given, the resulting plugin will have the prefix "lib", otherwise it won't. It creates and installs an appropriate libtool la-file.

KDE3_ADD_KDEINIT_EXECUTABLE(name file1 ... fileN)

Create a KDE application in the form of a module loadable via **kdeinit**.

A library named `kdeinit_<name>` will be created and a small executable which links to it.

The option `KDE3_ENABLE_FINAL` to enable all-in-one compilation is no longer supported.

Author: Alexander Neundorf <neundorf@kde.org>

FindKDE4

Find KDE4 and provide all necessary variables and macros to compile software for it. It looks for KDE 4 in the following directories in the given order:

```
CMAKE_INSTALL_PREFIX
KDEDIRS
/opt/kde4
```

Please look in **FindKDE4Internal.cmake** and **KDE4Macros.cmake** for more information. They are installed with the KDE 4 libraries in `$KDEDIRS/share/apps/cmake/modules/`.

Author: Alexander Neundorf <neundorf@kde.org>

FindLAPACK

Find Linear Algebra PACKage (LAPACK) library

This module finds an installed Fortran library that implements the *LAPACK linear-algebra interface*.

At least one of the **C**, **CXX**, or **Fortran** languages must be enabled.

Input Variables

The following variables may be set to influence this module's behavior:

BLA_STATIC

if **ON** use static linkage

BLA_VENDOR

Set to one of the *BLAS/LAPACK Vendors* to search for BLAS only from the specified vendor. If not set, all vendors are considered.

BLA_F95

if **ON** tries to find the BLAS95/LAPACK95 interfaces

BLA_PREFER_PKGCONFIG

Added in version 3.20.

if set **pkg-config** will be used to search for a LAPACK library first and if one is found that is preferred

BLA_PKGCONFIG_LAPACK

Added in version 3.25.

If set, the **pkg-config** method will look for this module name instead of just **lapack**.

BLA_SIZEOF_INTEGER

Added in version 3.22.

Specify the BLAS/LAPACK library integer size:

- 4** Search for a BLAS/LAPACK with 32-bit integer interfaces.
- 8** Search for a BLAS/LAPACK with 64-bit integer interfaces.
- ANY** Search for any BLAS/LAPACK. Most likely, a BLAS/LAPACK with 32-bit integer interfaces will be found.

Imported Targets

This module defines the following *IMPORTED* targets:

LAPACK::LAPACK

Added in version 3.18.

The libraries to use for LAPACK, if found.

Result Variables

This module defines the following variables:

LAPACK_FOUND

library implementing the LAPACK interface is found

LAPACK_LINKER_FLAGS

uncached list of required linker flags (excluding **-l** and **-L**).

LAPACK_LIBRARIES

uncached list of libraries (using full path name) to link against to use LAPACK

LAPACK95_LIBRARIES

uncached list of libraries (using full path name) to link against to use LAPACK95

LAPACK95_FOUND

library implementing the LAPACK95 interface is found

Intel MKL

To use the Intel MKL implementation of LAPACK, a project must enable at least one of the **C** or **CXX** languages. Set **BLA_VENDOR** to an Intel MKL variant either on the command-line as **-DBLA_VENDOR=Intel10_64lp** or in project code:

```
set(BLA_VENDOR Intel10_64lp)
find_package(LAPACK)
```

In order to build a project using Intel MKL, and end user must first establish an Intel MKL environment. See the *FindBLAS* module section on *Intel MKL* for details.

FindLATEX

Find LaTeX

This module finds an installed LaTeX and determines the location of the compiler. Additionally the module looks for Latex-related software like BibTeX.

Added in version 3.2: Component processing; support for hlatex, pdftops, Biber, xindy, XeLaTeX, LuaLaTeX.

This module sets the following result variables:

```
LATEX_FOUND:           whether found Latex and requested components
LATEX_<component>_FOUND: whether found <component>
LATEX_COMPILER:        path to the LaTeX compiler
PDFLATEX_COMPILER:     path to the PdfLaTeX compiler
XELATEX_COMPILER:      path to the XeLaTeX compiler
```

```

LUALATEX_COMPILER:  path to the LuaLaTeX compiler
BIBTEX_COMPILER:    path to the BibTeX compiler
BIBER_COMPILER:     path to the Biber compiler
MAKEINDEX_COMPILER: path to the MakeIndex compiler
XINDY_COMPILER:     path to the xindy compiler
DVIPS_CONVERTER:    path to the DVIPS converter
DVIPDF_CONVERTER:   path to the DVIPDF converter
PS2PDF_CONVERTER:   path to the PS2PDF converter
PDFTOPS_CONVERTER:  path to the pdftops converter
LATEX2HTML_CONVERTER: path to the LaTeX2Html converter
HTLATEX_COMPILER:   path to the htlatex compiler

```

Possible components are:

```

PDFLATEX
XELATEX
LUALATEX
BIBTEX
BIBER
MAKEINDEX
XINDY
DVIPS
DVIPDF
PS2PDF
PDFTOPS
LATEX2HTML
HTLATEX

```

Example Usages:

```

find_package(LATEX)
find_package(LATEX COMPONENTS PDFLATEX)
find_package(LATEX COMPONENTS BIBTEX PS2PDF)

```

FindLibArchive

Find libarchive library and headers. Libarchive is multi-format archive and compression library.

The module defines the following variables:

```

LibArchive_FOUND      - true if libarchive was found
LibArchive_INCLUDE_DIRS - include search path
LibArchive_LIBRARIES   - libraries to link
LibArchive_VERSION     - libarchive 3-component version number

```

The module defines the following **IMPORTED** targets:

```

LibArchive::LibArchive - target for linking against libarchive

```

Added in version 3.6: Support for new libarchive 3.2 version string format.

Added in version 3.17: Provides an imported target.

FindLibinput

Added in version 3.14.

Find libinput headers and library.

Imported Targets**Libinput::Libinput**

The libinput library, if found.

Result Variables

This will define the following variables in your project:

Libinput_FOUND

true if (the requested version of) libinput is available.

Libinput_VERSION

the version of libinput.

Libinput_LIBRARIES

the libraries to link against to use libinput.

Libinput_INCLUDE_DIRS

where to find the libinput headers.

Libinput_COMPILE_OPTIONS

this should be passed to `target_compile_options()`, if the target is not used for linking

FindLibLZMA

Find LZMA compression algorithm headers and library.

Imported Targets

Added in version 3.14.

This module defines *IMPORTED* target **LibLZMA::LibLZMA**, if liblzma has been found.

Result variables

This module will set the following variables in your project:

LIBLZMA_FOUND

True if liblzma headers and library were found.

LIBLZMA_INCLUDE_DIRS

Directory where liblzma headers are located.

LIBLZMA_LIBRARIES

Lzma libraries to link against.

LIBLZMA_HAS_AUTO_DECODER

True if `lzma_auto_decoder()` is found (required).

LIBLZMA_HAS_EASY_ENCODER

True if `lzma_easy_encoder()` is found (required).

LIBLZMA_HAS_LZMA_PRESET

True if `lzma_lzma_preset()` is found (required).

LIBLZMA_VERSION

Added in version 3.26: the version of LZMA found.

See also legacy variable **LIBLZMA_VERSION_STRING**.

Legacy Variables

The following variables are provided for backward compatibility:

LIBLZMA_VERSION_MAJOR

The major version of lzma

LIBLZMA_VERSION_MINOR

The minor version of lzma

LIBLZMA_VERSION_PATCH

The patch version of lzma

LIBLZMA_VERSION_STRING

version number as a string (ex: "5.0.3")

Changed in version 3.26: Superseded by **LIBLZMA_VERSION**.

FindLibXml2

Find the XML processing library (libxml2).

Imported Targets

Added in version 3.12.

The following *IMPORTED* targets may be defined:

LibXml2::LibXml2

libxml2 library.

LibXml2::xmllint

Added in version 3.17.

xmllint command-line executable.

Result variables

This module will set the following variables in your project:

LibXml2_FOUND

true if libxml2 headers and libraries were found

LIBXML2_INCLUDE_DIR

the directory containing LibXml2 headers

LIBXML2_INCLUDE_DIRS

list of the include directories needed to use LibXml2

LIBXML2_LIBRARIES

LibXml2 libraries to be linked

LIBXML2_DEFINITIONS

the compiler switches required for using LibXml2

LIBXML2_XMLLINT_EXECUTABLE

path to the XML checking tool xmllint coming with LibXml2

LIBXML2_VERSION_STRING

the version of LibXml2 found (since CMake 2.8.8)

Cache variables

The following cache variables may also be set:

LIBXML2_INCLUDE_DIR

the directory containing LibXml2 headers

LIBXML2_LIBRARY

path to the LibXml2 library

FindLibXslt

Find the XSL Transformations, Extensible Stylesheet Language Transformations (XSLT) library (LibXslt)

Imported Targets

Added in version 3.18.

The following *IMPORTED* targets may be defined:

LibXslt::LibXslt

If the libxslt library has been found

LibXslt::LibExslt

If the libexslt library has been found

LibXslt::xsltproc

If the xsltproc command-line executable has been found

Result variables

This module will set the following variables in your project:

LIBXSLT_FOUND – system has LibXslt
 LIBXSLT_INCLUDE_DIR – the LibXslt include directory
 LIBXSLT_LIBRARIES – Link these to LibXslt
 LIBXSLT_DEFINITIONS – Compiler switches required for using LibXslt
 LIBXSLT_VERSION_STRING – version of LibXslt found (since CMake 2.8.8)

Additionally, the following two variables are set (but not required for using xslt):

LIBXSLT_EXSLT_INCLUDE_DIR

Added in version 3.18: The include directory for exslt.

LIBXSLT_EXSLT_LIBRARIES

Link to these if you need to link against the exslt library.

LIBXSLT_XSLTPROC_EXECUTABLE

Contains the full path to the xsltproc executable if found.

FindLTTngUST

Added in version 3.6.

Find *Linux Trace Toolkit Next Generation (LTTng-UST)* library.

Imported target

This module defines the following *IMPORTED* target:

LTTng::UST

The LTTng-UST library, if found

Result variables

This module sets the following

LTTNGUST_FOUND

TRUE if system has LTTng-UST

LTTNGUST_INCLUDE_DIRS

The LTTng-UST include directories

LTTNGUST_LIBRARIES

The libraries needed to use LTTng-UST

LTTNGUST_VERSION_STRING

The LTTng-UST version

LTTNGUST_HAS_TRACEF

TRUE if the **tracef()** API is available in the system's LTTng-UST

LTTNGUST_HAS_TRACELOG

TRUE if the **tracelog()** API is available in the system's LTTng-UST

FindLua

Locate Lua library.

Added in version 3.18: Support for Lua 5.4.

This module defines:

LUA_FOUND

if false, do not try to link to Lua

LUA_LIBRARIES

both lua and lua5.4

LUA_INCLUDE_DIR

where to find lua.h

LUA_VERSION_STRING

the version of Lua found

LUA_VERSION_MAJOR

the major version of Lua

LUA_VERSION_MINOR

the minor version of Lua

LUA_VERSION_PATCH

the patch version of Lua

Note that the expected include convention is

```
#include "lua.h"
```

and not

```
#include <lua/lua.h>
```

This is because, the lua location is not standardized and may exist in locations other than lua/

FindLua50

Locate Lua library.

This module defines:

LUA50_FOUND, if false, do not try to link to Lua

LUA_LIBRARIES, both lua and lua5.4

LUA_INCLUDE_DIR, where to find lua.h and lua5.4.h (and probably lua5.4lib.h)

Note that the expected include convention is:

```
#include "lua.h"
```

and not:

```
#include <lua/lua.h>
```

This is because, the lua location is not standardized and may exist in locations other than lua/

FindLua51

Locate Lua library. This module defines:

```
LUA51_FOUND, if false, do not try to link to Lua
LUA_LIBRARIES
LUA_INCLUDE_DIR, where to find lua.h
LUA_VERSION_STRING, the version of Lua found (since CMake 2.8.8)
```

Note that the expected include convention is:

```
#include "lua.h"
```

and not:

```
#include <lua/lua.h>
```

This is because, the lua location is not standardized and may exist in locations other than lua/

FindMatlab

Finds Matlab or Matlab Compiler Runtime (MCR) and provides Matlab tools, libraries and compilers to CMake.

This package primary purpose is to find the libraries associated with Matlab or the MCR in order to be able to build Matlab extensions (mex files). It can also be used:

- to run specific commands in Matlab in case Matlab is available
- for declaring Matlab unit test
- to retrieve various information from Matlab (mex extensions, versions and release queries, ...)

Added in version 3.12: Added Matlab Compiler Runtime (MCR) support.

The module supports the following components:

- **ENG_LIBRARY** and **MAT_LIBRARY**: respectively the **ENG** and **MAT** libraries of Matlab
- **MAIN_PROGRAM** the Matlab binary program. Note that this component is not available on the MCR version, and will yield an error if the MCR is found instead of the regular Matlab installation.
- **MEX_COMPILER** the MEX compiler.
- **MCC_COMPILER** the MCC compiler, included with the Matlab Compiler add-on.
- **SIMULINK** the Simulink environment.

Added in version 3.7: Added the **MAT_LIBRARY** component.

Added in version 3.13: Added the **ENGINE_LIBRARY**, **DATAARRAY_LIBRARY** and **MCC_COMPILER** components.

Changed in version 3.14: Removed the **MX_LIBRARY**, **ENGINE_LIBRARY** and **DATAARRAY_LIBRARY** components. These libraries are found unconditionally.

Added in version 3.30: Added support for specifying a version range to *find_package()* and added support for specifying **REGISTRY_VIEW** to *find_package()*, *matlab_extract_all_installed_versions_from_registry()* and *matlab_get_all_valid_matlab_roots_from_registry()*. The default behavior remained unchanged, by using the registry view **TARGET**.

NOTE:

The version given to the *find_package()* directive is the Matlab **version**, which should not be confused with the Matlab *release* name (e.g. *R2023b*). The *matlab_get_version_from_release_name()* and *matlab_get_release_name_from_version()* provide a mapping between the release name and the version.

The variable *Matlab_ROOT_DIR* may be specified in order to give the path of the desired Matlab version. Otherwise, the behavior is platform specific:

- Windows: The installed versions of Matlab/MCR are retrieved from the Windows registry. The **REGISTRY_VIEW** argument may optionally be specified to manually control whether 32bit or 64bit versions shall be searched for.
- macOS: The installed versions of Matlab/MCR are given by the MATLAB default installation paths under **\$HOME/Applications** and **/Applications**. If no such application is found, it falls back to the one that might be accessible from the **PATH**.
- Unix: The desired Matlab should be accessible from the **PATH**. This does not work for MCR installation and *Matlab_ROOT_DIR* should be specified on this platform.

Additional information is provided when *MATLAB_FIND_DEBUG* is set. When a Matlab/MCR installation is found automatically and the **MATLAB_VERSION** is not given, the version is queried from Matlab directly (on Windows this may pop up a Matlab window) or from the MCR installation.

The mapping of the release names and the version of Matlab is performed by defining pairs (name, version). The variable *MATLAB_ADDITIONAL_VERSIONS* may be provided before the call to the *find_package()* in order to handle additional versions.

A Matlab scripts can be added to the set of tests using the *matlab_add_unit_test()*. By default, the Matlab unit test framework will be used ($\geq 2013a$) to run this script, but regular **.m** files returning an exit code can be used as well (0 indicating a success).

Module Input Variables

Users or projects may set the following variables to configure the module behavior:

Matlab_ROOT

Added in version 3.25.

Default value for *Matlab_ROOT_DIR*, the root of the Matlab installation.

Matlab_ROOT_DIR

The root of the Matlab installation.

MATLAB_FIND_DEBUG

outputs debug information

MATLAB_ADDITIONAL_VERSIONS

additional versions of Matlab for the automatic retrieval of the installed versions.

Imported Targets

Added in version 3.22.

This module defines the following *IMPORTED* targets:

Matlab::mex

The **mex** library, always available for MATLAB installations. Available for MCR installations if provided by MCR.

Matlab::mx

The **mx** library of Matlab (arrays), always available for MATLAB installations. Available for MCR installations if provided by MCR.

Matlab::eng

Matlab engine library. Available only if the **ENG_LIBRARY** component is requested.

Matlab::mat

Matlab matrix library. Available only if the **MAT_LIBRARY** component is requested.

Matlab::MatlabEngine

Matlab C++ engine library, always available for MATLAB R2018a and newer. Available for MCR installations if provided by MCR.

Matlab::MatlabDataArray

Matlab C++ data array library, always available for MATLAB R2018a and newer. Available for MCR installations if provided by MCR.

Variables defined by the module**Result variables****Matlab_FOUND**

TRUE if the Matlab installation is found, **FALSE** otherwise. All variable below are defined if Matlab is found.

Matlab_VERSION

Added in version 3.27.

the numerical version (e.g. 23.2.0) of Matlab found. Not to be confused with Matlab release name (e.g. R2023b) that can be obtained with *matlab_get_release_name_from_version()*.

Matlab_ROOT_DIR

the final root of the Matlab installation determined by the FindMatlab module.

Matlab_MAIN_PROGRAM

the Matlab binary program. Available only if the component **MAIN_PROGRAM** is given in the *find_package()* directive.

Matlab_INCLUDE_DIRS

the path of the Matlab libraries headers

Matlab_MEX_LIBRARY

library for mex, always available for MATLAB installations. Available for MCR installations if provided by MCR.

Matlab_MX_LIBRARY

mx library of Matlab (arrays), always available for MATLAB installations. Available for MCR installations if provided by MCR.

Matlab_ENG_LIBRARY

Matlab engine library. Available only if the component **ENG_LIBRARY** is requested.

Matlab_MAT_LIBRARY

Matlab matrix library. Available only if the component **MAT_LIBRARY** is requested.

Matlab_ENGINE_LIBRARY

Added in version 3.13.

Matlab C++ engine library, always available for MATLAB R2018a and newer. Available for MCR installations if provided by MCR.

Matlab_DATAARRAY_LIBRARY

Added in version 3.13.

Matlab C++ data array library, always available for MATLAB R2018a and newer. Available for MCR installations if provided by MCR.

Matlab_LIBRARIES

the whole set of libraries of Matlab

Matlab_MEX_COMPILER

the mex compiler of Matlab. Currently not used. Available only if the component **MEX_COMPILER** is requested.

Matlab_MCC_COMPILER

Added in version 3.13.

the mcc compiler of Matlab. Included with the Matlab Compiler add-on. Available only if the component **MCC_COMPILER** is requested.

Cached variables**Matlab_MEX_EXTENSION**

the extension of the mex files for the current platform (given by Matlab).

Matlab_ROOT_DIR

the location of the root of the Matlab installation found. If this value is changed by the user, the result variables are recomputed.

Provided commands

matlab_get_version_from_release_name()

returns the version from the Matlab release name

matlab_get_release_name_from_version()

returns the release name from the Matlab version

matlab_add_mex()

adds a target compiling a MEX file.

matlab_add_unit_test()

adds a Matlab unit test file as a test to the project.

matlab_extract_all_installed_versions_from_registry()

parses the registry for all Matlab versions. Available on Windows only. The part of the registry parsed is dependent on the host processor

matlab_get_all_valid_matlab_roots_from_registry()

returns all the possible Matlab or MCR paths, according to a previously given list. Only the existing/accessible paths are kept. This is mainly useful for the searching all possible Matlab installation.

matlab_get_mex_suffix()

returns the suffix to be used for the mex files (platform/architecture dependent)

matlab_get_version_from_matlab_run()

returns the version of Matlab/MCR, given the full directory of the Matlab/MCR installation path.

Known issues

Symbol clash in a MEX target

By default, every symbols inside a MEX file defined with the command *matlab_add_mex()* have hidden visibility, except for the entry point. This is the default behavior of the MEX compiler, which lowers the risk of symbol collision between the libraries shipped with Matlab, and the libraries to which the MEX file is linking to. This is also the default on Windows platforms.

However, this is not sufficient in certain case, where for instance your MEX file is linking against libraries that are already loaded by Matlab, even if those libraries have different SONAMES. A possible solution is to hide the symbols of the libraries to which the MEX target is linking to. This can be achieved in GNU GCC compilers with the linker option **-Wl,--exclude-libs,ALL**.

Tests using GPU resources

in case your MEX file is using the GPU and in order to be able to run unit tests on this MEX file, the GPU resources should be properly released by Matlab. A possible solution is to make Matlab aware of the use of the GPU resources in the session, which can be performed by a command such as **D = gpuDevice()** at the beginning of the test script (or via a fixture).

Reference

Matlab_ROOT_DIR

The root folder of the Matlab installation. If set before the call to *find_package()*, the module will look for the components in that path. If not set, then an automatic search of Matlab will be performed. If set, it should point to a valid version of Matlab.

MATLAB_FIND_DEBUG

If set, the lookup of Matlab and the intermediate configuration steps are outputted to the console.

MATLAB_ADDITIONAL_VERSIONS

If set, specifies additional versions of Matlab that may be looked for. The variable should be a list of strings, organized by pairs of release name and versions, such as follows:

```
set(MATLAB_ADDITIONAL_VERSIONS
    "release_name1=corresponding_version1"
    "release_name2=corresponding_version2"
    ...
)
```

Example:

```
set(MATLAB_ADDITIONAL_VERSIONS
    "R2013b=8.2"
    "R2013a=8.1"
    "R2012b=8.0")
```

The order of entries in this list matters when several versions of Matlab are installed. The priority is set according to the ordering in this list.

matlab_get_version_from_release_name

```
matlab_get_version_from_release_name(release version)
```

- Input: **release** is the release name (e.g. R2023b)
- Output: **version** is the version of Matlab (e.g. 23.2.0)

Returns the version of Matlab from a release name

NOTE:

This command provides correct versions mappings for Matlab but not MCR.

matlab_get_release_name_from_version

```
matlab_get_release_name_from_version(version release_name)
```

- Input: **version** is the version of Matlab (e.g. 23.2.0)
- Output: **release_name** is the release name (R2023b)

Returns the release name from the version of Matlab

NOTE:

This command provides correct version mappings for Matlab but not MCR.

matlab_extract_all_installed_versions_from_registry

This function parses the Windows registry and finds the Matlab versions that are installed. The found versions are stored in a given **<versions-var>**.

matlab_extract_all_installed_versions_from_registry(<versions-var> [REGISTRY_VIEW view])

Added in version 3.30.

- Output: **<versions-var>** is a list of all the versions of Matlab found
- Input: **REGISTRY_VIEW** Optional registry view to use for registry interaction. The argument is passed (or omitted) to *cmake_host_system_information()* without further checks or modification.

matlab_extract_all_installed_versions_from_registry(<win64> <versions-var>)

- Input: **win64** is a boolean to search for the 64 bit version of Matlab. Set to **ON** to use the 64bit registry view or **OFF** to use the 32bit registry view. If finer control is needed, see signature above.
- Output: **<versions-var>** is a list of all the versions of Matlab found

The returned list contains all versions under **HKLM\SOFTWARE\Mathworks\MATLAB**, **HKLM\SOFTWARE\Mathworks\MATLAB Runtime** and **HKLM\SOFTWARE\Mathworks\MATLAB Compiler Runtime** or an empty list in case an error occurred (or nothing found).

NOTE:

Only the versions are provided. No check is made over the existence of the installation referenced in the registry,

matlab_get_all_valid_matlab_roots_from_registry

Populates the Matlab root with valid versions of Matlab or Matlab Runtime (MCR). The returned **matlab_roots** is organized in triplets (**type,version_number,matlab_root_path**), where **type** indicates either **MATLAB** or **MCR**.

```
matlab_get_all_valid_matlab_roots_from_registry(matlab_versions matlab_root
```

- Input: **matlab_versions** of each of the Matlab or MCR installations
- Output: **matlab_roots** location of each of the Matlab or MCR installations
- Input: **REGISTRY_VIEW** Optional registry view to use for registry interaction. The argument is passed (or omitted) to *cmake_host_system_information()* without further checks or modification.

Added in version 3.30: The optional **REGISTRY_VIEW** argument was added to provide a more precise interface on how to interact with the Windows Registry.

matlab_get_mex_suffix

Returns the extension of the mex files (the suffixes). This function should not be called before the appropriate Matlab root has been found.

```
matlab_get_mex_suffix(matlab_root mex_suffix)
```

- Input: **matlab_root** root of Matlab/MCR install e.g. **Matlab_ROOT_DIR**
- Output: **mex_suffix** variable name in which the suffix will be returned.

matlab_get_version_from_matlab_run

This function runs Matlab program specified on arguments and extracts its version. If the path provided for the Matlab installation points to an MCR installation, the version is extracted from the installed files.

```
matlab_get_version_from_matlab_run(matlab_binary_path matlab_list_versions)
```

- Input: **matlab_binary_path** path of the *matlab* binary executable
- Output: **matlab_list_versions** the version extracted from Matlab

matlab_add_unit_test

Adds a Matlab unit test to the test set of cmake/ctest. This command requires the component **MAIN_PROGRAM** and hence is not available for an MCR installation.

The unit test uses the Matlab unittest framework (default, available starting Matlab 2013b+) except if the option **NO_UNITTEST_FRAMEWORK** is given.

The function expects one Matlab test script file to be given. In the case **NO_UNITTEST_FRAMEWORK** is given, the unittest script file should contain the script to be run, plus an exit command with the exit value. This exit value will be passed to the ctest framework (0 success, non 0 failure). Additional arguments accepted by *add_test()* can be passed through **TEST_ARGS** (eg. **CONFIGURATION <config> ...**).

```
matlab_add_unit_test(
    NAME <name>
    UNITTEST_FILE matlab_file_containing_unittest.m
    [CUSTOM_TEST_COMMAND matlab_command_to_run_as_test]
    [UNITTEST_PRECOMMAND matlab_command_to_run]
    [TIMEOUT timeout]
    [ADDITIONAL_PATH path1 [path2 ...]]
    [MATLAB_ADDITIONAL_STARTUP_OPTIONS option1 [option2 ...]]
    [TEST_ARGS arg1 [arg2 ...]]
    [NO_UNITTEST_FRAMEWORK]
)
```

Function Parameters:

NAME name of the unittest in ctest.

UNITTEST_FILE

the matlab unittest file. Its path will be automatically added to the Matlab path.

CUSTOM_TEST_COMMAND

Matlab script command to run as the test. If this is not set, then the following is run:
runtests('matlab_file_name'), exit(max([ans(1,:).Failed])) where **matlab_file_name** is the **UNITTEST_FILE** without the extension.

UNITTEST_PRECOMMAND

Matlab script command to be ran before the file containing the test (eg. GPU device initialization based on CMake variables).

TIMEOUT

the test timeout in seconds. Defaults to 180 seconds as the Matlab unit test may hang.

ADDITIONAL_PATH

a list of paths to add to the Matlab path prior to running the unit test.

MATLAB_ADDITIONAL_STARTUP_OPTIONS

a list of additional option in order to run Matlab from the command line. **-nosplash**
-nodesktop **-nodisplay** are always added.

TEST_ARGS

Additional options provided to the `add_test` command. These options are added to the default options (eg. "CONFIGURATIONS Release")

NO_UNITTEST_FRAMEWORK

when set, indicates that the test should not use the unittest framework of Matlab (available for versions $\geq$ R2013a).

WORKING_DIRECTORY

This will be the working directory for the test. If specified it will also be the output directory used for the log file of the test run. If not specified the temporary directory **\${CMAKE_BINARY_DIR}/Matlab** will be used as the working directory and the log location.

matlab_add_mex

Adds a Matlab MEX target. This commands compiles the given sources with the current tool-chain in order to produce a MEX file. The final name of the produced output may be specified, as well as additional link libraries, and a documentation entry for the MEX file. Remaining arguments of the call are passed to the `add_library()` or `add_executable()` command.

```
matlab_add_mex(
    NAME <name>
    [EXECUTABLE | MODULE | SHARED]
    SRC src1 [src2 ...]
    [OUTPUT_NAME output_name]
    [DOCUMENTATION file.txt]
    [LINK_TO target1 target2 ...]
    [R2017b | R2018a]
    [EXCLUDE_FROM_ALL]
    [NO_IMPLICIT_LINK_TO_MATLAB_LIBRARIES]
    [...]
)
```

Function Parameters:

NAME name of the target.

SRC list of source files.

LINK_TO

a list of additional link dependencies. The target links to **libmex** and **libmx** by default, unless the **NO_IMPLICIT_LINK_TO_MATLAB_LIBRARIES** option is passed.

OUTPUT_NAME

if given, overrides the default name. The default name is the name of the target without any prefix and with **Matlab_MEX_EXTENSION** suffix.

DOCUMENTATION

if given, the file **file.txt** will be considered as being the documentation file for the MEX file. This file is copied into the same folder without any processing, with the same name as the final mex file, and with extension **.m**. In that case, typing **help <name>** in Matlab prints the documentation contained in this file.

R2017b or R2018a

Added in version 3.14.

May be given to specify the version of the C API to use: **R2017b** specifies the traditional (separate complex) C API, and corresponds to the **-R2017b** flag for the *mex* command. **R2018a** specifies the new interleaved complex C API, and corresponds to the **-R2018a** flag for the *mex* command. Ignored if MATLAB version prior to R2018a. Defaults to **R2017b**.

MODULE or SHARED

Added in version 3.7.

May be given to specify the type of library to be created.

EXECUTABLE

Added in version 3.7.

May be given to create an executable instead of a library. If no type is given explicitly, the type is **SHARED**.

EXCLUDE_FROM_ALL

This option has the same meaning as for *EXCLUDE_FROM_ALL* and is forwarded to *add_library()* or *add_executable()* commands.

NO_IMPLICIT_LINK_TO_MATLAB_LIBRARIES

Added in version 3.24.

This option permits to disable the automatic linking of MATLAB libraries, so that only the libraries that are actually required can be linked via the **LINK_TO** option.

The documentation file is not processed and should be in the following format:

```
% This is the documentation
function ret = mex_target_output_name(input1)
```

FindMFC

Find Microsoft Foundation Class Library (MFC) on Windows

Find the native MFC – i.e. decide if an application can link to the MFC libraries.

MFC_FOUND – Was MFC support found

You don't need to include anything or link anything to use it.

FindMotif

Try to find Motif (or lesstif)

Once done this will define:

```
MOTIF_FOUND      - system has MOTIF
MOTIF_INCLUDE_DIR - include paths to use Motif
MOTIF_LIBRARIES   - Link these to use Motif
```

FindMPEG

Find the native MPEG includes and library

This module defines

```
MPEG_INCLUDE_DIR, where to find MPEG.h, etc.
MPEG_LIBRARIES, the libraries required to use MPEG.
MPEG_FOUND, If false, do not try to use MPEG.
```

also defined, but not for general use are

```
MPEG_mpeg2_LIBRARY, where to find the MPEG library.
MPEG_vo_LIBRARY, where to find the vo library.
```

FindMPEG2

Find the native MPEG2 includes and library

This module defines

```
MPEG2_INCLUDE_DIR, path to mpeg2dec/mpeg2.h, etc.
MPEG2_LIBRARIES, the libraries required to use MPEG2.
MPEG2_FOUND, If false, do not try to use MPEG2.
```

also defined, but not for general use are

```
MPEG2_mpeg2_LIBRARY, where to find the MPEG2 library.
MPEG2_vo_LIBRARY, where to find the vo library.
```

FindMPI

Find a Message Passing Interface (MPI) implementation.

The Message Passing Interface (MPI) is a library used to write high-performance distributed-memory parallel applications, and is typically deployed on a cluster. MPI is a standard interface (defined by the MPI forum) for which many implementations are available.

Added in version 3.10: Major overhaul of the module: many new variables, per-language components, support for a wider variety of runtimes.

Variables for using MPI

The module exposes the components **C**, **CXX**, **MPICXX** and **Fortran**. Each of these controls the various MPI languages to search for. The difference between **CXX** and **MPICXX** is that **CXX** refers to the MPI C

API being usable from C++, whereas **MPICXX** refers to the MPI-2 C++ API that was removed again in MPI-3.

Depending on the enabled components the following variables will be set:

MPI_FOUND

Variable indicating that MPI settings for all requested languages have been found. If no components are specified, this is true if MPI settings for all enabled languages were detected. Note that the **MPICXX** component does not affect this variable.

MPI_VERSION

Minimal version of MPI detected among the requested languages, or all enabled languages if no components were specified.

This module will set the following variables per language in your project, where **<lang>** is one of C, CXX, or Fortran:

MPI_<lang>_FOUND

Variable indicating the MPI settings for **<lang>** were found and that simple MPI test programs compile with the provided settings.

MPI_<lang>_COMPILER

MPI compiler for **<lang>** if such a program exists.

MPI_<lang>_COMPILE_OPTIONS

Compilation options for MPI programs in **<lang>**, given as a *;-list*.

MPI_<lang>_COMPILE_DEFINITIONS

Compilation definitions for MPI programs in **<lang>**, given as a *;-list*.

MPI_<lang>_INCLUDE_DIRS

Include path(s) for MPI header.

MPI_<lang>_LINK_FLAGS

Linker flags for MPI programs.

MPI_<lang>_LIBRARIES

All libraries to link MPI programs against.

Added in version 3.9: Additionally, the following *IMPORTED* targets are defined:

MPI::MPI_<lang>

Target for using MPI from **<lang>**.

The following variables indicating which bindings are present will be defined:

MPI_MPICXX_FOUND

Variable indicating whether the MPI-2 C++ bindings are present (introduced in MPI-2, removed with MPI-3).

MPI_Fortran_HAVE_F77_HEADER

True if the Fortran 77 header **mpif.h** is available.

MPI_Fortran_HAVE_F90_MODULE

True if the Fortran 90 module **mpi** can be used for accessing MPI (MPI-2 and higher only).

MPI_Fortran_HAVE_F08_MODULE

True if the Fortran 2008 **mpi_f08** is available to MPI programs (MPI-3 and higher only).

If possible, the MPI version will be determined by this module. The facilities to detect the MPI version were introduced with MPI-1.2, and therefore cannot be found for older MPI versions.

MPI_<lang>_VERSION_MAJOR

Major version of MPI implemented for <lang> by the MPI distribution.

MPI_<lang>_VERSION_MINOR

Minor version of MPI implemented for <lang> by the MPI distribution.

MPI_<lang>_VERSION

MPI version implemented for <lang> by the MPI distribution.

Note that there's no variable for the C bindings being accessible through **mpi.h**, since the MPI standards always have required this binding to work in both C and C++ code.

For running MPI programs, the module sets the following variables

MPIEXEC_EXECUTABLE

Executable for running MPI programs, if such exists.

MPIEXEC_NUMPROC_FLAG

Flag to pass to **mpiexec** before giving it the number of processors to run on.

MPIEXEC_MAX_NUMPROCS

Number of MPI processors to utilize. Defaults to the number of processors detected on the host system.

MPIEXEC_PREFLAGS

Flags to pass to **mpiexec** directly before the executable to run.

MPIEXEC_POSTFLAGS

Flags to pass to **mpiexec** after other flags.

Variables for locating MPI

This module performs a four step search for an MPI implementation:

1. Search for **MPIEXEC_EXECUTABLE** and, if found, use its base directory.
2. Check if the compiler has MPI support built-in. This is the case if the user passed a compiler wrapper as **CMAKE_<LANG>_COMPILER** or if they use Cray system compiler wrappers.
3. Attempt to find an MPI compiler wrapper and determine the compiler information from it.
4. Try to find an MPI implementation that does not ship such a wrapper by guessing settings. Currently, only Microsoft MPI and MPICH2 on Windows are supported.

For controlling the **MPIEXEC_EXECUTABLE** step, the following variables may be set:

MPIEXEC_EXECUTABLE

Manually specify the location of **mpiexec**.

MPI_HOME

Specify the base directory of the MPI installation.

ENV{MPI_HOME}

Environment variable to specify the base directory of the MPI installation.

ENV{I_MPI_ROOT}

Environment variable to specify the base directory of the MPI installation.

For controlling the compiler wrapper step, the following variables may be set:

MPI_<lang>_COMPILER

Search for the specified compiler wrapper and use it.

MPI_<lang>_COMPILER_FLAGS

Flags to pass to the MPI compiler wrapper during interrogation. Some compiler wrappers support linking debug or tracing libraries if a specific flag is passed and this variable may be used to obtain

them.

MPI_COMPILER_FLAGS

Used to initialize **MPI_<lang>_COMPILER_FLAGS** if no language specific flag has been given. Empty by default.

MPI_EXECUTABLE_SUFFIX

A suffix which is appended to all names that are being looked for. For instance you may set this to **.mpich** or **.openmpi** to prefer the one or the other on Debian and its derivatives.

In order to control the guessing step, the following variable may be set:

MPI_GUESS_LIBRARY_NAME

Valid values are **MSMPI** and **MPICH2**. If set, only the given library will be searched for. By default, **MSMPI** will be preferred over **MPICH2** if both are available. This also sets **MPI_SKIP_COMPILER_WRAPPER** to **true**, which may be overridden.

Each of the search steps may be skipped with the following control variables:

MPI_ASSUME_NO_BUILTIN_MPI

If true, the module assumes that the compiler itself does not provide an MPI implementation and skips to step 2.

MPI_SKIP_COMPILER_WRAPPER

If true, no compiler wrapper will be searched for.

MPI_SKIP_GUESSING

If true, the guessing step will be skipped.

Additionally, the following control variable is available to change search behavior:

MPI_CXX_SKIP_MPICXX

Add some definitions that will disable the MPI-2 C++ bindings. Currently supported are MPICH, Open MPI, Platform MPI and derivatives thereof, for example MVAPICH or Intel MPI.

If the find procedure fails for a variable **MPI_<lang>_WORKS**, then the settings detected by or passed to the module did not work and even a simple MPI test program failed to compile.

If all of these parameters were not sufficient to find the right MPI implementation, a user may disable the entire autodetection process by specifying both a list of libraries in **MPI_<lang>_LIBRARIES** and a list of include directories in **MPI_<lang>_ADDITIONAL_INCLUDE_DIRS**. Any other variable may be set in addition to these two. The module will then validate the MPI settings and store the settings in the cache.

Cache variables for MPI

The variable **MPI_<lang>_INCLUDE_DIRS** will be assembled from the following variables. For C and CXX:

MPI_<lang>_HEADER_DIR

Location of the **mpi.h** header on disk.

For Fortran:

MPI_Fortran_F77_HEADER_DIR

Location of the Fortran 77 header **mpif.h**, if it exists.

MPI_Fortran_MODULE_DIR

Location of the **mpi** or **mpi_f08** modules, if available.

For all languages the following variables are additionally considered:

MPI_<lang>_ADDITIONAL_INCLUDE_DIRS

A ; -list of paths needed in addition to the normal include directories.

MPI_<include_name>_INCLUDE_DIR

Path variables for include folders referred to by <include_name>.

MPI_<lang>_ADDITIONAL_INCLUDE_VARS

A ; -list of <include_name> that will be added to the include locations of <lang>.

The variable **MPI_<lang>_LIBRARIES** will be assembled from the following variables:

MPI_<lib_name>_LIBRARY

The location of a library called <lib_name> for use with MPI.

MPI_<lang>_LIB_NAMES

A ; -list of <lib_name> that will be added to the include locations of <lang>.

Usage of mpiexec

When using **MPIEXEC_EXECUTABLE** to execute MPI applications, you should typically use all of the **MPIEXEC_EXECUTABLE** flags as follows:

```
{MPIEXEC_EXECUTABLE} {MPIEXEC_NUMPROC_FLAG} {MPIEXEC_MAX_NUMPROCS}
{MPIEXEC_PREFLAGS} EXECUTABLE {MPIEXEC_POSTFLAGS} ARGS
```

where **EXECUTABLE** is the MPI program, and **ARGS** are the arguments to pass to the MPI program.

Advanced variables for using MPI

The module can perform some advanced feature detections upon explicit request.

Important notice: The following checks cannot be performed without *executing* an MPI test program. Consider the special considerations for the behavior of *try_run()* during cross compilation. Moreover, running an MPI program can cause additional issues, like a firewall notification on some systems. You should only enable these detections if you absolutely need the information.

If the following variables are set to true, the respective search will be performed:

MPI_DETERMINE_Fortran_CAPABILITIES

Determine for all available Fortran bindings what the values of **MPI_SUBARRAYS_SUPPORTED** and **MPI_ASYNC_PROTECTS_NONBLOCKING** are and make their values available as **MPI_Fortran_<binding>_SUBARRAYS** and **MPI_Fortran_<binding>_ASYNCPROT**, where <binding> is one of **F77_HEADER**, **F90_MODULE** and **F08_MODULE**.

MPI_DETERMINE_LIBRARY_VERSION

For each language, find the output of **MPI_Get_library_version** and make it available as **MPI_<lang>_LIBRARY_VERSION_STRING**. This information is usually tied to the runtime component of an MPI implementation and might differ depending on <lang>. Note that the return value is entirely implementation defined. This information might be used to identify the MPI vendor and for example pick the correct one of multiple third party binaries that matches the MPI vendor.

Backward Compatibility

Deprecated since version 3.10.

For backward compatibility with older versions of FindMPI, these variables are set:

```
MPI_COMPILER      MPI_LIBRARY      MPI_EXTRA_LIBRARY
MPI_COMPILE_FLAGS MPI_INCLUDE_PATH  MPI_LINK_FLAGS
MPI_LIBRARIES
```

In new projects, please use the **MPI_<lang>_XXX** equivalents. Additionally, the following variables are deprecated:

MPI_<lang>_COMPILE_FLAGS

Use **MPI_<lang>_COMPILE_OPTIONS** and **MPI_<lang>_COMPILE_DEFINITIONS** instead.

MPI_<lang>_INCLUDE_PATH

For consumption use **MPI_<lang>_INCLUDE_DIRS** and for specifying folders use **MPI_<lang>_ADDITIONAL_INCLUDE_DIRS** instead.

MPIEXEC

Use **MPIEXEC_EXECUTABLE** instead.

FindMsys

Added in version 3.21.

Find MSYS, a POSIX-compatible environment that runs natively on Microsoft Windows

FindODBC

Added in version 3.12.

Find an Open Database Connectivity (ODBC) include directory and library.

On Windows, when building with Visual Studio, this module assumes the ODBC library is provided by the available Windows SDK.

On Unix, this module allows to search for ODBC library provided by unixODBC or iODBC implementations of ODBC API. This module reads hint about location of the config program:

ODBC_CONFIG

Location of odbc_config or iodbc-config program

Otherwise, this module tries to find the config program, first from unixODBC, then from iODBC. If no config program found, this module searches for ODBC header and library in list of known locations.

Imported Targets

This module defines the following *IMPORTED* targets:

ODBC::ODBC

Imported target for using the ODBC library, if found.

Result variables

ODBC_FOUND

Set to true if ODBC library found, otherwise false or undefined.

ODBC_INCLUDE_DIRS

Paths to include directories listed in one variable for use by ODBC client. May be empty on Windows, where the include directory corresponding to the expected Windows SDK is already available in the compilation environment.

ODBC_LIBRARIES

Paths to libraries to linked against to use ODBC. May just a library name on Windows, where the library directory corresponding to the expected Windows SDK is already available in the compilation environment.

ODBC_CONFIG

Path to unixODBC or iODBC config program, if found or specified.

Cache variables

For users who wish to edit and control the module behavior, this module reads hints about search locations from the following variables:

ODBC_INCLUDE_DIR

Path to ODBC include directory with **sql.h** header.

ODBC_LIBRARY

Path to ODBC library to be linked.

These variables should not be used directly by project code.

Limitations

On Windows, this module does not search for iODBC. On Unix, there is no way to prefer unixODBC over iODBC, or vice versa, other than providing the config program location using the **ODBC_CONFIG**. This module does not allow to search for a specific ODBC driver.

FindOpenACC

Added in version 3.10.

Detect OpenACC support by the compiler.

This module can be used to detect OpenACC support in a compiler. If the compiler supports OpenACC, the flags required to compile with OpenACC support are returned in variables for the different languages. Currently, only NVHPC, PGI, GNU and Cray compilers are supported.

Imported Targets

Added in version 3.16.

The module provides *IMPORTED* targets:

OpenACC::OpenACC_<lang>

Target for using OpenACC from <lang>.

Variables

The module defines the following variables:

OpenACC_FOUND

Added in version 3.25.

Variable indicating that OpenACC flags for at least one languages have been found.

This module will set the following variables per language in your project, where <lang> is one of C, CXX, or Fortran:

OpenACC_<lang>_FOUND

Variable indicating if OpenACC support for <lang> was detected.

OpenACC_<lang>_FLAGS

OpenACC compiler flags for <lang>, separated by spaces.

OpenACC_<lang>_OPTIONS

Added in version 3.16.

OpenACC compiler flags for <lang>, as a list. Suitable for usage with `target_compile_options` or `target_link_options`.

The module will also try to provide the OpenACC version variables:

OpenACC_<lang>_SPEC_DATE

Date of the OpenACC specification implemented by the <lang> compiler.

OpenACC_<lang>_VERSION_MAJOR

Major version of OpenACC implemented by the <lang> compiler.

OpenACC_<lang>_VERSION_MINOR

Minor version of OpenACC implemented by the <lang> compiler.

OpenACC_<lang>_VERSION

OpenACC version implemented by the <lang> compiler.

The specification date is formatted as given in the OpenACC standard: **yyyymm** where **yyyy** and **mm** represents the year and month of the OpenACC specification implemented by the <lang> compiler.

Input Variables

OpenACC_ACCEL_TARGET=<target> If set, will the correct target accelerator flag set to the <target> will be returned with OpenACC_<lang>_FLAGS.

FindOpenAL

Finds Open Audio Library (OpenAL).

Projects using this module should use **#include "al.h"** to include the OpenAL header file, **not #include <AL/al.h>**. The reason for this is that the latter is not entirely portable. Windows/Creative Labs does not by default put their headers in **AL/** and macOS uses the convention **<OpenAL/al.h>**.

Hints

Environment variable **\$OPENALDIR** can be used to set the prefix of OpenAL installation to be found.

By default on macOS, system framework is search first. In other words, OpenAL is searched in the following order:

1. System framework: **/System/Library/Frameworks**, whose priority can be changed via setting the **CMAKE_FIND_FRAMEWORK** variable.
2. Environment variable **\$OPENALDIR**.
3. System paths.
4. User-compiled framework: **~/Library/Frameworks**.
5. Manually compiled framework: **/Library/Frameworks**.
6. Add-on package: **/opt**.

Imported Targets

Added in version 3.25.

This module defines the *IMPORTED* target:

OpenAL::OpenAL

The OpenAL library, if found.

Result Variables

This module defines the following variables:

OPENAL_FOUND

If false, do not try to link to OpenAL

OPENAL_INCLUDE_DIR

OpenAL include directory

OPENAL_LIBRARY

Path to the OpenAL library

OPENAL_VERSION_STRING

Human-readable string containing the version of OpenAL

FindOpenCL

Added in version 3.1.

Finds Open Computing Language (OpenCL)

Added in version 3.10: Detection of OpenCL 2.1 and 2.2.

Imported Targets

Added in version 3.7.

This module defines *IMPORTED* target **OpenCL::OpenCL**, if OpenCL has been found.

Result Variables

This module defines the following variables:

OpenCL_FOUND	- True if OpenCL was found
OpenCL_INCLUDE_DIRS	- include directories for OpenCL
OpenCL_LIBRARIES	- link against this library to use OpenCL
OpenCL_VERSION_STRING	- Highest supported OpenCL version (eg. 1.2)
OpenCL_VERSION_MAJOR	- The major version of the OpenCL implementation
OpenCL_VERSION_MINOR	- The minor version of the OpenCL implementation

The module will also define two cache variables:

OpenCL_INCLUDE_DIR	- the OpenCL include directory
OpenCL_LIBRARY	- the path to the OpenCL library

FindOpenGL

FindModule for OpenGL and OpenGL Utility Library (GLU).

Changed in version 3.2: X11 is no longer added as a dependency on Unix/Linux systems.

Added in version 3.10: GLVND support on Linux. See the *Linux-specific* section below.

Optional COMPONENTS

Added in version 3.10.

This module respects several optional COMPONENTS:

EGL The EGL interface between OpenGL, OpenGL ES and the underlying windowing system.

GLX An extension to X that interfaces OpenGL, OpenGL ES with X window system.

OpenGL

The cross platform API for 3D graphics.

GLES2

Added in version 3.27.

A subset of OpenGL API for embedded systems with limited capabilities.

GLES3

Added in version 3.27.

A subset of OpenGL API for embedded systems with more capabilities.

Imported Targets

Added in version 3.8.

This module defines the *IMPORTED* targets:

OpenGL::GL

Defined to the platform-specific OpenGL libraries if the system has OpenGL.

OpenGL::GLU

Defined if the system has OpenGL Utility Library (GLU).

Added in version 3.10: Additionally, the following GLVND-specific library targets are defined:

OpenGL::OpenGL

Defined to libOpenGL if the system is GLVND-based.

OpenGL::GLX

Defined if the system has OpenGL Extension to the X Window System (GLX).

OpenGL::EGL

Defined if the system has EGL.

OpenGL::GLES2

Added in version 3.27.

Defined if the system has GLES2.

OpenGL::GLES3

Added in version 3.27.

Defined if the system has GLES3.

Result Variables

This module sets the following variables:

OPENGL_FOUND

True, if the system has OpenGL and all components are found.

OPENGL_XMESA_FOUND

True, if the system has XMESA.

OPENGL_GLU_FOUND

True, if the system has GLU.

OpenGL_OpenGL_FOUND

True, if the system has an OpenGL library.

OpenGL_GLX_FOUND

True, if the system has GLX.

OpenGL_EGL_FOUND

True, if the system has EGL.

OpenGL::GLES2

Defined if the system has GLES2.

OpenGL::GLES3

Defined if the system has GLES3.

OPENGL_INCLUDE_DIR

Path to the OpenGL include directory. The **OPENGL_INCLUDE_DIRS** variable is preferred.

OPENGL_EGL_INCLUDE_DIRS

Path to the EGL include directory.

OPENGL_LIBRARIES

Paths to the OpenGL library, windowing system libraries, and GLU libraries. On Linux, this assumes GLX and is never correct for EGL-based targets. Clients are encouraged to use the **OpenGL::*** import targets instead.

OPENGL_INCLUDE_DIRS

Added in version 3.29.

Paths to the OpenGL include directories.

Added in version 3.10: Variables for GLVND-specific libraries **OpenGL**, **EGL** and **GLX**.

Cache variables

The following cache variables may also be set:

OPENGL_egl_LIBRARY

Path to the EGL library.

OPENGL_glu_LIBRARY

Path to the GLU library.

OPENGL_glx_LIBRARY

Path to the GLVND 'GLX' library.

OPENGL_opengl_LIBRARY

Path to the GLVND 'OpenGL' library

OPENGL_gl_LIBRARY

Path to the OpenGL library. New code should prefer the **OpenGL::*** import targets.

OPENGL_gles2_LIBRARY

Added in version 3.27.

Path to the OpenGL GLES2 library.

OPENGL_gles3_LIBRARY

Added in version 3.27.

Path to the OpenGL GLES3 library.

OPENGL_GLU_INCLUDE_DIR

Added in version 3.29.

Path to the OpenGL GLU include directory.

Added in version 3.10: Variables for GLVND-specific libraries **OpenGL**, **EGL** and **GLX**.

Linux-specific

Some Linux systems utilize GLVND as a new ABI for OpenGL. GLVND separates context libraries from OpenGL itself; OpenGL lives in "libOpenGL", and contexts are defined in "libGLX" or "libEGL". GLVND is currently the only way to get OpenGL 3+ functionality via EGL in a manner portable across vendors. Projects may use GLVND explicitly with target **OpenGL::OpenGL** and either **OpenGL::GLX** or **OpenGL::EGL**.

Projects may use the **OpenGL::GL** target (or **OPENGL_LIBRARIES** variable) to use legacy GL interfaces. These will use the legacy GL library located by **OPENGL_gl_LIBRARY**, if available. If **OPENGL_gl_LIBRARY** is empty or not found and GLVND is available, the **OpenGL::GL** target will use GLVND **OpenGL::OpenGL** and **OpenGL::GLX** (and the **OPENGL_LIBRARIES** variable will use the corresponding libraries). Thus, for non-EGL-based Linux targets, the **OpenGL::GL** target is most portable.

A **OpenGL_GL_PREFERENCE** variable may be set to specify the preferred way to provide legacy GL interfaces in case multiple choices are available. The value may be one of:

GLVND

If the GLVND OpenGL and GLX libraries are available, prefer them. This forces **OPENGL_gl_LIBRARY** to be empty.

Changed in version 3.11: This is the default, unless policy *CMP0072* is set to **OLD** and no components are requested (since components correspond to GLVND libraries).

LEGACY

Prefer to use the legacy libGL library, if available.

For EGL targets the client must rely on GLVND support on the user's system. Linking should use the **OpenGL::OpenGL** **OpenGL::EGL** targets. Using GLES* libraries is theoretically possible in place of **OpenGL::OpenGL**, but this module does not currently support that; contributions welcome.

OPENGL_gl_LIBRARY and **OPENGL_EGL_INCLUDE_DIRS** are defined in the case of GLVND. For non-GLVND Linux and other systems these are left undefined.

macOS-Specific

On macOS this module defaults to using the macOS-native framework version of OpenGL. To use the X11 version of OpenGL on macOS, one can disable searching of frameworks. For example:

```
find_package(X11)
if(APPLE AND X11_FOUND)
  set(CMAKE_FIND_FRAMEWORK NEVER)
  find_package(OpenGL)
  unset(CMAKE_FIND_FRAMEWORK)
else()
  find_package(OpenGL)
endif()
```

An end user building this project may need to point CMake at their X11 installation, e.g., with `-DOpenGL_ROOT=/opt/X11`.

FindOpenMP

Finds Open Multi-Processing (OpenMP) support.

This module can be used to detect OpenMP support in a compiler. If the compiler supports OpenMP, the flags required to compile with OpenMP support are returned in variables for the different languages. The variables may be empty if the compiler does not need a special flag to support OpenMP.

Added in version 3.5: Clang support.

Input Variables

The following variables may be set to influence this module's behavior:

OpenMP_RUNTIME_MSVC

Added in version 3.30.

Specify the *OpenMP Runtime* when compiling with MSVC. If set to a non-empty value, such as **experimental** or **llvm**, it will be passed as the value of the `-openmp:` flag.

Result Variables

Added in version 3.10: The module exposes the components **C**, **CXX**, and **Fortran**. Each of these controls the various languages to search OpenMP support for.

Added in version 3.31: The **CUDA** language component is supported when using a CUDA compiler that supports OpenMP on the host.

Depending on the enabled components the following variables will be set:

OpenMP_FOUND

Variable indicating that OpenMP flags for all requested languages have been found. If no components are specified, this is true if OpenMP settings for all enabled languages were detected.

OpenMP_VERSION

Minimal version of the OpenMP standard detected among the requested languages, or all enabled languages if no components were specified.

This module will set the following variables per language in your project, where **<lang>** is one of C, CXX, CUDA, or Fortran:

OpenMP_<lang>_FOUND

Variable indicating if OpenMP support for **<lang>** was detected.

OpenMP_<lang>_FLAGS

OpenMP compiler flags for **<lang>**, separated by spaces.

OpenMP_<lang>_INCLUDE_DIRS

Directories that must be added to the header search path for **<lang>** when using OpenMP.

For linking with OpenMP code written in **<lang>**, the following variables are provided:

OpenMP_<lang>_LIB_NAMES

;-list of libraries for OpenMP programs for **<lang>**.

OpenMP_<libname>_LIBRARY

Location of the individual libraries needed for OpenMP support in <lang>.

OpenMP_<lang>_LIBRARIES

A list of libraries needed to link with OpenMP code written in <lang>.

Additionally, the module provides *IMPORTED* targets:

OpenMP::OpenMP_<lang>

Target for using OpenMP from <lang>.

Specifically for Fortran, the module sets the following variables:

OpenMP_Fortran_HAVE_OMPLIB_HEADER

Boolean indicating if OpenMP is accessible through **omp_lib.h**.

OpenMP_Fortran_HAVE_OMPLIB_MODULE

Boolean indicating if OpenMP is accessible through the **omp_lib** Fortran module.

The module will also try to provide the OpenMP version variables:

OpenMP_<lang>_SPEC_DATE

Added in version 3.7.

Date of the OpenMP specification implemented by the <lang> compiler.

OpenMP_<lang>_VERSION_MAJOR

Major version of OpenMP implemented by the <lang> compiler.

OpenMP_<lang>_VERSION_MINOR

Minor version of OpenMP implemented by the <lang> compiler.

OpenMP_<lang>_VERSION

OpenMP version implemented by the <lang> compiler.

The specification date is formatted as given in the OpenMP standard: **yyyymm** where **yyyy** and **mm** represents the year and month of the OpenMP specification implemented by the <lang> compiler.

For some compilers, it may be necessary to add a header search path to find the relevant OpenMP headers. This location may be language-specific. Where this is needed, the module may attempt to find the location, but it can be provided directly by setting the **OpenMP_<lang>_INCLUDE_DIR** cache variable. Note that this variable is an *_input_* control to the module. Project code should use the **OpenMP_<lang>_INCLUDE_DIRS** *_output_* variable if it needs to know what include directories are needed.

FindOpenSceneGraph

Find OpenSceneGraph (3D graphics application programming interface)

This module searches for the OpenSceneGraph core "osg" library as well as *FindOpenThreads*, and whatever additional **COMPONENTS** (nodekits) that you specify.

See <http://www.openscenegraph.org>

NOTE: To use this module effectively you must either require **CMake** **>= 2.6.3** with *cmake_minimum_required(VERSION 2.6.3)* or download and place *FindOpenThreads*, *Findosg* functions, *Find<etc>.cmake* files into your *CMAKE_MODULE_PATH*.

This module accepts the following variables (note mixed case)

OpenSceneGraph_DEBUG – Enable debugging output

OpenSceneGraph_MARK_AS_ADVANCED – Mark cache variables as advanced automatically

The following environment variables are also respected for finding the OSG and it's various components. *CMAKE_PREFIX_PATH* can also be used for this (see *find_library()* CMake documentation).

<MODULE>_DIR

(where **MODULE** is of the form "OSGVOLUME" and there is a *F indosgVolume.cmake`* file)

OSG_DIR

OSGDIR

OSG_ROOT

[CMake 2.8.10]: The CMake variable **OSG_DIR** can now be used as well to influence detection, instead of needing to specify an environment variable.

This module defines the following output variables:

OPENSCENEGGRAPH_FOUND – Was the OSG and all of the specified components found?

OPENSCENEGGRAPH_VERSION – The version of the OSG which was found

OPENSCENEGGRAPH_INCLUDE_DIRS – Where to find the headers

OPENSCENEGGRAPH_LIBRARIES – The OSG libraries

===== Example Usage:

```
find_package(OpenSceneGraph 2.0.0 REQUIRED osgDB osgUtil)
# libOpenThreads & libosg automatically searched
include_directories(${OPENSCENEGGRAPH_INCLUDE_DIRS})

add_executable(foo foo.cc)
target_link_libraries(foo ${OPENSCENEGGRAPH_LIBRARIES})
```

FindOpenSP

Added in version 3.25.

Try to find the OpenSP library.

Result Variables

This will define the following variables:

OpenSP_FOUND

True if (the requested version of) **OpenSP** is available

OpenSP_VERSION

The version of **OpenSP**

OpenSP_VERSION_MAJOR

The major version of **OpenSP**

OpenSP_VERSION_MINOR

The minor version of **OpenSP**

OpenSP_VERSION_PATCH

The patch version of **OpenSP**

OpenSP_INCLUDE_DIRS

The include dirs of **OpenSP** with its headers

OpenSP_LIBRARIES

The OpenSP library for use with `target_link_libraries()`. This can be passed to `target_link_libraries()` instead of the *IMPORTED* **OpenSP::OpenSP** target

OpenSP_MULTI_BYTE

True if **SP_MULTI_BYTE** was found to be defined in OpenSP's **config.h** header file, which indicates that the **OpenSP** library was compiled with support for multi-byte characters. The consuming target needs to define the **SP_MULTI_BYTE** to match this value in order to avoid issues with character decoding.

Imported Targets

This module defines the *IMPORTED* target **OpenSP::OpenSP**, if OpenSP has been found.

Cache variables

The following cache variables may also be set:

OpenSP_INCLUDE_DIR

the OpenSP include directory

OpenSP_LIBRARY

the absolute path of the osp library

FindOpenSSL

Find the OpenSSL encryption library.

This module finds an installed OpenSSL library and determines its version.

Added in version 3.19: When a version is requested, it can be specified as a simple value or as a range. For a detailed description of version range usage and capabilities, refer to the *find_package()* command.

Added in version 3.18: Support for OpenSSL 3.0.

Optional COMPONENTS

Added in version 3.12.

This module supports two optional COMPONENTS: **Crypto** and **SSL**. Both components have associated imported targets, as described below.

Imported Targets

Added in version 3.4.

This module defines the following *IMPORTED* targets:

OpenSSL::SSL

The OpenSSL **ssl** library, if found.

OpenSSL::Crypto

The OpenSSL **crypto** library, if found.

OpenSSL::applink

Added in version 3.18.

The OpenSSL **applink** components that might be need to be compiled into projects under MSVC. This target is available only if found OpenSSL version is not less than 0.9.8. By linking this target the above OpenSSL targets can be linked even if the project has different MSVC runtime configurations with the above OpenSSL targets. This target has no effect on platforms other than MSVC.

NOTE: Due to how **INTERFACE_SOURCES** are consumed by the consuming target, unless you certainly know what you are doing, it is always preferred to link **OpenSSL::applink** target as **PRIVATE** and to make sure that this target is linked at most once for the whole dependency graph of any library or executable:

```
target_link_libraries(myTarget PRIVATE OpenSSL::applink)
```

Otherwise you would probably encounter unexpected random problems when building and linking, as both the ISO C and the ISO C++ standard claims almost nothing about what a link process should be.

Result Variables

This module will set the following variables in your project:

OPENSSL_FOUND

System has the OpenSSL library. If no components are requested it only requires the crypto library.

OPENSSL_INCLUDE_DIR

The OpenSSL include directory.

OPENSSL_CRYPTO_LIBRARY

The OpenSSL crypto library.

OPENSSL_CRYPTO_LIBRARIES

The OpenSSL crypto library and its dependencies.

OPENSSL_SSL_LIBRARY

The OpenSSL SSL library.

OPENSSL_SSL_LIBRARIES

The OpenSSL SSL library and its dependencies.

OPENSSL_LIBRARIES

All OpenSSL libraries and their dependencies.

OPENSSL_VERSION

This is set to **\$major.\$minor.\$revision\$patch** (e.g. **0.9.8s**).

OPENSSL_APPLINK_SOURCE

The sources in the target **OpenSSL::applink** that is mentioned above. This variable shall always be undefined if found openssl version is less than 0.9.8 or if platform is not MSVC.

Hints

The following variables may be set to control search behavior:

OPENSSL_ROOT_DIR

Set to the root directory of an OpenSSL installation.

OPENSSL_USE_STATIC_LIBS

Added in version 3.4.

Set to **TRUE** to look for static libraries.

OPENSSL_MSVC_STATIC_RT

Added in version 3.5.

Set to **TRUE** to choose the MT version of the lib.

ENV{PKG_CONFIG_PATH}

On UNIX-like systems, **pkg-config** is used to locate the system OpenSSL. Set the **PKG_CONFIG_PATH** environment variable to look in alternate locations. Useful on multi-lib systems.

FindOpenThreads

OpenThreads is a C++ based threading library. Its largest userbase seems to be OpenSceneGraph so you might notice I accept OSGDIR as an environment path. I consider this part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module.

This module defines:

OPENTHREADS_LIBRARY**OPENTHREADS_FOUND**

if false, do not try to link to OpenThreads

OPENTHREADS_INCLUDE_DIR

where to find the headers

\$OPENTHREADS_DIR is an environment variable that would correspond to the:

```
./configure --prefix=$OPENTHREADS_DIR
```

used in building osg.

Added in version 2.8.10: The CMake variables **OPENTHREADS_DIR** or **OSG_DIR** can now be used as well to influence detection, instead of needing to specify an environment variable.

Findosg**NOTE:**

It is highly recommended that you use the new *FindOpenSceneGraph* introduced in CMake 2.6.3 and not use this Find module directly.

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osg This module defines:

OSG_FOUND

Was the Osg found?

OSG_INCLUDE_DIR

Where to find the headers

OSG_LIBRARIES

The libraries to link against for the OSG (use this)

OSG_LIBRARY

The OSG library

OSG_LIBRARY_DEBUG

The OSG debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

Findosg_functions

This CMake file contains two macros to assist with searching for OSG libraries and nodekits. Please see *FindOpenSceneGraph* for full documentation.

FindosgAnimation

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgAnimation This module defines:

OSGANIMATION_FOUND

Was osgAnimation found?

OSGANIMATION_INCLUDE_DIR

Where to find the headers

OSGANIMATION_LIBRARIES

The libraries to link against for the OSG (use this)

OSGANIMATION_LIBRARY

The OSG library

OSGANIMATION_LIBRARY_DEBUG

The OSG debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg. Created by Eric Wing.

FindosgDB

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgDB This module defines:

OSGDB_FOUND

Was osgDB found?

OSGDB_INCLUDE_DIR

Where to find the headers

OSGDB_LIBRARIES

The libraries to link against for the osgDB

OSGDB_LIBRARY

The osgDB library

OSGDB_LIBRARY_DEBUG

The osgDB debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

FindosgFX

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgFX This module defines:

OSGFX_FOUND

Was osgFX found?

OSGFX_INCLUDE_DIR

Where to find the headers

OSGFX_LIBRARIES

The libraries to link against for the osgFX (use this)

OSGFX_LIBRARY

The osgFX library

OSGFX_LIBRARY_DEBUG

The osgFX debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgGA

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you

want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgGA This module defines:

OSGGA_FOUND

Was osgGA found?

OSGGA_INCLUDE_DIR

Where to find the headers

OSGGA_LIBRARIES

The libraries to link against for the osgGA (use this)

OSGGA_LIBRARY

The osgGA library

OSGGA_LIBRARY_DEBUG

The osgGA debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgIntrospection

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgINTROSPECTION This module defines:

OSGINTROSPECTION_FOUND

Was osgIntrospection found?

OSGINTROSPECTION_INCLUDE_DIR

Where to find the headers

OSGINTROSPECTION_LIBRARIES

The libraries to link for osgIntrospection (use this)

OSGINTROSPECTION_LIBRARY

The osgIntrospection library

OSGINTROSPECTION_LIBRARY_DEBUG

The osgIntrospection debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgManipulator

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgManipulator This module defines:

OSGMANIPULATOR_FOUND

Was osgManipulator found?

OSGMANIPULATOR_INCLUDE_DIR

Where to find the headers

OSGMANIPULATOR_LIBRARIES

The libraries to link for osgManipulator (use this)

OSGMANIPULATOR_LIBRARY

The osgManipulator library

OSGMANIPULATOR_LIBRARY_DEBUG

The osgManipulator debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgParticle

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgParticle This module defines:

OSGPARTICLE_FOUND

Was osgParticle found?

OSGPARTICLE_INCLUDE_DIR

Where to find the headers

OSGPARTICLE_LIBRARIES

The libraries to link for osgParticle (use this)

OSGPARTICLE_LIBRARY

The osgParticle library

OSGPARTICLE_LIBRARY_DEBUG

The osgParticle debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgPresentation

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgPresentation This module defines:

OSGPresentation_FOUND

Was osgPresentation found?

OSGPresentation_INCLUDE_DIR

Where to find the headers

OSGPresentation_LIBRARIES

The libraries to link for osgPresentation (use this)

OSGPresentation_LIBRARY

The osgPresentation library

OSGPresentation_LIBRARY_DEBUG

The osgPresentation debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg. Created by Eric Wing. Modified to work with osgPresentation by Robert Osfield, January 2012.

FindosgProducer

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgProducer This module defines:

OSGProducer_FOUND

Was osgProducer found?

OSGProducer_INCLUDE_DIR

Where to find the headers

OSGProducer_LIBRARIES

The libraries to link for osgProducer (use this)

OSGPRODUCER_LIBRARY

The osgProducer library

OSGPRODUCER_LIBRARY_DEBUG

The osgProducer debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgQt

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgQt This module defines:

OSGQT_FOUND

Was osgQt found?

OSGQT_INCLUDE_DIR

Where to find the headers

OSGQT_LIBRARIES

The libraries to link for osgQt (use this)

OSGQT_LIBRARY

The osgQt library

OSGQT_LIBRARY_DEBUG

The osgQt debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing. Modified to work with osgQt by Robert Osfield, January 2012.

FindosgShadow

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgShadow This module defines:

OSGSHADOW_FOUND

Was osgShadow found?

OSGSHADOW_INCLUDE_DIR

Where to find the headers

OSGSHADOW_LIBRARIES

The libraries to link for osgShadow (use this)

OSGSHADOW_LIBRARY

The osgShadow library

OSGSHADOW_LIBRARY_DEBUG

The osgShadow debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgSim

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgSim This module defines:

OSGSIM_FOUND

Was osgSim found?

OSGSIM_INCLUDE_DIR

Where to find the headers

OSGSIM_LIBRARIES

The libraries to link for osgSim (use this)

OSGSIM_LIBRARY

The osgSim library

OSGSIM_LIBRARY_DEBUG

The osgSim debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgTerrain

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by

piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgTerrain This module defines:

OSGTERRAIN_FOUND

Was osgTerrain found?

OSGTERRAIN_INCLUDE_DIR

Where to find the headers

OSGTERRAIN_LIBRARIES

The libraries to link for osgTerrain (use this)

OSGTERRAIN_LIBRARY

The osgTerrain library

OSGTERRAIN_LIBRARY_DEBUG

The osgTerrain debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgText

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgText This module defines:

OSGTEXT_FOUND

Was osgText found?

OSGTEXT_INCLUDE_DIR

Where to find the headers

OSGTEXT_LIBRARIES

The libraries to link for osgText (use this)

OSGTEXT_LIBRARY

The osgText library

OSGTEXT_LIBRARY_DEBUG

The osgText debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgUtil

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgUtil This module defines:

OSGUTIL_FOUND

Was osgUtil found?

OSGUTIL_INCLUDE_DIR

Where to find the headers

OSGUTIL_LIBRARIES

The libraries to link for osgUtil (use this)

OSGUTIL_LIBRARY

The osgUtil library

OSGUTIL_LIBRARY_DEBUG

The osgUtil debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgViewer

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgViewer This module defines:

OSGVIEWER_FOUND

Was osgViewer found?

OSGVIEWER_INCLUDE_DIR

Where to find the headers

OSGVIEWER_LIBRARIES

The libraries to link for osgViewer (use this)

OSGVIEWER_LIBRARY

The osgViewer library

OSGVIEWER_LIBRARY_DEBUG

The osgViewer debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgVolume

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgVolume This module defines:

OSGVOLUME_FOUND

Was osgVolume found?

OSGVOLUME_INCLUDE_DIR

Where to find the headers

OSGVOLUME_LIBRARIES

The libraries to link for osgVolume (use this)

OSGVOLUME_LIBRARY

The osgVolume library

OSGVOLUME_LIBRARY_DEBUG

The osgVolume debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

Created by Eric Wing.

FindosgWidget

This is part of the **Findosg*** suite used to find OpenSceneGraph components. Each component is separate and you must opt in to each module. You must also opt into OpenGL and OpenThreads (and Producer if needed) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the **Findosg*.cmake** modules.

Locate osgWidget This module defines:

OSGWIDGET_FOUND

Was osgWidget found?

OSGWIDGET_INCLUDE_DIR

Where to find the headers

OSGWIDGET_LIBRARIES

The libraries to link for osgWidget (use this)

OSGWIDGET_LIBRARY

The osgWidget library

OSGWIDGET_LIBRARY_DEBUG

The osgWidget debug library

\$OSGDIR is an environment variable that would correspond to:

```
./configure --prefix=$OSGDIR
```

used in building osg.

FindosgWidget.cmake tweaked from Findosg* suite as created by Eric Wing.

FindPatch

Added in version 3.10.

The module defines the following variables:

Patch_EXECUTABLE

Path to patch command-line executable.

Patch_FOUND

True if the patch command-line executable was found.

The following *IMPORTED* targets are also defined:

Patch::patch

The command-line executable.

Changed in version 4.0: Imported target is defined only when *CMAKE_ROLE* is **PROJECT**.

Example usage:

```
find_package(Patch)
if(Patch_FOUND)
  message("Patch found: ${Patch_EXECUTABLE}")
endif()
```

FindPerl

Find a Perl interpreter.

This module defines the following variables:

PERL_EXECUTABLE

The full path to Perl.

PERL_FOUND

True if the Perl executable was found.

PERL_VERSION_STRING

Added in version 2.8.8.

The version of Perl found.

FindPerlLibs

Find Perl libraries

This module finds if PERL is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

```

PERLLIBS_FOUND      = True if perl.h & libperl were found
PERL_INCLUDE_PATH   = path to where perl.h is found
PERL_LIBRARY        = path to libperl
PERL_EXECUTABLE     = full path to the perl binary

```

The minimum required version of Perl can be specified using the standard syntax, e.g. `find_package(Perl-Libs 6.0)`

The following variables are also available if needed (introduced after CMake 2.6.4)

```

PERL_SITESEARCH      = path to the sitesearch install dir (-V:installsite)
PERL_SITEARCH        = path to the sitelib install directory (-V:installsite)
PERL_SITELIB         = path to the sitelib install directory (-V:installsite)
PERL_VENDORARCH      = path to the vendor arch install directory (-V:installvendor)
PERL_VENDORLIB       = path to the vendor lib install directory (-V:installvendor)
PERL_ARCHLIB         = path to the core arch lib install directory (-V:archlib)
PERL_PRIVLIB         = path to the core priv lib install directory (-V:privlib)
PERL_UPDATE_ARCHLIB  = path to the update arch lib install directory (-V:installupdate)
PERL_UPDATE_PRIVLIB  = path to the update priv lib install directory (-V:installupdate)
PERL_EXTRA_C_FLAGS   = Compilation flags used to build perl

```

FindPHP4

Find PHP4

This module finds if PHP4 is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

```

PHP4_INCLUDE_PATH    = path to where php.h can be found
PHP4_EXECUTABLE      = full path to the php4 binary

```

FindPhysFS

Locate PhysFS library This module defines:

PHYSFS_LIBRARY

the name of the library to link against

PHYSFS_FOUND

if false, do not try to link to PHYSFS

PHYSFS_INCLUDE_DIR

where to find physfs.h

\$PHYSFSDIR is an environment variable that would correspond to:

```
./configure --prefix=$PHYSFSDIR
```

used in building PHYSFS.

FindPike

Find Pike

This module finds if PIKE is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

```

PIKE_INCLUDE_PATH      = path to where program.h is found
PIKE_EXECUTABLE        = full path to the pike binary

```

FindPkgConfig

A **pkg-config** module for CMake.

Finds the **pkg-config** executable and adds the *pkg_get_variable()*, *pkg_check_modules()* and *pkg_search_module()* commands. The following variables will also be set:

PKG_CONFIG_FOUND

True if a **pkg-config** executable was found.

PKG_CONFIG_VERSION_STRING

Added in version 2.8.8.

The version of **pkg-config** that was found.

PKG_CONFIG_EXECUTABLE

The pathname of the **pkg-config** program.

PKG_CONFIG_ARGN

Added in version 3.22.

A list of arguments to pass to **pkg-config**.

Both **PKG_CONFIG_EXECUTABLE** and **PKG_CONFIG_ARGN** are initialized by the module, but may be overridden by the user. See *Variables Affecting Behavior* for how these variables are initialized.

pkg_check_modules

Checks for all the given modules, setting a variety of result variables in the calling scope.

```

pkg_check_modules(<prefix>
    [REQUIRED] [QUIET]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    [IMPORTED_TARGET [GLOBAL]]
    <moduleSpec> [<moduleSpec>...])

```

When the **REQUIRED** argument is given, the command will fail with an error if module(s) could not be found.

When the **QUIET** argument is given, no status messages will be printed.

Added in version 3.3: The *CMAKE_PREFIX_PATH*, *CMAKE_FRAMEWORK_PATH*, and *CMAKE_APPBUNDLE_PATH* cache and environment variables will be added to the **pkg-config** search path. The **NO_CMAKE_PATH** and **NO_CMAKE_ENVIRONMENT_PATH** arguments disable this behavior for the cache variables and environment variables respectively. The *PKG_CONFIG_USE_CMAKE_PREFIX_PATH* variable set to **FALSE** disables this behavior globally.

Added in version 3.7: The **IMPORTED_TARGET** argument will create an imported target named **PkgConfig::<prefix>** that can be passed directly as an argument to *target_link_libraries()*.

Added in version 3.13: The **GLOBAL** argument will make the imported target available in global scope.

Added in version 3.15: Non-library linker options reported by **pkg-config** are stored in the *INTERFACE_LINK_OPTIONS* target property.

Changed in version 3.18: Include directories specified with **-isystem** are stored in the *INTERFACE_INCLUDE_DIRECTORIES* target property. Previous versions of CMake left them in the *INTERFACE_COMPILE_OPTIONS* property.

Each **<moduleSpec>** can be either a bare module name or it can be a module name with a version constraint (operators =, <, >, <= and >= are supported). The following are examples for a module named **foo** with various constraints:

- **foo** matches any version.
- **foo<2** only matches versions before 2.
- **foo>=3.1** matches any version from 3.1 or later.
- **foo=1.2.3** requires that foo must be exactly version 1.2.3.

The following variables may be set upon return. Two sets of values exist: One for the common case (**<XXX> = <prefix>**) and another for the information **pkg-config** provides when called with the **--static** option (**<XXX> = <prefix>_STATIC**).

<XXX>_FOUND
set to 1 if module(s) exist

<XXX>_LIBRARIES
only the libraries (without the '-l')

<XXX>_LINK_LIBRARIES
the libraries and their absolute paths

<XXX>_LIBRARY_DIRS
the paths of the libraries (without the '-L')

<XXX>_LDFLAGS
all required linker flags

<XXX>_LDFLAGS_OTHER
all other linker flags

<XXX>_INCLUDE_DIRS
the '-I' preprocessor flags (without the '-I')

<XXX>_CFLAGS
all required cflags

<XXX>_CFLAGS_OTHER
the other compiler flags

All but **<XXX>_FOUND** may be a **;-list** if the associated variable returned from **pkg-config** has multiple values.

Changed in version 3.18: Include directories specified with **-isystem** are stored in the **<XXX>_INCLUDE_DIRS** variable. Previous versions of CMake left them in **<XXX>_CFLAGS_OTHER**.

There are some special variables whose prefix depends on the number of **<moduleSpec>** given. When there is only one **<moduleSpec>**, **<YYY>** will simply be **<prefix>**, but if two or more **<moduleSpec>** items are given, **<YYY>** will be **<prefix>_<moduleName>**.

<YYY>_VERSION

version of the module

<YYY>_PREFIX

prefix directory of the module

<YYY>_INCLUDEDIR

include directory of the module

<YYY>_LIBDIR

lib directory of the module

Changed in version 3.8: For any given **<prefix>**, **pkg_check_modules()** can be called multiple times with different parameters. Previous versions of CMake cached and returned the first successful result.

Changed in version 3.16: If a full path to the found library can't be determined, but it's still visible to the linker, pass it through as **-l<name>**. Previous versions of CMake failed in this case.

Examples:

```
pkg_check_modules (GLIB2 glib-2.0)
```

Looks for any version of glib2. If found, the output variable **GLIB2_VERSION** will hold the actual version found.

```
pkg_check_modules (GLIB2 glib-2.0>=2.10)
```

Looks for at least version 2.10 of glib2. If found, the output variable **GLIB2_VERSION** will hold the actual version found.

```
pkg_check_modules (FOO glib-2.0>=2.10 gtk+-2.0)
```

Looks for both glib2-2.0 (at least version 2.10) and any version of gtk2+-2.0. Only if both are found will **FOO** be considered found. The **FOO_glib-2.0_VERSION** and **FOO_gtk+-2.0_VERSION** variables will be set to their respective found module versions.

```
pkg_check_modules (XRENDER REQUIRED xrender)
```

Requires any version of **xrender**. Example output variables set by a successful call:

```
XRENDER_LIBRARIES=Xrender;X11
XRENDER_STATIC_LIBRARIES=Xrender;X11;pthread;Xau;Xdmcp
```

pkg_search_module

The behavior of this command is the same as *pkg_check_modules()*, except that rather than checking for all the specified modules, it searches for just the first successful match.

```
pkg_search_module(<prefix>
                  [REQUIRED] [QUIET]
                  [NO_CMAKE_PATH]
```

```
[NO_CMAKE_ENVIRONMENT_PATH]
[IMPORTED_TARGET [GLOBAL]]
<moduleSpec> [<moduleSpec>...])
```

Added in version 3.16: If a module is found, the **<prefix>_MODULE_NAME** variable will contain the name of the matching module. This variable can be used if you need to run *pkg_get_variable()*.

Example:

```
pkg_search_module (BAR libxml-2.0 libxml2 libxml>=2)
```

pkg_get_variable

Added in version 3.4.

Retrieves the value of a pkg-config variable **varName** and stores it in the result variable **resultVar** in the calling scope.

```
pkg_get_variable(<resultVar> <moduleName> <varName>
[DEFINE_VARIABLES <key>=<value>...])
```

If **pkg-config** returns multiple values for the specified variable, **resultVar** will contain a *;-list*.

Options:

DEFINE_VARIABLES <key>=<value>...

Added in version 3.28.

Specify key-value pairs to redefine variables affecting the variable retrieved with **pkg-config**.

For example:

```
pkg_get_variable(GI_GIRDIR gobject-introspection-1.0 giridir)
```

Variables Affecting Behavior

PKG_CONFIG_EXECUTABLE

This cache variable can be set to the path of the pkg-config executable. *find_program()* is called internally by the module with this variable.

Added in version 3.1: The **PKG_CONFIG** environment variable can be used as a hint if **PKG_CONFIG_EXECUTABLE** has not yet been set.

Changed in version 3.22: If the **PKG_CONFIG** environment variable is set, only the first argument is taken from it when using it as a hint.

PKG_CONFIG_ARGN

Added in version 3.22.

This cache variable can be set to a list of arguments to additionally pass to pkg-config if needed. If not provided, it will be initialized from the **PKG_CONFIG** environment variable, if set. The first

argument in that environment variable is assumed to be the `pkg-config` program, while all remaining arguments after that are used to initialize **PKG_CONFIG_ARGN**. If no such environment variable is defined, **PKG_CONFIG_ARGN** is initialized to an empty string. The module does not update the variable once it has been set in the cache.

PKG_CONFIG_USE_CMAKE_PREFIX_PATH

Added in version 3.1.

Specifies whether `pkg_check_modules()` and `pkg_search_module()` should add the paths in the `CMAKE_PREFIX_PATH`, `CMAKE_FRAMEWORK_PATH` and `CMAKE_APPBUNDLE_PATH` cache and environment variables to the **pkg-config** search path.

If this variable is not set, this behavior is enabled by default if `CMAKE_MINIMUM_REQUIRED_VERSION` is 3.1 or later, disabled otherwise.

FindPNG

Find `libpng`, the official reference library for the PNG image format.

Imported Targets

Added in version 3.5.

This module defines the following *IMPORTED* target:

PNG::PNG

The `libpng` library, if found.

Result variables

This module will set the following variables in your project:

PNG_INCLUDE_DIRS

where to find `png.h`, etc.

PNG_LIBRARIES

the libraries to link against to use PNG.

PNG_DEFINITIONS

You should add `add_definitions(${PNG_DEFINITIONS})` before compiling code that includes `png` library files.

PNG_FOUND

If false, do not try to use PNG.

PNG_VERSION_STRING

the version of the PNG library found (since CMake 2.8.8)

Obsolete variables

The following variables may also be set, for backwards compatibility:

PNG_LIBRARY

where to find the PNG library.

PNG_INCLUDE_DIR

where to find the PNG headers (same as `PNG_INCLUDE_DIRS`)

Since PNG depends on the ZLib compression library, none of the above will be defined unless ZLib can be found.

FindPostgreSQL

Find the PostgreSQL installation.

Imported Targets

Added in version 3.14.

This module defines *IMPORTED* target **PostgreSQL::PostgreSQL** if PostgreSQL has been found.

Result Variables

This module will set the following variables in your project:

PostgreSQL_FOUND

True if PostgreSQL is found.

PostgreSQL_LIBRARIES

the PostgreSQL libraries needed for linking

PostgreSQL_INCLUDE_DIRS

the directories of the PostgreSQL headers

PostgreSQL_LIBRARY_DIRS

the link directories for PostgreSQL libraries

PostgreSQL_VERSION_STRING

the version of PostgreSQL found

PostgreSQL_TYPE_INCLUDE_DIR

the directories of the PostgreSQL server headers

Components

This module contains additional **Server** component, that forcibly checks for the presence of server headers. Note that **PostgreSQL_TYPE_INCLUDE_DIR** is set regardless of the presence of the **Server** component in `find_package` call.

FindProducer

Though Producer isn't directly part of OpenSceneGraph, its primary user is OSG so I consider this part of the Findosg* suite used to find OpenSceneGraph components. You'll notice that I accept OSGDIR as an environment path.

Each component is separate and you must opt in to each module. You must also opt into OpenGL (and OpenThreads?) as these modules won't do it for you. This is to allow you control over your own system piece by piece in case you need to opt out of certain components or change the Find behavior for a particular module (perhaps because the default *FindOpenGL* module doesn't work with your system as an example). If you want to use a more convenient module that includes everything, use the *FindOpenSceneGraph* instead of the Findosg*.cmake modules.

Locate Producer This module defines:

PRODUCER_LIBRARY**PRODUCER_FOUND**

if false, do not try to link to Producer

PRODUCER_INCLUDE_DIR

where to find the headers

\$PRODUCER_DIR is an environment variable that would correspond to:

```
./configure --prefix=$PRODUCER_DIR
```

used in building osg.

FindProtobuf

Locate and configure the Google Protocol Buffers library.

Added in version 3.6: Support for *find_package()* version checks.

Changed in version 3.6: All input and output variables use the **Protobuf_** prefix. Variables with **PROTOBUF_** prefix are still supported for compatibility.

The following variables can be set and are optional:

Protobuf_SRC_ROOT_FOLDER

When compiling with MSVC, if this cache variable is set the protobuf–default VS project build locations (vsprojects/Debug and vsprojects/Release or vsprojects/x64/Debug and vsprojects/x64/Release) will be searched for libraries and binaries.

Protobuf_IMPORT_DIRS

List of additional directories to be searched for imported .proto files.

Protobuf_DEBUG

Added in version 3.6.

Show debug messages.

Protobuf_USE_STATIC_LIBS

Added in version 3.9.

Set to ON to force the use of the static libraries. Default is OFF.

Defines the following variables:

Protobuf_FOUND

Found the Google Protocol Buffers library (libprotobuf & header files)

Protobuf_VERSION

Added in version 3.6.

Version of package found.

Protobuf_INCLUDE_DIRS

Include directories for Google Protocol Buffers

Protobuf_LIBRARIES

The protobuf libraries

Protobuf_PROTOC_LIBRARIES

The protoc libraries

Protobuf_LITE_LIBRARIES

The protobuf–lite libraries

Added in version 3.9: The following *IMPORTED* targets are also defined:

protobuf::libprotobuf

The protobuf library.

protobuf::libprotobuf-lite

The protobuf lite library.

protobuf::libprotoc

The protoc library.

protobuf::protoc

Added in version 3.10: The protoc compiler.

The following cache variables are also available to set or use:

Protobuf_LIBRARY

The protobuf library

Protobuf_PROTOC_LIBRARY

The protoc library

Protobuf_INCLUDE_DIR

The include directory for protocol buffers

Protobuf_PROTOC_EXECUTABLE

The protoc compiler

Protobuf_LIBRARY_DEBUG

The protobuf library (debug)

Protobuf_PROTOC_LIBRARY_DEBUG

The protoc library (debug)

Protobuf_LITE_LIBRARY

The protobuf lite library

Protobuf_LITE_LIBRARY_DEBUG

The protobuf lite library (debug)

Example:

```
find_package(Protobuf REQUIRED)
include_directories(${Protobuf_INCLUDE_DIRS})
include_directories(${CMAKE_CURRENT_BINARY_DIR})
protobuf_generate_cpp(PROTO_SRCS PROTO_HDRS foo.proto)
protobuf_generate_cpp(PROTO_SRCS PROTO_HDRS EXPORT_MACRO DLL_EXPORT foo.proto)
protobuf_generate_cpp(PROTO_SRCS PROTO_HDRS DESCRIPTORS PROTO_DESCS foo.proto)
protobuf_generate_python(PROTO_PY foo.proto)
add_executable(bar bar.cc ${PROTO_SRCS} ${PROTO_HDRS})
target_link_libraries(bar ${Protobuf_LIBRARIES})
```

NOTE:

The **protobuf_generate_cpp** and **protobuf_generate_python** functions and *add_executable()* or *add_library()* calls only work properly within the same directory.

protobuf_generate_cpp

Add custom commands to process **.proto** files to C++:

```
protobuf_generate_cpp (
  <srcs-var> <hdrs-var>
  [DESCRIPTORS <var>]
  [EXPORT_MACRO <macro>]
  [<proto-file>...])
```

<srcs-var>

Variable to define with autogenerated source files

<hdrs-var>

Variable to define with autogenerated header files

DESCRIPTORS <var>

Added in version 3.10: Variable to define with autogenerated descriptor files, if requested.

EXPORT_MACRO <macro>

is a macro which should expand to **__declspec(dllexport)** or **__declspec(dllimport)** depending on what is being compiled.

<proto-file>...

.proto files

protobuf_generate_python

Added in version 3.4.

Add custom commands to process **.proto** files to Python:

```
protobuf_generate_python (<py-srcs-var> [<proto-file>...])
```

<py-srcs-var>

Variable to define with autogenerated Python files

<proto-file>...

.proto files

protobuf_generate

Added in version 3.13.

Automatically generate source files from **.proto** schema files at build time:

```
protobuf_generate (
  TARGET <target>
  [LANGUAGE <lang>]
  [OUT_VAR <var>]
  [EXPORT_MACRO <macro>]
  [PROTOC_OUT_DIR <dir>]
  [PLUGIN <plugin>]
  [PLUGIN_OPTIONS <plugin-options>]
  [DEPENDENCIES <dependencies>]
  [PROTOS <proto-file>...]
  [IMPORT_DIRS <dir>...]
  [GENERATE_EXTENSIONS <extension>...]
  [PROTOC_OPTIONS <option>...]
  [PROTOC_EXE <executable>]
  [APPEND_PATH])
```

APPEND_PATH

A flag that causes the base path of all proto schema files to be added to **IMPORT_DIRS**.

LANGUAGE <lang>

A single value: **c++** or **python**. Determines what kind of source files are being generated. Defaults to **c++**.

OUT_VAR <var>

Name of a CMake variable that will be filled with the paths to the generated source files.

EXPORT_MACRO <macro>

Name of a macro that is applied to all generated Protobuf message classes and extern variables. It can, for example, be used to declare DLL exports.

PROTOUT_DIR <dir>

Output directory of generated source files. Defaults to **CMAKE_CURRENT_BINARY_DIR**.

PLUGIN <plugin>

Added in version 3.21.

An optional plugin executable. This could, for example, be the path to **grpc_cpp_plugin**.

PLUGIN_OPTIONS <plugin-options>

Added in version 3.28.

Additional options provided to the plugin, such as **generate_mock_code=true** for the gRPC cpp plugin.

DEPENDENCIES <dependencies>

Added in version 3.28.

Arguments forwarded to the **DEPENDS** of the underlying **add_custom_command** invocation.

TARGET <target>

CMake target that will have the generated files added as sources.

PROTOS <proto-file>...

List of proto schema files. If omitted, then every source file ending in *proto* of **TARGET** will be used.

IMPORT_DIRS <dir>...

A common parent directory for the schema files. For example, if the schema file is **proto/helloworld/helloworld.proto** and the import directory **proto/** then the generated files are **\${PROTOUT_DIR}/helloworld/helloworld.pb.h** and **\${PROTOUT_DIR}/helloworld/helloworld.pb.cc**.

GENERATE_EXTENSIONS <extension>...

If **LANGUAGE** is omitted then this must be set to the extensions that protoc generates.

PROTOUT_OPTIONS <option>...

Added in version 3.28.

Additional arguments that are forwarded to protoc.

PROTOUT_EXE <executable>

Added in version 4.0.

Command name, path, or CMake executable used to generate protobuf bindings. If omitted, **protoc::protoc** is used.

Example:

```

find_package(gRPC CONFIG REQUIRED)
find_package(Protobuf REQUIRED)
add_library(ProtoTest Test.proto)
target_link_libraries(ProtoTest PUBLIC gRPC::grpc++)
protobuf_generate(TARGET ProtoTest)
protobuf_generate(
    TARGET ProtoTest
    LANGUAGE grpc
    PLUGIN protoc-gen-grpc=${<TARGET_FILE:gRPC::grpc_cpp_plugin>}
    PLUGIN_OPTIONS generate_mock_code=true
    GENERATE_EXTENSIONS .grpc.pb.h .grpc.pb.cc)

```

FindPython

Added in version 3.12.

Find Python interpreter, compiler and development environment (include directories and libraries).

Added in version 3.19: When a version is requested, it can be specified as a simple value or as a range. For a detailed description of version range usage and capabilities, refer to the *find_package()* command.

The following components are supported:

- **Interpreter:** search for Python interpreter.
- **Compiler:** search for Python compiler. Only offered by IronPython.
- **Development:** search for development artifacts (include directories and libraries).

Added in version 3.18: This component includes two sub-components which can be specified independently:

- **Development.Module:** search for artifacts for Python module developments.
- **Development.Embed:** search for artifacts for Python embedding developments.

Added in version 3.26:

- **Development.SABIModule:** search for artifacts for Python module developments using the *Stable Application Binary Interface*. This component is available only for version 3.2 and upper.
- **NumPy:** search for NumPy include directories.

Added in version 3.14: Added the **NumPy** component.

If no **COMPONENTS** are specified, **Interpreter** is assumed.

If component **Development** is specified, it implies sub-components **Development.Module** and **Development.Embed**.

To ensure consistent versions between components **Interpreter**, **Compiler**, **Development** (or one of its sub-components) and **NumPy**, specify all components at the same time:

```
find_package (Python COMPONENTS Interpreter Development)
```

This module looks preferably for version 3 of Python. If not found, version 2 is searched. To manage concurrent versions 3 and 2 of Python, use *FindPython3* and *FindPython2* modules rather than this one.

NOTE:

If components **Interpreter** and **Development** (or one of its sub-components) are both specified, this module search only for interpreter with same platform architecture as the one defined by CMake configuration. This constraint does not apply if only **Interpreter** component is specified.

Imported Targets

This module defines the following *Imported Targets*:

Changed in version 3.14: *Imported Targets* are only created when *CMAKE_ROLE* is **PROJECT**.

Python::Interpreter

Python interpreter. This target is defined only if the **Interpreter** component is found.

Python::InterpreterDebug

Added in version 3.30.

Python debug interpreter. This target is defined only if the **Interpreter** component is found and the **Python_EXECUTABLE_DEBUG** variable is defined. The target is only defined on the **Windows** platform.

Python::InterpreterMultiConfig

Added in version 3.30.

Python interpreter. The release or debug version of the interpreter will be used, based on the context (platform, configuration). This target is defined only if the **Interpreter** component is found

Python::Compiler

Python compiler. This target is defined only if the **Compiler** component is found.

Python::Module

Added in version 3.15.

Python library for Python module. Target defined if component **Development.Module** is found.

Python::SABIModule

Added in version 3.26.

Python library for Python module using the Stable Application Binary Interface. Target defined if component **Development.SABIModule** is found.

Python::Python

Python library for Python embedding. Target defined if component **Development.Embed** is found.

Python::NumPy

Added in version 3.14.

NumPy Python library. Target defined if component **NumPy** is found.

Result Variables

This module will set the following variables in your project (see *Standard Variable Names*):

Python_FOUND

System has the Python requested components.

Python_Interpreter_FOUND

System has the Python interpreter.

Python_EXECUTABLE

Path to the Python interpreter.

Python_EXECUTABLE_DEBUG

Added in version 3.30.

Path to the debug Python interpreter. It is only defined on the **Windows** platform.

Python_INTERPRETER

Added in version 3.30.

Path to the Python interpreter, defined as a *generator expression* selecting the **Python_EXECUTABLE** or **Python_EXECUTABLE_DEBUG** variable based on the context (platform, configuration).

Python_INTERPRETER_ID

A short string unique to the interpreter. Possible values include:

- Python
- ActivePython
- Anaconda
- Canopy
- IronPython
- PyPy

Python_STDLIB

Standard platform independent installation directory.

Information returned by `sysconfig.get_path('stdlib')`.

Python_STDARCH

Standard platform dependent installation directory.

Information returned by `sysconfig.get_path('platstdlib')`.

Python_SITELIB

Third-party platform independent installation directory.

Information returned by `sysconfig.get_path('purelib')`.

Python_SITEARCH

Third-party platform dependent installation directory.

Information returned by `sysconfig.get_path('platlib')`.

Python_SOABI

Added in version 3.17.

Extension suffix for modules.

Information computed from `sysconfig.get_config_var('EXT_SUFFIX')` or `sysconfig.get_config_var('SOABI')` or `python3-config --extension-suffix`.

Python_SOSABI

Added in version 3.26.

Extension suffix for modules using the Stable Application Binary Interface.

Information computed from `importlib.machinery.EXTENSION_SUFFIXES` if the COMPONENT **Interpreter** was specified. Otherwise, the extension is **abi3** except for **Windows**, **MSYS** and **CYGWIN** for which this is an empty string.

Python_Compiler_FOUND

System has the Python compiler.

Python_COMPILER

Path to the Python compiler. Only offered by IronPython.

Python_COMPILER_ID

A short string unique to the compiler. Possible values include:

- IronPython

Python_DOTNET_LAUNCHER

Added in version 3.18.

The .Net interpreter. Only used by **IronPython** implementation.

Python_Development_FOUND

System has the Python development artifacts.

Python_Development.Module_FOUND

Added in version 3.18.

System has the Python development artifacts for Python module.

Python_Development.SABIModule_FOUND

Added in version 3.26.

System has the Python development artifacts for Python module using the Stable Application Binary Interface.

Python_Development.Embed_FOUND

Added in version 3.18.

System has the Python development artifacts for Python embedding.

Python_INCLUDE_DIRS

The Python include directories.

Python_DEFINITIONS

Added in version 3.30.3.

The Python preprocessor definitions.

Python_DEBUG_POSTFIX

Added in version 3.30.

Postfix of debug python module. This variable can be used to define the *DEBUG_POSTFIX* target property.

Python_LINK_OPTIONS

Added in version 3.19.

The Python link options. Some configurations require specific link options for a correct build and execution.

Python_LIBRARIES

The Python libraries.

Python_LIBRARY_DIRS

The Python library directories.

Python_RUNTIME_LIBRARY_DIRS

The Python runtime library directories.

Python_SABI_LIBRARIES

Added in version 3.26.

The Python libraries for the Stable Application Binary Interface.

Python_SABI_LIBRARY_DIRS

Added in version 3.26.

The Python **SABI** library directories.

Python_RUNTIME_SABI_LIBRARY_DIRS

Added in version 3.26.

The Python runtime **SABI** library directories.

Python_VERSION

Python version.

Python_VERSION_MAJOR

Python major version.

Python_VERSION_MINOR

Python minor version.

Python_VERSION_PATCH

Python patch version.

Python_PyPy_VERSION

Added in version 3.18.

Python PyPy version.

Python_NumPy_FOUND

Added in version 3.14.

System has the NumPy.

Python_NumPy_INCLUDE_DIRS

Added in version 3.14.

The NumPy include directories.

Python_NumPy_VERSION

Added in version 3.14.

The NumPy version.

Hints

Python_ROOT_DIR

Define the root directory of a Python installation.

Python_USE_STATIC_LIBS

- If not defined, search for shared libraries and static libraries in that order.
- If set to TRUE, search **only** for static libraries.
- If set to FALSE, search **only** for shared libraries.

NOTE:

This hint will be ignored on **Windows** because static libraries are not available on this platform.

Python_FIND_ABI

Added in version 3.16.

This variable defines which ABIs, as defined in *PEP 3149*, should be searched.

NOTE:

This hint will be honored only when searched for **Python** version 3.

The **Python_FIND_ABI** variable is a 4-tuple specifying, in that order, **pydebug** (**d**), **pymalloc** (**m**), **unicode** (**u**) and **gil_disabled** (**t**) flags.

Added in version 3.30: A fourth element, specifying the **gil_disabled** flag (i.e. free threaded python), is added and is optional. If not specified, the value is **OFF**.

Each element can be set to one of the following:

- **ON**: Corresponding flag is selected.
- **OFF**: Corresponding flag is not selected.
- **ANY**: The two possibilities (**ON** and **OFF**) will be searched.

NOTE:

If **Python3_FIND_ABI** is not defined, any ABI, excluding the **gil_disabled** flag, will be searched.

From this 4-tuple, various ABIs will be searched starting from the most specialized to the most general. Moreover, when **ANY** is specified for **pydebug** and **gil_disabled**, **debug** and **free threaded** versions will be searched **after non-debug** and **non-gil-disabled** ones.

For example, if we have:

```
set (Python_FIND_ABI "ON" "ANY" "ANY" "ON")
```

The following flags combinations will be appended, in that order, to the artifact names: **tdmu**, **tdm**, **tdu**, and **td**.

And to search any possible ABIs:

```
set (Python_FIND_ABI "ANY" "ANY" "ANY" "ANY")
```

The following combinations, in that order, will be used: **mu**, **m**, **u**, **<empty>**, **dmu**, **dm**, **du**, **d**, **tmu**, **tm**, **tu**, **t**, **tdmu**, **tdm**, **tdu**, and **td**.

NOTE:

This hint is useful only on **POSIX** systems except for the **gil_disabled** flag. So, on **Windows** systems, when **Python_FIND_ABI** is defined, **Python** distributions from *python.org* will be found only if the value for each flag is **OFF** or **ANY** except for the fourth one (**gil_disabled**).

Python_FIND_STRATEGY

Added in version 3.15.

This variable defines how lookup will be done. The **Python_FIND_STRATEGY** variable can be set to one of the following:

- **VERSION**: Try to find the most recent version in all specified locations. This is the default if policy *CMP0094* is undefined or set to **OLD**.
- **LOCATION**: Stops lookup as soon as a version satisfying version constraints is founded. This is the default if policy *CMP0094* is set to **NEW**.

See also **Python_FIND_UNVERSIONED_NAMES**.

Python_FIND_REGISTRY

Added in version 3.13.

On Windows the **Python_FIND_REGISTRY** variable determine the order of preference between registry and environment variables. the **Python_FIND_REGISTRY** variable can be set to one of the following:

- **FIRST**: Try to use registry before environment variables. This is the default.
- **LAST**: Try to use registry after environment variables.
- **NEVER**: Never try to use registry.

Python_FIND_FRAMEWORK

Added in version 3.15.

On macOS the **Python_FIND_FRAMEWORK** variable determine the order of preference between Apple-style and unix-style package components. This variable can take same values as *CMAKE_FIND_FRAMEWORK* variable.

NOTE:

Value **ONLY** is not supported so **FIRST** will be used instead.

If **Python_FIND_FRAMEWORK** is not defined, **CMAKE_FIND_FRAMEWORK** variable will be used, if any.

Python_FIND_VIRTUALENV

Added in version 3.15.

This variable defines the handling of virtual environments managed by **virtualenv** or **conda**. It is meaningful only when a virtual environment is active (i.e. the **activate** script has been evaluated). In this case, it takes precedence over **Python_FIND_REGISTRY** and **CMAKE_FIND_FRAMEWORK** variables. The **Python_FIND_VIRTUALENV** variable can be set to one of the following:

- **FIRST**: The virtual environment is used before any other standard paths to look-up for the interpreter. This is the default.
- **ONLY**: Only the virtual environment is used to look-up for the interpreter.
- **STANDARD**: The virtual environment is not used to look-up for the interpreter but environment variable **PATH** is always considered. In this case, variable **Python_FIND_REGISTRY** (Windows) or **CMAKE_FIND_FRAMEWORK** (macOS) can be set with value **LAST** or **NEVER** to select preferably the interpreter from the virtual environment.

Added in version 3.17: Added support for **conda** environments.

NOTE:

If the component **Development** is requested (or one of its sub-components) and is not found or the wrong artifacts are returned, including also the component **Interpreter** may be helpful.

Python_FIND_IMPLEMENTATIONS

Added in version 3.18.

This variable defines, in an ordered list, the different implementations which will be searched. The **Python_FIND_IMPLEMENTATIONS** variable can hold the following values:

- **CPython**: this is the standard implementation. Various products, like **Anaconda** or **ActivePython**, rely on this implementation.
- **IronPython**: This implementation use the **CSharp** language for **.NET Framework** on top of the *Dynamic Language Runtime (DLR)*. See *IronPython*.
- **PyPy**: This implementation use **RPython** language and **RPython translation toolchain** to produce the python interpreter. See *PyPy*.

The default value is:

- Windows platform: **CPython, IronPython**
- Other platforms: **CPython**

NOTE:

This hint has the lowest priority of all hints, so even if, for example, you specify **IronPython** first and **CPython** in second, a python product based on **CPython** can be selected because, for example with **Python_FIND_STRATEGY=LOCATION**, each location will be search first for **IronPython** and second for **CPython**.

NOTE:

When **IronPython** is specified, on platforms other than **Windows**, the **.Net** interpreter (i.e.

mono command) is expected to be available through the **PATH** variable.

Python_FIND_UNVERSIONED_NAMES

Added in version 3.20.

This variable defines how the generic names will be searched. Currently, it only applies to the generic names of the interpreter, namely, **python3** or **python2** and **python**. The **Python_FIND_UNVERSIONED_NAMES** variable can be set to one of the following values:

- **FIRST**: The generic names are searched before the more specialized ones (such as **python2.5** for example).
- **LAST**: The generic names are searched after the more specialized ones. This is the default.
- **NEVER**: The generic name are not searched at all.

See also **Python_FIND_STRATEGY**.

Artifacts Specification

Added in version 3.16.

To solve special cases, it is possible to specify directly the artifacts by setting the following variables:

Python_EXECUTABLE

The path to the interpreter.

Python_COMPILER

The path to the compiler.

Python_DOTNET_LAUNCHER

Added in version 3.18.

The **.Net** interpreter. Only used by **IronPython** implementation.

Python_LIBRARY

The path to the library. It will be used to compute the variables **Python_LIBRARIES**, **Python_LIBRARY_DIRS** and **Python_RUNTIME_LIBRARY_DIRS**.

Python_SABI_LIBRARY

Added in version 3.26.

The path to the library for Stable Application Binary Interface. It will be used to compute the variables **Python_SABI_LIBRARIES**, **Python_SABI_LIBRARY_DIRS** and **Python_RUNTIME_SABI_LIBRARY_DIRS**.

Python_INCLUDE_DIR

The path to the directory of the **Python** headers. It will be used to compute the variable **Python_INCLUDE_DIRS**.

Python_NumPy_INCLUDE_DIR

The path to the directory of the **NumPy** headers. It will be used to compute the variable **Python_NumPy_INCLUDE_DIRS**.

NOTE:

All paths must be absolute. Any artifact specified with a relative path will be ignored.

NOTE:

When an artifact is specified, all **HINTS** will be ignored and no search will be performed for this

artifact.

If more than one artifact is specified, it is the user's responsibility to ensure the consistency of the various artifacts.

By default, this module supports multiple calls in different directories of a project with different version/component requirements while providing correct and consistent results for each call. To support this behavior, CMake cache is not used in the traditional way which can be problematic for interactive specification. So, to enable also interactive specification, module behavior can be controlled with the following variable:

Python_ARTIFACTS_INTERACTIVE

Added in version 3.18.

Selects the behavior of the module. This is a boolean variable:

- If set to **TRUE**: Create CMake cache entries for the above artifact specification variables so that users can edit them interactively. This disables support for multiple version/component requirements.
- If set to **FALSE** or undefined: Enable multiple version/component requirements.

Python_ARTIFACTS_PREFIX

Added in version 4.0.

Define a custom prefix which will be used for the definition of all the result variables, targets, and commands. By using this variable, this module supports multiple calls in the same directory with different version/component requirements. For example, in case of cross-compilation, development components are needed but the native python interpreter can also be required:

```
find_package(Python COMPONENTS Development)
```

```
set(Python_ARTIFACTS_PREFIX "_HOST")
find_package(Python COMPONENTS Interpreter)
```

```
# Here Python_HOST_EXECUTABLE and Python_HOST::Interpreter artifacts are de
```

NOTE:

For consistency with standard behavior of modules, the various standard **_FOUND** variables (i.e. without the custom prefix) are also defined by each call to the *find_package()* command.

Commands

This module defines the command **Python_add_library** (when *CMAKE_ROLE* is **PROJECT**), which has the same semantics as *add_library()* and adds a dependency to target **Python::Python** or, when library type is **MODULE**, to target **Python::Module** or **Python::SABIModule** (when *USE_SABI* option is specified) and takes care of Python module naming rules:

```
Python_add_library (<name> [STATIC | SHARED | MODULE [USE_SABI <version>] [WITH_S
                    <source1> [<source2> ...]])
```

If the library type is not specified, **MODULE** is assumed.

Added in version 3.17: For **MODULE** library type, if option **WITH_SOABI** is specified, the module suffix will include the **Python_SOABI** value, if any.

Added in version 3.26: For **MODULE** type, if the option **USE_SABI** is specified, the preprocessor definition **Py_LIMITED_API** will be specified, as **PRIVATE**, for the target <name> with the value computed from <version> argument. The expected format for <version> is **major[.minor]**, where each component is a numeric value. If **minor** component is specified, the version should be, at least, **3.2** which is the version where the *Stable Application Binary Interface* was introduced. Specifying only major version **3** is equivalent to **3.2**.

When option **WITH_SOABI** is also specified, the module suffix will include the **Python_SOSABI** value, if any.

Added in version 3.30: For **MODULE** type, the **DEBUG_POSTFIX** target property is initialized with the value of **Python_DEBUG_POSTFIX** variable if defined.

FindPython2

Added in version 3.12.

Find Python 2 interpreter, compiler and development environment (include directories and libraries).

Added in version 3.19: When a version is requested, it can be specified as a simple value or as a range. For a detailed description of version range usage and capabilities, refer to the *find_package()* command.

The following components are supported:

- **Interpreter**: search for Python 2 interpreter
- **Compiler**: search for Python 2 compiler. Only offered by IronPython.
- **Development**: search for development artifacts (include directories and libraries).

Added in version 3.18: This component includes two sub-components which can be specified independently:

- **Development.Module**: search for artifacts for Python 2 module developments.
- **Development.Embed**: search for artifacts for Python 2 embedding developments.
- **NumPy**: search for NumPy include directories.

Added in version 3.14: Added the **NumPy** component.

If no **COMPONENTS** are specified, **Interpreter** is assumed.

If component **Development** is specified, it implies sub-components **Development.Module** and **Development.Embed**.

To ensure consistent versions between components **Interpreter**, **Compiler**, **Development** (or one of its sub-components) and **NumPy**, specify all components at the same time:

```
find_package (Python2 COMPONENTS Interpreter Development)
```

This module looks only for version 2 of Python. This module can be used concurrently with *FindPython3* module to use both Python versions.

The *FindPython* module can be used if Python version does not matter for you.

NOTE:

If components **Interpreter** and **Development** (or one of its sub-components) are both specified, this module search only for interpreter with same platform architecture as the one defined by CMake configuration. This constraint does not apply if only **Interpreter** component is specified.

Imported Targets

This module defines the following *Imported Targets*:

Changed in version 3.14: *Imported Targets* are only created when *CMAKE_ROLE* is **PROJECT**.

Python2::Interpreter

Python 2 interpreter. This target is defined only if the **Interpreter** component is found.

Python2::InterpreterDebug

Added in version 3.30.

Python 2 debug interpreter. This target is defined only if the **Interpreter** component is found and the **Python2_EXECUTABLE_DEBUG** variable is defined. The target is only defined on the **Windows** platform.

Python2::InterpreterMultiConfig

Added in version 3.30.

Python 2 interpreter. The release or debug version of the interpreter will be used, based on the context (platform, configuration). This target is defined only if the **Interpreter** component is found

Python2::Compiler

Python 2 compiler. This target is defined only if the **Compiler** component is found.

Python2::Module

Added in version 3.15.

Python 2 library for Python module. Target defined if component **Development.Module** is found.

Python2::Python

Python 2 library for Python embedding. Target defined if component **Development.Embed** is found.

Python2::NumPy

Added in version 3.14.

NumPy library for Python 2. Target defined if component **NumPy** is found.

Result Variables

This module will set the following variables in your project (see *Standard Variable Names*):

Python2_FOUND

System has the Python 2 requested components.

Python2_Interpreter_FOUND

System has the Python 2 interpreter.

Python2_EXECUTABLE

Path to the Python 2 interpreter.

Python2_EXECUTABLE_DEBUG

Added in version 3.30.

Path to the debug Python 2 interpreter. It is only defined on the **Windows** platform.

Python2_INTERPRETER

Added in version 3.30.

Path to the Python 2 interpreter, defined as a *generator expression* selecting the **Python2_EXECUTABLE** or **Python2_EXECUTABLE_DEBUG** variable based on the context (platform, configuration).

Python2_INTERPRETER_ID

A short string unique to the interpreter. Possible values include:

- Python
- ActivePython
- Anaconda
- Canopy
- IronPython
- PyPy

Python2_STDLIB

Standard platform independent installation directory.

Information returned by `sysconfig.get_path('stdlib')` or else `distutils.sysconfig.get_python_lib(plat_specific=False, standard_lib=True)`.

Python2_STDARCH

Standard platform dependent installation directory.

Information returned by `sysconfig.get_path('platstdlib')` or else `distutils.sysconfig.get_python_lib(plat_specific=True, standard_lib=True)`.

Python2_SITELIB

Third-party platform independent installation directory.

Information returned by `sysconfig.get_path('purelib')` or else `distutils.sysconfig.get_python_lib(plat_specific=False, standard_lib=False)`.

Python2_SITEARCH

Third-party platform dependent installation directory.

Information returned by `sysconfig.get_path('platlib')` or else `distutils.sysconfig.get_python_lib(plat_specific=True, standard_lib=False)`.

Python2_Compiler_FOUND

System has the Python 2 compiler.

Python2_COMPILER

Path to the Python 2 compiler. Only offered by IronPython.

Python2_COMPILER_ID

A short string unique to the compiler. Possible values include:

- IronPython

Python2_DOTNET_LAUNCHER

Added in version 3.18.

The .Net interpreter. Only used by **IronPython** implementation.

Python2_Development_FOUND

System has the Python 2 development artifacts.

Python2_Development.Module_FOUND

Added in version 3.18.

System has the Python 2 development artifacts for Python module.

Python2_Development.Embed_FOUND

Added in version 3.18.

System has the Python 2 development artifacts for Python embedding.

Python2_INCLUDE_DIRS

The Python 2 include directories.

Python2_DEBUG_POSTFIX

Added in version 3.30.

Postfix of debug python module. This variable can be used to define the *DEBUG_POSTFIX* target property.

Python2_LINK_OPTIONS

Added in version 3.19.

The Python 2 link options. Some configurations require specific link options for a correct build and execution.

Python2_LIBRARIES

The Python 2 libraries.

Python2_LIBRARY_DIRS

The Python 2 library directories.

Python2_RUNTIME_LIBRARY_DIRS

The Python 2 runtime library directories.

Python2_VERSION

Python 2 version.

Python2_VERSION_MAJOR

Python 2 major version.

Python2_VERSION_MINOR

Python 2 minor version.

Python2_VERSION_PATCH

Python 2 patch version.

Python2_PyPy_VERSION

Added in version 3.18.

Python 2 PyPy version.

Python2_NumPy_FOUND

Added in version 3.14.

System has the NumPy.

Python2_NumPy_INCLUDE_DIRS

Added in version 3.14.

The NumPy include directories.

Python2_NumPy_VERSION

Added in version 3.14.

The NumPy version.

Hints**Python2_ROOT_DIR**

Define the root directory of a Python 2 installation.

Python2_USE_STATIC_LIBS

- If not defined, search for shared libraries and static libraries in that order.
- If set to TRUE, search **only** for static libraries.
- If set to FALSE, search **only** for shared libraries.

NOTE:

This hint will be ignored on **Windows** because static libraries are not available on this platform.

Python2_FIND_STRATEGY

Added in version 3.15.

This variable defines how lookup will be done. The **Python2_FIND_STRATEGY** variable can be set to one of the following:

- **VERSION**: Try to find the most recent version in all specified locations. This is the default if policy *CMP0094* is undefined or set to **OLD**.
- **LOCATION**: Stops lookup as soon as a version satisfying version constraints is founded. This is the default if policy *CMP0094* is set to **NEW**.

See also **Python2_FIND_UNVERSIONED_NAMES**.

Python2_FIND_REGISTRY

Added in version 3.13.

On Windows the **Python2_FIND_REGISTRY** variable determine the order of preference between registry and environment variables. the **Python2_FIND_REGISTRY** variable can be set to one of the following:

- **FIRST**: Try to use registry before environment variables. This is the default.
- **LAST**: Try to use registry after environment variables.
- **NEVER**: Never try to use registry.

Python2_FIND_FRAMEWORK

Added in version 3.15.

On macOS the **Python2_FIND_FRAMEWORK** variable determine the order of preference between Apple-style and unix-style package components. This variable can take same values as **CMAKE_FIND_FRAMEWORK** variable.

NOTE:

Value **ONLY** is not supported so **FIRST** will be used instead.

If **Python2_FIND_FRAMEWORK** is not defined, **CMAKE_FIND_FRAMEWORK** variable will be used, if any.

Python2_FIND_VIRTUALENV

Added in version 3.15.

This variable defines the handling of virtual environments managed by **virtualenv** or **conda**. It is meaningful only when a virtual environment is active (i.e. the **activate** script has been evaluated). In this case, it takes precedence over **Python2_FIND_REGISTRY** and **CMAKE_FIND_FRAMEWORK** variables. The **Python2_FIND_VIRTUALENV** variable can be set to one of the following:

- **FIRST**: The virtual environment is used before any other standard paths to look-up for the interpreter. This is the default.
- **ONLY**: Only the virtual environment is used to look-up for the interpreter.
- **STANDARD**: The virtual environment is not used to look-up for the interpreter but environment variable **PATH** is always considered. In this case, variable **Python2_FIND_REGISTRY** (Windows) or **CMAKE_FIND_FRAMEWORK** (macOS) can be set with value **LAST** or **NEVER** to select preferably the interpreter from the virtual environment.

Added in version 3.17: Added support for **conda** environments.

NOTE:

If the component **Development** is requested (or one of its sub-components) and is not found or the wrong artifacts are returned, including also the component **Interpreter** may be helpful.

Python2_FIND_IMPLEMENTATIONS

Added in version 3.18.

This variable defines, in an ordered list, the different implementations which will be searched. The **Python2_FIND_IMPLEMENTATIONS** variable can hold the following values:

- **CPython**: this is the standard implementation. Various products, like **Anaconda** or **ActivePython**, rely on this implementation.
- **IronPython**: This implementation use the **CSharp** language for **.NET Framework** on top of the *Dynamic Language Runtime (DLR)*. See *IronPython*.

- **PyPy**: This implementation use **RPython** language and **RPython translation toolchain** to produce the python interpreter. See *PyPy* .

The default value is:

- Windows platform: **CPython**, **IronPython**
- Other platforms: **CPython**

NOTE:

This hint has the lowest priority of all hints, so even if, for example, you specify **IronPython** first and **CPython** in second, a python product based on **CPython** can be selected because, for example with **Python2_FIND_STRATEGY=LOCATION**, each location will be search first for **IronPython** and second for **CPython**.

NOTE:

When **IronPython** is specified, on platforms other than **Windows**, the **.Net** interpreter (i.e. **mono** command) is expected to be available through the **PATH** variable.

Python2_FIND_UNVERSIONED_NAMES

Added in version 3.20.

This variable defines how the generic names will be searched. Currently, it only applies to the generic names of the interpreter, namely, **python2** and **python**. The **Python2_FIND_UNVERSIONED_NAMES** variable can be set to one of the following values:

- **FIRST**: The generic names are searched before the more specialized ones (such as **python2.5** for example).
- **LAST**: The generic names are searched after the more specialized ones. This is the default.
- **NEVER**: The generic name are not searched at all.

See also **Python2_FIND_STRATEGY**.

Artifacts Specification

Added in version 3.16.

To solve special cases, it is possible to specify directly the artifacts by setting the following variables:

Python2_EXECUTABLE

The path to the interpreter.

Python2_COMPILER

The path to the compiler.

Python2_DOTNET_LAUNCHER

Added in version 3.18.

The **.Net** interpreter. Only used by **IronPython** implementation.

Python2_LIBRARY

The path to the library. It will be used to compute the variables **Python2_LIBRARIES**, **Python2_LIBRARY_DIRS** and **Python2_RUNTIME_LIBRARY_DIRS**.

Python2_INCLUDE_DIR

The path to the directory of the **Python** headers. It will be used to compute the variable **Python2_INCLUDE_DIRS**.

Python2_NumPy_INCLUDE_DIR

The path to the directory of the **NumPy** headers. It will be used to compute the variable **Python2_NumPy_INCLUDE_DIRS**.

NOTE:

All paths must be absolute. Any artifact specified with a relative path will be ignored.

NOTE:

When an artifact is specified, all **HINTS** will be ignored and no search will be performed for this artifact.

If more than one artifact is specified, it is the user's responsibility to ensure the consistency of the various artifacts.

By default, this module supports multiple calls in different directories of a project with different version/component requirements while providing correct and consistent results for each call. To support this behavior, CMake cache is not used in the traditional way which can be problematic for interactive specification. So, to enable also interactive specification, module behavior can be controlled with the following variable:

Python2_ARTIFACTS_INTERACTIVE

Added in version 3.18.

Selects the behavior of the module. This is a boolean variable:

- If set to **TRUE**: Create CMake cache entries for the above artifact specification variables so that users can edit them interactively. This disables support for multiple version/component requirements.
- If set to **FALSE** or undefined: Enable multiple version/component requirements.

Python2_ARTIFACTS_PREFIX

Added in version 4.0.

Define a custom prefix which will be used for the definition of all the result variables, targets, and commands. By using this variable, this module supports multiple calls in the same directory with different version/component requirements. For example, in case of cross-compilation, development components are needed but the native python interpreter can also be required:

```
find_package(Python2 COMPONENTS Development)
```

```
set(Python2_ARTIFACTS_PREFIX "_HOST")
find_package(Python2 COMPONENTS Interpreter)
```

```
# Here Python2_HOST_EXECUTABLE and Python2_HOST::Interpreter artifacts are
```

NOTE:

For consistency with standard behavior of modules, the various standard **_FOUND** variables (i.e. without the custom prefix) are also defined by each call to the *find_package()* command.

Commands

This module defines the command **Python2_add_library** (when *CMAKE_ROLE* is **PROJECT**), which has the same semantics as *add_library()* and adds a dependency to target **Python2::Python** or, when library type is **MODULE**, to target **Python2::Module** and takes care of Python module naming rules:

```
Python2_add_library (<name> [STATIC | SHARED | MODULE]
                    <source1> [<source2> ...])
```

If library type is not specified, **MODULE** is assumed.

Added in version 3.30: For **MODULE** type, the *DEBUG_POSTFIX* target property is initialized with the value of **Python2_DEBUG_POSTFIX** variable if defined.

FindPython3

Added in version 3.12.

Find Python 3 interpreter, compiler and development environment (include directories and libraries).

Added in version 3.19: When a version is requested, it can be specified as a simple value or as a range. For a detailed description of version range usage and capabilities, refer to the *find_package()* command.

The following components are supported:

- **Interpreter**: search for Python 3 interpreter
- **Compiler**: search for Python 3 compiler. Only offered by IronPython.
- **Development**: search for development artifacts (include directories and libraries).

Added in version 3.18: This component includes two sub-components which can be specified independently:

- **Development.Module**: search for artifacts for Python 3 module developments.
- **Development.Embed**: search for artifacts for Python 3 embedding developments.

Added in version 3.26:

- **Development.SABIModule**: search for artifacts for Python 3 module developments using the *Stable Application Binary Interface*. This component is available only for version **3.2** and upper.
- **NumPy**: search for NumPy include directories.

Added in version 3.14: Added the **NumPy** component.

If no **COMPONENTS** are specified, **Interpreter** is assumed.

If component **Development** is specified, it implies sub-components **Development.Module** and **Development.Embed**.

To ensure consistent versions between components **Interpreter**, **Compiler**, **Development** (or one of its sub-components) and **NumPy**, specify all components at the same time:

```
find_package (Python3 COMPONENTS Interpreter Development)
```

This module looks only for version 3 of Python. This module can be used concurrently with *FindPython2* module to use both Python versions.

The *FindPython* module can be used if Python version does not matter for you.

NOTE:

If components **Interpreter** and **Development** (or one of its sub-components) are both specified, this module search only for interpreter with same platform architecture as the one defined by CMake configuration. This constraint does not apply if only **Interpreter** component is specified.

Imported Targets

This module defines the following *Imported Targets*:

Changed in version 3.14: *Imported Targets* are only created when *CMAKE_ROLE* is **PROJECT**.

Python3::Interpreter

Python 3 interpreter. This target is defined only if the **Interpreter** component is found.

Python3::InterpreterDebug

Added in version 3.30.

Python 3 debug interpreter. This target is defined only if the **Interpreter** component is found and the **Python3_EXECUTABLE_DEBUG** variable is defined. The target is only defined on the **Windows** platform.

Python3::InterpreterMultiConfig

Added in version 3.30.

Python 3 interpreter. The release or debug version of the interpreter will be used, based on the context (platform, configuration). This target is defined only if the **Interpreter** component is found

Python3::Compiler

Python 3 compiler. This target is defined only if the **Compiler** component is found.

Python3::Module

Added in version 3.15.

Python 3 library for Python module. Target defined if component **Development.Module** is found.

Python3::SABIModule

Added in version 3.26.

Python 3 library for Python module using the Stable Application Binary Interface. Target defined if component **Development.SABIModule** is found.

Python3::Python

Python 3 library for Python embedding. Target defined if component **Development.Embed** is found.

Python3::NumPy

Added in version 3.14.

NumPy library for Python 3. Target defined if component **NumPy** is found.

Result Variables

This module will set the following variables in your project (see *Standard Variable Names*):

Python3_FOUND

System has the Python 3 requested components.

Python3_Interpreter_FOUND

System has the Python 3 interpreter.

Python3_EXECUTABLE

Path to the Python 3 interpreter.

Python3_EXECUTABLE_DEBUG

Added in version 3.30.

Path to the debug Python 3 interpreter. It is only defined on **Windows** platform.

Python3_INTERPRETER

Added in version 3.30.

Path to the Python 3 interpreter, defined as a *generator expression* selecting the **Python3_EXECUTABLE** or **Python3_EXECUTABLE_DEBUG** variable based on the context (platform, configuration).

Python3_INTERPRETER_ID

A short string unique to the interpreter. Possible values include:

- Python
- ActivePython
- Anaconda
- Canopy
- IronPython
- PyPy

Python3_STDLIB

Standard platform independent installation directory.

Information returned by `sysconfig.get_path('stdlib')`.

Python3_STDARCH

Standard platform dependent installation directory.

Information returned by `sysconfig.get_path('platstdlib')`.

Python3_SITELIB

Third-party platform independent installation directory.

Information returned by `sysconfig.get_path('purelib')`.

Python3_SITEARCH

Third-party platform dependent installation directory.

Information returned by `sysconfig.get_path('platlib')`.

Python3_SOABI

Added in version 3.17.

Extension suffix for modules.

Information computed from `sysconfig.get_config_var('EXT_SUFFIX')` or `sysconfig.get_config_var('SOABI')` or `python3-config --extension-suffix`.

Python3_SOSABI

Added in version 3.26.

Extension suffix for modules using the Stable Application Binary Interface.

Information computed from `importlib.machinery.EXTENSION_SUFFIXES` if the COMPONENT **Interpreter** was specified. Otherwise, the extension is **abi3** except for **Windows**, **MSYS** and **CYGWIN** for which this is an empty string.

Python3_Compiler_FOUND

System has the Python 3 compiler.

Python3_COMPILER

Path to the Python 3 compiler. Only offered by IronPython.

Python3_COMPILER_ID

A short string unique to the compiler. Possible values include:

- IronPython

Python3_DOTNET_LAUNCHER

Added in version 3.18.

The .Net interpreter. Only used by **IronPython** implementation.

Python3_Development_FOUND

System has the Python 3 development artifacts.

Python3_Development.Module_FOUND

Added in version 3.18.

System has the Python 3 development artifacts for Python module.

Python3_Development.SABIModule_FOUND

Added in version 3.26.

System has the Python 3 development artifacts for Python module using the Stable Application Binary Interface.

Python3_Development.Embed_FOUND

Added in version 3.18.

System has the Python 3 development artifacts for Python embedding.

Python3_INCLUDE_DIRS

The Python 3 include directories.

Python3_DEFINITIONS

Added in version 3.30.3.

The Python 3 preprocessor definitions.

Python3_DEBUG_POSTFIX

Added in version 3.30.

Postfix of debug python module. This variable can be used to define the *DEBUG_POSTFIX* target property.

Python3_LINK_OPTIONS

Added in version 3.19.

The Python 3 link options. Some configurations require specific link options for a correct build and execution.

Python3_LIBRARIES

The Python 3 libraries.

Python3_LIBRARY_DIRS

The Python 3 library directories.

Python3_RUNTIME_LIBRARY_DIRS

The Python 3 runtime library directories.

Python3_SABI_LIBRARIES

Added in version 3.26.

The Python 3 libraries for the Stable Application Binary Interface.

Python3_SABI_LIBRARY_DIRS

Added in version 3.26.

The Python 3 **SABI** library directories.

Python3_RUNTIME_SABI_LIBRARY_DIRS

Added in version 3.26.

The Python 3 runtime **SABI** library directories.

Python3_VERSION

Python 3 version.

Python3_VERSION_MAJOR

Python 3 major version.

Python3_VERSION_MINOR

Python 3 minor version.

Python3_VERSION_PATCH

Python 3 patch version.

Python3_PyPy_VERSION

Added in version 3.18.

Python 3 PyPy version.

Python3_NumPy_FOUND

Added in version 3.14.

System has the NumPy.

Python3_NumPy_INCLUDE_DIRS

Added in version 3.14.

The NumPy include directories.

Python3_NumPy_VERSION

Added in version 3.14.

The NumPy version.

Hints

Python3_ROOT_DIR

Define the root directory of a Python 3 installation.

Python3_USE_STATIC_LIBS

- If not defined, search for shared libraries and static libraries in that order.
- If set to TRUE, search **only** for static libraries.
- If set to FALSE, search **only** for shared libraries.

NOTE:

This hint will be ignored on **Windows** because static libraries are not available on this platform.

Python3_FIND_ABI

Added in version 3.16.

This variable defines which ABIs, as defined in *PEP 3149*, should be searched.

The **Python3_FIND_ABI** variable is a 4-tuple specifying, in that order, **pydebug** (**d**), **pymalloc** (**m**), **unicode** (**u**) and **gil_disabled** (**t**) flags.

Added in version 3.30: A fourth element, specifying the **gil_disabled** flag (i.e. free threaded python), is added and is optional. If not specified, the value is **OFF**.

Each element can be set to one of the following:

- **ON**: Corresponding flag is selected.
- **OFF**: Corresponding flag is not selected.
- **ANY**: The two possibilities (**ON** and **OFF**) will be searched.

NOTE:

If **Python3_FIND_ABI** is not defined, any ABI, excluding the **gil_disabled** flag, will be searched.

From this 4-tuple, various ABIs will be searched starting from the most specialized to the most general. Moreover, when **ANY** is specified for **pydebug** and **gil_disabled**, **debug** and **free threaded** versions will be searched **after non-debug** and **non-gil-disabled** ones.

For example, if we have:

```
set (Python3_FIND_ABI "ON" "ANY" "ANY" "ON")
```

The following flags combinations will be appended, in that order, to the artifact names: **tdmu**, **tdm**, **tdu**, and **td**.

And to search any possible ABIs:

```
set (Python3_FIND_ABI "ANY" "ANY" "ANY" "ANY")
```

The following combinations, in that order, will be used: **mu**, **m**, **u**, **<empty>**, **dmu**, **dm**, **du**, **d**, **tmu**, **tm**, **tu**, **t**, **tdmu**, **tdm**, **tdu**, and **td**.

NOTE:

This hint is useful only on **POSIX** systems except for the **gil_disabled** flag. So, on **Windows** systems, when **Python_FIND_ABI** is defined, **Python** distributions from *python.org* will be found only if the value for each flag is **OFF** or **ANY** except for the fourth one (**gil_disabled**).

Python3_FIND_STRATEGY

Added in version 3.15.

This variable defines how lookup will be done. The **Python3_FIND_STRATEGY** variable can be set to one of the following:

- **VERSION**: Try to find the most recent version in all specified locations. This is the default if policy *CMP0094* is undefined or set to **OLD**.
- **LOCATION**: Stops lookup as soon as a version satisfying version constraints is founded. This is the default if policy *CMP0094* is set to **NEW**.

See also **Python3_FIND_UNVERSIONED_NAMES**.

Python3_FIND_REGISTRY

Added in version 3.13.

On Windows the **Python3_FIND_REGISTRY** variable determine the order of preference between registry and environment variables. The **Python3_FIND_REGISTRY** variable can be set to one of the following:

- **FIRST**: Try to use registry before environment variables. This is the default.
- **LAST**: Try to use registry after environment variables.
- **NEVER**: Never try to use registry.

Python3_FIND_FRAMEWORK

Added in version 3.15.

On macOS the **Python3_FIND_FRAMEWORK** variable determine the order of preference between Apple-style and unix-style package components. This variable can take same values as *CMAKE_FIND_FRAMEWORK* variable.

NOTE:

Value **ONLY** is not supported so **FIRST** will be used instead.

If **Python3_FIND_FRAMEWORK** is not defined, *CMAKE_FIND_FRAMEWORK* variable will be used, if any.

Python3_FIND_VIRTUALENV

Added in version 3.15.

This variable defines the handling of virtual environments managed by **virtualenv** or **conda**. It is meaningful only when a virtual environment is active (i.e. the **activate** script has been evaluated). In this case, it takes precedence over **Python3_FIND_REGISTRY** and **CMAKE_FIND_FRAMEWORK** variables. The **Python3_FIND_VIRTUALENV** variable can be set to one of the following:

- **FIRST**: The virtual environment is used before any other standard paths to look-up for the interpreter. This is the default.
- **ONLY**: Only the virtual environment is used to look-up for the interpreter.
- **STANDARD**: The virtual environment is not used to look-up for the interpreter but environment variable **PATH** is always considered. In this case, variable **Python3_FIND_REGISTRY** (Windows) or **CMAKE_FIND_FRAMEWORK** (macOS) can be set with value **LAST** or **NEVER** to select preferably the interpreter from the virtual environment.

Added in version 3.17: Added support for **conda** environments.

NOTE:

If the component **Development** is requested (or one of its sub-components) and is not found or the wrong artifacts are returned, including also the component **Interpreter** may be helpful.

Python3_FIND_IMPLEMENTATIONS

Added in version 3.18.

This variable defines, in an ordered list, the different implementations which will be searched. The **Python3_FIND_IMPLEMENTATIONS** variable can hold the following values:

- **CPython**: this is the standard implementation. Various products, like **Anaconda** or **ActivePython**, rely on this implementation.
- **IronPython**: This implementation use the **CSharp** language for **.NET Framework** on top of the *Dynamic Language Runtime (DLR)*. See *IronPython*.
- **PyPy**: This implementation use **RPython** language and **RPython translation toolchain** to produce the python interpreter. See *PyPy*.

The default value is:

- Windows platform: **CPython**, **IronPython**
- Other platforms: **CPython**

NOTE:

This hint has the lowest priority of all hints, so even if, for example, you specify **IronPython** first and **CPython** in second, a python product based on **CPython** can be selected because, for example with **Python3_FIND_STRATEGY=LOCATION**, each location will be search first for **IronPython** and second for **CPython**.

NOTE:

When **IronPython** is specified, on platforms other than **Windows**, the **.Net** interpreter (i.e. **mono** command) is expected to be available through the **PATH** variable.

Python3_FIND_UNVERSIONED_NAMES

Added in version 3.20.

This variable defines how the generic names will be searched. Currently, it only applies to the generic names of the interpreter, namely, **python3** and **python**. The **Python3_FIND_UNVERSIONED_NAMES** variable can be set to one of the following values:

- **FIRST**: The generic names are searched before the more specialized ones (such as **python3.5** for example).
- **LAST**: The generic names are searched after the more specialized ones. This is the default.
- **NEVER**: The generic name are not searched at all.

See also **Python3_FIND_STRATEGY**.

Artifacts Specification

Added in version 3.16.

To solve special cases, it is possible to specify directly the artifacts by setting the following variables:

Python3_EXECUTABLE

The path to the interpreter.

Python3_COMPILER

The path to the compiler.

Python3_DOTNET_LAUNCHER

Added in version 3.18.

The .Net interpreter. Only used by **IronPython** implementation.

Python3_LIBRARY

The path to the library. It will be used to compute the variables **Python3_LIBRARIES**, **Python3_LIBRARY_DIRS** and **Python3_RUNTIME_LIBRARY_DIRS**.

Python3_SABI_LIBRARY

Added in version 3.26.

The path to the library for Stable Application Binary Interface. It will be used to compute the variables **Python3_SABI_LIBRARIES**, **Python3_SABI_LIBRARY_DIRS** and **Python3_RUNTIME_SABI_LIBRARY_DIRS**.

Python3_INCLUDE_DIR

The path to the directory of the **Python** headers. It will be used to compute the variable **Python3_INCLUDE_DIRS**.

Python3_NumPy_INCLUDE_DIR

The path to the directory of the **NumPy** headers. It will be used to compute the variable **Python3_NumPy_INCLUDE_DIRS**.

NOTE:

All paths must be absolute. Any artifact specified with a relative path will be ignored.

NOTE:

When an artifact is specified, all **HINTS** will be ignored and no search will be performed for this artifact.

If more than one artifact is specified, it is the user's responsibility to ensure the consistency of the various artifacts.

By default, this module supports multiple calls in different directories of a project with different version/component requirements while providing correct and consistent results for each call. To support this behavior, CMake cache is not used in the traditional way which can be problematic for interactive specification. So, to enable also interactive specification, module behavior can be controlled with the following variable:

Python3_ARTIFACTS_INTERACTIVE

Added in version 3.18.

Selects the behavior of the module. This is a boolean variable:

- If set to **TRUE**: Create CMake cache entries for the above artifact specification variables so that users can edit them interactively. This disables support for multiple version/component requirements.
- If set to **FALSE** or undefined: Enable multiple version/component requirements.

Python3_ARTIFACTS_PREFIX

Added in version 4.0.

Define a custom prefix which will be used for the definition of all the result variables, targets, and commands. By using this variable, this module supports multiple calls in the same directory with different version/component requirements. For example, in case of cross-compilation, development components are needed but the native python interpreter can also be required:

```
find_package(Python3 COMPONENTS Development)
```

```
set(Python3_ARTIFACTS_PREFIX "_HOST")
find_package(Python3 COMPONENTS Interpreter)
```

```
# Here Python3_HOST_EXECUTABLE and Python3_HOST::Interpreter artifacts are
```

NOTE:

For consistency with standard behavior of modules, the various standard **_FOUND** variables (i.e. without the custom prefix) are also defined by each call to the *find_package()* command.

Commands

This module defines the command **Python3_add_library** (when *CMAKE_ROLE* is **PROJECT**), which has the same semantics as *add_library()* and adds a dependency to target **Python3::Python** or, when library type is **MODULE**, to target **Python3::Module** or **Python3::SABIModule** (when **USE_SABI** option is specified) and takes care of Python module naming rules:

```
Python3_add_library (<name> [STATIC | SHARED | MODULE [USE_SABI <version>] [WITH_
<source1> [<source2> ...])
```

If the library type is not specified, **MODULE** is assumed.

Added in version 3.17: For **MODULE** library type, if option **WITH_SOABI** is specified, the module suffix will include the **Python3_SOABI** value, if any.

Added in version 3.26: For **MODULE** type, if the option **USE_SABI** is specified, the preprocessor

definition **Py_LIMITED_API** will be specified, as **PRIVATE**, for the target **<name>** with the value computed from **<version>** argument. The expected format for **<version>** is **major[.minor]**, where each component is a numeric value. If **minor** component is specified, the version should be, at least, **3.2** which is the version where the *Stable Application Binary Interface* was introduced. Specifying only major version **3** is equivalent to **3.2**.

When option **WITH_SOABI** is also specified, the module suffix will include the **Python3_SOSABI** value, if any.

Added in version 3.30: For **MODULE** type, the **DEBUG_POSTFIX** target property is initialized with the value of **Python3_DEBUG_POSTFIX** variable if defined.

FindQt3

Locate Qt include paths and libraries

This module defines:

```

QT_INCLUDE_DIR    - where to find qt.h, etc.
QT_LIBRARIES      - the libraries to link against to use Qt.
QT_DEFINITIONS    - definitions to use when
                    compiling code that uses Qt.
QT_FOUND          - If false, don't try to use Qt.
QT_VERSION_STRING - the version of Qt found

```

If you need the multithreaded version of Qt, set **QT_MT_REQUIRED** to **TRUE**

Also defined, but not for general use are:

```

QT_MOC_EXECUTABLE, where to find the moc tool.
QT_UIC_EXECUTABLE, where to find the uic tool.
QT_QT_LIBRARY, where to find the Qt library.
QT_QTMAIN_LIBRARY, where to find the qtmain
library. This is only required by Qt3 on Windows.

```

FindQt4

Finding and Using Qt4

This module can be used to find Qt4. The most important issue is that the Qt4 qmake is available via the system path. This qmake is then used to detect basically everything else. This module defines a number of *IMPORTED* targets, macros and variables.

Typical usage could be something like:

```

set(CMAKE_AUTOMOC ON)
set(CMAKE_INCLUDE_CURRENT_DIR ON)
find_package(Qt4 4.4.3 REQUIRED QtGui QtXml)
add_executable(myexe main.cpp)
target_link_libraries(myexe Qt4::QtGui Qt4::QtXml)

```

NOTE:

When using *IMPORTED* targets, the qtmain.lib static library is automatically linked on Windows for WIN32 executables. To disable that globally, set the **QT4_NO_LINK_QTMAIN** variable before finding Qt4. To disable that for a particular executable, set the **QT4_NO_LINK_QTMAIN** target property to **TRUE** on the executable.

Qt Build Tools

Qt relies on some bundled tools for code generation, such as **moc** for meta-object code generation, **uic** for widget layout and population, and **rcc** for virtual filesystem content generation. These tools may be automatically invoked by *cmake(1)* if the appropriate conditions are met. See *cmake-qt(7)* for more.

Qt Macros

In some cases it can be necessary or useful to invoke the Qt build tools in a more-manual way. Several macros are available to add targets for such uses.

```
macro QT4_WRAP_CPP(outfiles inputfile ... [TARGET tgt] OPTIONS ...)
    create moc code from a list of files containing Qt class with
    the Q_OBJECT declaration. Per-directory preprocessor definitions
    are also added. If the <tgt> is specified, the
    INTERFACE_INCLUDE_DIRECTORIES and INTERFACE_COMPILE_DEFINITIONS from
    the <tgt> are passed to moc. Options may be given to moc, such as
    those found when executing "moc -help".
```

```
macro QT4_WRAP_UI(outfiles inputfile ... OPTIONS ...)
    create code from a list of Qt designer ui files.
    Options may be given to uic, such as those found
    when executing "uic -help"
```

```
macro QT4_ADD_RESOURCES(outfiles inputfile ... OPTIONS ...)
    create code from a list of Qt resource files.
    Options may be given to rcc, such as those found
    when executing "rcc -help"
```

```
macro QT4_GENERATE_MOC(inputfile outputfile [TARGET tgt])
    creates a rule to run moc on infile and create outfile.
    Use this if for some reason QT4_WRAP_CPP() isn't appropriate, e.g.
    because you need a custom filename for the moc file or something
    similar. If the <tgt> is specified, the
    INTERFACE_INCLUDE_DIRECTORIES and INTERFACE_COMPILE_DEFINITIONS from
    the <tgt> are passed to moc.
```

```
macro QT4_ADD_DBUS_INTERFACE(outfiles interface basename)
    Create the interface header and implementation files with the
    given basename from the given interface xml file and add it to
    the list of sources.
```

You can pass additional parameters to the qdbusxml2cpp call by setting properties on the input file:

INCLUDE the given file will be included in the generate interface header

CLASSNAME the generated class is named accordingly

NO_NAMESPACE the generated class is not wrapped in a namespace

```
macro QT4_ADD_DBUS_INTERFACES(outfiles inputfile ... )
    Create the interface header and implementation files
    for all listed interface xml files.
    The basename will be automatically determined from the name
    of the xml file.
```

The source file properties described for QT4_ADD_DBUS_INTERFACE also apply here.

```
macro QT4_ADD_DBUS_ADAPTOR(outfiles xmlfile parentheader parentclassname
                           [basename] [classname])
    create a dbus adaptor (header and implementation file) from the xml file
    describing the interface, and add it to the list of sources. The adaptor
    forwards the calls to a parent class, defined in parentheader and named
    parentclassname. The name of the generated files will be
    <basename>adaptor.{cpp,h} where basename defaults to the basename of the
    xml file.
    If <classname> is provided, then it will be used as the classname of the
    adaptor itself.
```

```
macro QT4_GENERATE_DBUS_INTERFACE( header [interfacename] OPTIONS ...)
    generate the xml interface file from the given header.
    If the optional argument interfacename is omitted, the name of the
    interface file is constructed from the basename of the header with
    the suffix .xml appended.
    Options may be given to qdbuscpp2xml, such as those found when
    executing "qdbuscpp2xml --help"
```

```
macro QT4_CREATE_TRANSLATION( qm_files directories ... sources ...
                              ts_files ... OPTIONS ...)
    out: qm_files
    in:  directories sources ts_files
    options: flags to pass to lupdate, such as -extensions to specify
    extensions for a directory scan.
    generates commands to create .ts (via lupdate) and .qm
    (via lrelease) - files from directories and/or sources. The ts files are
    created and/or updated in the source tree (unless given with full paths).
    The qm files are generated in the build tree.
    Updating the translations can be done by adding the qm_files
    to the source list of your library/executable, so they are
    always updated, or by adding a custom target to control when
    they get updated/generated.
```

```
macro QT4_ADD_TRANSLATION( qm_files ts_files ... )
    out: qm_files
    in:  ts_files
    generates commands to create .qm from .ts - files. The generated
    filenames can be found in qm_files. The ts_files
    must exist and are not updated in any way.
```

```
macro QT4_AUTOMOC(sourcefile1 sourcefile2 ... [TARGET tgt])
    The qt4_automoc macro is obsolete. Use the CMAKE_AUTOMOC feature instead.
    This macro is still experimental.
    It can be used to have moc automatically handled.
    So if you have the files foo.h and foo.cpp, and in foo.h a
    class uses the Q_OBJECT macro, moc has to run on it. If you don't
    want to use QT4_WRAP_CPP() (which is reliable and mature), you can insert
    #include "foo.moc"
    in foo.cpp and then give foo.cpp as argument to QT4_AUTOMOC(). This will
    scan all listed files at cmake-time for such included moc files and if it
```

finds them cause a rule to be generated to run moc at build time on the accompanying header file foo.h.
 If a source file has the SKIP_AUTOMOC property set it will be ignored by this macro.
 If the <tgt> is specified, the INTERFACE_INCLUDE_DIRECTORIES and INTERFACE_COMPILE_DEFINITIONS from the <tgt> are passed to moc.

function QT4_USE_MODULES(target [link_type] modules...)

This function is obsolete. Use target_link_libraries with IMPORTED targets instead.

Make <target> use the <modules> from Qt. Using a Qt module means to link to the library, add the relevant include directories for the module, and add the relevant compiler defines for using the module. Modules are roughly equivalent to components of Qt4, so usage would be something like:

```
qt4_use_modules(myexe Core Gui Declarative)
to use QtCore, QtGui and QtDeclarative. The optional <link_type> argument
can be specified as either LINK_PUBLIC or LINK_PRIVATE to specify the
same argument to the target_link_libraries call.
```

Imported Targets

A particular Qt library may be used by using the corresponding *IMPORTED* target with the *target_link_libraries()* command:

```
target_link_libraries(myexe Qt4::QtGui Qt4::QtXml)
```

Using a target in this way causes :cmake(1) to use the appropriate include directories and compile definitions for the target when compiling **myexe**.

Targets are aware of their dependencies, so for example it is not necessary to list **Qt4::QtCore** if another Qt library is listed, and it is not necessary to list **Qt4::QtGui** if **Qt4::QtDeclarative** is listed. Targets may be tested for existence in the usual way with the *if(TARGET)* command.

The Qt toolkit may contain both debug and release libraries. *cmak e(1)* will choose the appropriate version based on the build configuration.

Qt4::QtCore

The QtCore target

Qt4::QtGui

The QtGui target

Qt4::Qt3Support

The Qt3Support target

Qt4::QtAssistant

The QtAssistant target

Qt4::QtAssistantClient

The QtAssistantClient target

Qt4::QAxContainer

The QAxContainer target (Windows only)

Qt4::QAxServer

The QAxServer target (Windows only)

Qt4::QtDBus

The QtDBus target

Qt4::QtDeclarative

The QtDeclarative target

Qt4::QtDesigner

The QtDesigner target

Qt4::QtDesignerComponents

The QtDesignerComponents target

Qt4::QtHelp

The QtHelp target

Qt4::QtMotif

The QtMotif target

Qt4::QtMultimedia

The QtMultimedia target

Qt4::QtNetwork

The QtNetwork target

Qt4::QtNsPlugin

The QtNsPlugin target

Qt4::QtOpenGL

The QtOpenGL target

Qt4::QtScript

The QtScript target

Qt4::QtScriptTools

The QtScriptTools target

Qt4::QtSql

The QSql target

Qt4::QtSvg

The QtSvg target

Qt4::QtTest

The QtTest target

Qt4::QtUiTools

The QtUiTools target

Qt4::QtWebKit

The QtWebKit target

Qt4::QtXml

The QtXml target

Qt4::QtXmlPatterns

The QtXmlPatterns target

Qt4::phonon

The phonon target

Result Variables

Below is a detailed list of variables that FindQt4.cmake sets.

Qt4_FOUND

If false, don't try to use Qt 4.

QT_FOUND

If false, don't try to use Qt. This variable is for compatibility only.

QT4_FOUND

If false, don't try to use Qt 4. This variable is for compatibility only.

QT_VERSION_MAJOR

The major version of Qt found.

QT_VERSION_MINOR

The minor version of Qt found.

QT_VERSION_PATCH

The patch version of Qt found.

FindQuickTime

Locate QuickTime This module defines:

QUICKTIME_LIBRARY**QUICKTIME_FOUND**

if false, do not try to link to gdal

QUICKTIME_INCLUDE_DIR

where to find the headers

\$QUICKTIME_DIR is an environment variable that would correspond to:

```
./configure --prefix=$QUICKTIME_DIR
```

FindRTI

Try to find M&S HLA RTI libraries

This module finds if any HLA RTI is installed and locates the standard RTI include files and libraries.

RTI is a simulation infrastructure standardized by IEEE and SISO. It has a well defined C++ API that assures that simulation applications are independent on a particular RTI implementation.

[http://en.wikipedia.org/wiki/Run-Time_Infrastructure_\(simulation\)](http://en.wikipedia.org/wiki/Run-Time_Infrastructure_(simulation))

This code sets the following variables:

```
RTI_INCLUDE_DIR = the directory where RTI includes file are found
RTI_LIBRARIES = The libraries to link against to use RTI
RTI_DEFINITIONS = -DRTI_USES_STD_FSTREAM
RTI_FOUND = Set to FALSE if any HLA RTI was not found
```

Report problems to <certi-devel@nongnu.org>

FindRuby

This module determines if Ruby is installed and finds the locations of its include files and libraries. Ruby 1.8 through 3.4 are supported.

The minimum required version of Ruby can be specified using the standard syntax, e.g.

```
find_package(Ruby 3.2.6 EXACT REQUIRED)
# OR
find_package(Ruby 3.2)
```

Virtual environments, such as RVM or RBENV, are supported.

Result Variables

This module will set the following variables in your project:

Ruby_FOUND

set to true if ruby was found successfully

Ruby_EXECUTABLE

full path to the ruby binary

Ruby_INCLUDE_DIRS

include dirs to be used when using the ruby library

Ruby_LIBRARIES

Added in version 3.18: libraries needed to use ruby from C.

Ruby_VERSION

the version of ruby which was found, e.g. "3.2.6"

Ruby_VERSION_MAJOR

Ruby major version.

Ruby_VERSION_MINOR

Ruby minor version.

Ruby_VERSION_PATCH

Ruby patch version.

Changed in version 3.18: Previous versions of CMake used the **RUBY_** prefix for all variables.

Deprecated since version 4.0: The following variables are deprecated. See policy *CMP0185*.

RUBY_EXECUTABLE

same as **Ruby_EXECUTABLE**.

RUBY_INCLUDE_DIRS

same as **Ruby_INCLUDE_DIRS**.

RUBY_INCLUDE_PATH

same as **Ruby_INCLUDE_DIRS**.

RUBY_LIBRARY

same as **Ruby_LIBRARY**.

RUBY_VERSION

same as **Ruby_VERSION**.

RUBY_FOUND

same as **Ruby_FOUND**.

Hints**Ruby_FIND_VIRTUALENV**

Added in version 3.18.

This variable defines the handling of virtual environments. It can be left empty or be set to one of the following values:

-

FIRST: Virtual Ruby environments are searched for first,
then the system Ruby installation. This is the default.

- **ONLY:** Only virtual environments are searched
- **STANDARD:** Only the system Ruby installation is searched.

Virtual environments may be provided by:

- rvn** Requires that the **MY_RUBY_HOME** environment environment is defined.
- rbenv** Requires that **rbenv** is installed in `~/.rbenv/bin` or that the **RBENV_ROOT** environment variable is defined.

FindSDL

Locate the SDL library

Imported Targets

Added in version 3.19.

This module defines the following *IMPORTED* target:

SDL::SDL

The SDL library, if found

Result variables

This module will set the following variables in your project:

SDL_INCLUDE_DIRS

where to find SDL.h

SDL_LIBRARIES

the name of the library to link against

SDL_FOUND

if false, do not try to link to SDL

SDL_VERSION

the human-readable string containing the version of SDL if found

SDL_VERSION_MAJOR

SDL major version

SDL_VERSION_MINOR

SDL minor version

SDL_VERSION_PATCH

SDL patch version

Added in version 3.19: Added the **SDL_INCLUDE_DIRS**, **SDL_LIBRARIES** and **SDL_VERSION[_<PART>]** variables.

Cache variables

These variables may optionally be set to help this module find the correct files:

SDL_INCLUDE_DIR

where to find SDL.h

SDL_LIBRARY

the name of the library to link against

Variables for locating SDL

This module responds to the flag:

SDL_BUILDING_LIBRARY

If this is defined, then no `SDL_main` will be linked in because only applications need `main()`. Otherwise, it is assumed you are building an application and this module will attempt to locate and set

the proper link flags as part of the returned `SDL_LIBRARY` variable.

Obsolete variables

Deprecated since version 3.19.

These variables are obsolete and provided for backwards compatibility:

SDL_VERSION_STRING

the human-readable string containing the version of SDL if found. Identical to `SDL_VERSION`

Don't forget to include `SDLmain.h` and `SDLmain.m` your project for the OS X framework based version. (Other versions link to `-lSDLmain` which this module will try to find on your behalf.) Also for OS X, this module will automatically add the `-framework Cocoa` on your behalf.

Additional Note: If you see an empty `SDL_LIBRARY_TEMP` in your configuration and no `SDL_LIBRARY`, it means CMake did not find your SDL library (`SDL.dll`, `libsdl.so`, `SDL.framework`, etc). Set `SDL_LIBRARY_TEMP` to point to your SDL library, and configure again. Similarly, if you see an empty `SDLMAIN_LIBRARY`, you should set this value as appropriate. These values are used to generate the final `SDL_LIBRARY` variable, but when these values are unset, `SDL_LIBRARY` does not get created.

`$SDLDIR` is an environment variable that would correspond to the `./configure --prefix=$SDLDIR` used in building SDL. I.e.galup 9-20-02

On OSX, this will prefer the Framework version (if found) over others. People will have to manually change the cache values of `SDL_LIBRARY` to override this selection or set the CMake environment `CMAKE_INCLUDE_PATH` to modify the search paths.

Note that the header path has changed from `SDL/SDL.h` to just `SDL.h`. This needed to change because "proper" SDL convention is `#include "SDL.h"`, not `<SDL/SDL.h>`. This is done for portability reasons because not all systems place things in `SDL/` (see FreeBSD).

FindSDL_image

Locate `SDL_image` library

This module defines:

`SDL_IMAGE_LIBRARIES`, the name of the library to link against
`SDL_IMAGE_INCLUDE_DIRS`, where to find the headers
`SDL_IMAGE_FOUND`, if false, do not try to link against
`SDL_IMAGE_VERSION_STRING` - human-readable string containing the version of `SDL_image`

For backward compatibility the following variables are also set:

`SDLIMAGE_LIBRARY` (same value as `SDL_IMAGE_LIBRARIES`)
`SDLIMAGE_INCLUDE_DIR` (same value as `SDL_IMAGE_INCLUDE_DIRS`)
`SDLIMAGE_FOUND` (same value as `SDL_IMAGE_FOUND`)

`$SDLDIR` is an environment variable that would correspond to the `./configure --prefix=$SDLDIR` used in building SDL.

FindSDL_gfx

Added in version 3.25.

Locate `SDL_gfx` library

This module defines:

SDL::SDL_gfx, the name of the target to use with target_*() commands
 SDL_GFX_LIBRARIES, the name of the library to link against
 SDL_GFX_INCLUDE_DIRS, where to find the headers
 SDL_GFX_FOUND, if false, do not try to link against
 SDL_GFX_VERSION_STRING – human-readable string containing the
 version of SDL_gfx

\$SDLDIR is an environment variable that would correspond to the `./configure --prefix=$SDLDIR` used in building SDL.

FindSDL_mixer

Locate SDL_mixer library

This module defines:

SDL_MIXER_LIBRARIES, the name of the library to link against
 SDL_MIXER_INCLUDE_DIRS, where to find the headers
 SDL_MIXER_FOUND, if false, do not try to link against
 SDL_MIXER_VERSION_STRING – human-readable string containing the
 version of SDL_mixer

For backward compatibility the following variables are also set:

SDLMIXER_LIBRARY (same value as SDL_MIXER_LIBRARIES)
 SDLMIXER_INCLUDE_DIR (same value as SDL_MIXER_INCLUDE_DIRS)
 SDLMIXER_FOUND (same value as SDL_MIXER_FOUND)

\$SDLDIR is an environment variable that would correspond to the `./configure --prefix=$SDLDIR` used in building SDL.

FindSDL_net

Locate SDL_net library

This module defines:

SDL_NET_LIBRARIES, the name of the library to link against
 SDL_NET_INCLUDE_DIRS, where to find the headers
 SDL_NET_FOUND, if false, do not try to link against
 SDL_NET_VERSION_STRING – human-readable string containing the version of SDL_net

For backward compatibility the following variables are also set:

SDLNET_LIBRARY (same value as SDL_NET_LIBRARIES)
 SDLNET_INCLUDE_DIR (same value as SDL_NET_INCLUDE_DIRS)
 SDLNET_FOUND (same value as SDL_NET_FOUND)

\$SDLDIR is an environment variable that would correspond to the `./configure --prefix=$SDLDIR` used in building SDL.

FindSDL_sound

Locates the SDL_sound library

This module depends on SDL being found and must be called AFTER FindSDL.cmake is called.

This module defines

SDL_SOUND_INCLUDE_DIR, where to find SDL_sound.h
 SDL_SOUND_FOUND, if false, do not try to link to SDL_sound
 SDL_SOUND_LIBRARIES, this contains the list of libraries that you need to link against.
 SDL_SOUND_EXTRAS, this is an optional variable for you to add your own flags to SDL_SOUND_LIBRARIES. This is prepended to SDL_SOUND_LIBRARIES. This is available mostly for cases this module failed to anticipate for and you must add additional flags. This is marked as ADVANCED.
 SDL_SOUND_VERSION_STRING, human-readable string containing the version of SDL_sound

This module also defines (but you shouldn't need to use directly)

SDL_SOUND_LIBRARY, the name of just the SDL_sound library you would link against. Use SDL_SOUND_LIBRARIES for you link instructions and not this one.

And might define the following as needed

MIKMOD_LIBRARY
 MODPLUG_LIBRARY
 OGG_LIBRARY
 VORBIS_LIBRARY
 SMPEG_LIBRARY
 FLAC_LIBRARY
 SPEEX_LIBRARY

Typically, you should not use these variables directly, and you should use SDL_SOUND_LIBRARIES which contains SDL_SOUND_LIBRARY and the other audio libraries (if needed) to successfully compile on your system.

Responds to the \$SDLDIR and \$SDLSOUNDDIR environmental variable that would correspond to the ./configure --prefix=\$SDLDIR used in building SDL.

On OSX, this will prefer the Framework version (if found) over others. People will have to manually change the cache values of SDL_LIBRARY to override this selection or set the CMake environment CMAKE_INCLUDE_PATH to modify the search paths.

FindSDL_ttf

Locate SDL_ttf library

This module defines:

SDL_TTF_LIBRARIES, the name of the library to link against
 SDL_TTF_INCLUDE_DIRS, where to find the headers
 SDL_TTF_FOUND, if false, do not try to link against
 SDL_TTF_VERSION_STRING - human-readable string containing the version of SDL_ttf

For backward compatibility the following variables are also set:

SDLTTF_LIBRARY (same value as SDL_TTF_LIBRARIES)
 SDLTTF_INCLUDE_DIR (same value as SDL_TTF_INCLUDE_DIRS)
 SDLTTF_FOUND (same value as SDL_TTF_FOUND)

\$SDLDIR is an environment variable that would correspond to the `./configure --prefix=$SDLDIR` used in building SDL.

FindSelfPackers

Find upx

This module looks for some executable packers (i.e. software that compress executables or shared libs into on-the-fly self-extracting executables or shared libs. Examples:

UPX: <http://wildsau.idv.uni-linz.ac.at/mfx/upx.html>

FindSquish

-- Typical Use

This module can be used to find Squish.

SQUISH_FOUND	If false, don't try to use Squish
SQUISH_VERSION	The full version of Squish found
SQUISH_VERSION_MAJOR	The major version of Squish found
SQUISH_VERSION_MINOR	The minor version of Squish found
SQUISH_VERSION_PATCH	The patch version of Squish found
SQUISH_INSTALL_DIR	The Squish installation directory (containing bin, lib, etc)
SQUISH_SERVER_EXECUTABLE	The squishserver executable
SQUISH_CLIENT_EXECUTABLE	The squishrunner executable
SQUISH_INSTALL_DIR_FOUND	Was the install directory found?
SQUISH_SERVER_EXECUTABLE_FOUND	Was the server executable found?
SQUISH_CLIENT_EXECUTABLE_FOUND	Was the client executable found?

It provides the function `squish_add_test()` for adding a squish test to cmake using Squish $\geq 4.x$:

```
squish_add_test(cmakeTestName
  AUT targetName SUITE suiteName TEST squishTestName
  [SETTINGSGROUP group] [PRE_COMMAND command] [POST_COMMAND command] )
```

Changed in version 3.18: In previous CMake versions, this function was named **squish_v4_add_test**.

The arguments have the following meaning:

cmakeTestName

this will be used as the first argument for `add_test()`

AUT targetName

the name of the cmake target which will be used as AUT, i.e. the executable which will be tested.

SUITE suiteName

this is either the full path to the squish suite, or just the last directory of the suite, i.e. the suite name. In this case the `CMakeLists.txt` which calls `squish_add_test()` must be located in the parent directory of the suite directory.

TEST squishTestName

the name of the squish test, i.e. the name of the subdirectory of the test inside the suite directory.

SETTINGSGROUP group

deprecated, this argument will be ignored.

PRE_COMMAND command

if specified, the given command will be executed before starting the squish test.

POST_COMMAND command

same as PRE_COMMAND, but after the squish test has been executed.

```
enable_testing()
find_package(Squish 6.5)
if (SQUISH_FOUND)
    squish_add_test(myTestName
        AUT myApp
        SUITE ${CMAKE_SOURCE_DIR}/tests/mySuite
        TEST someSquishTest
    )
endif ()
```

For users of Squish version 3.x the macro `squish_v3_add_test()` is provided:

```
squish_v3_add_test(testName applicationUnderTest testCase envVars testWrapper)
Use this macro to add a test using Squish 3.x.
```

```
enable_testing()
find_package(Squish 3.0)
if (SQUISH_FOUND)
    squish_v3_add_test(myTestName myApplication testCase envVars testWrapper)
endif ()
```

FindSQLite3

Added in version 3.14.

Find the SQLite libraries, v3

Imported Targets

This module defines the following *IMPORTED* target:

SQLite::SQLite3**Result variables**

This module will set the following variables if found:

SQLite3_INCLUDE_DIRS

where to find `sqlite3.h`, etc.

SQLite3_LIBRARIES

the libraries to link against to use SQLite3.

SQLite3_VERSION

version of the SQLite3 library found

SQLite3_FOUND

TRUE if found

FindSubversion

Extract information from a subversion working copy

The module defines the following variables:

```
Subversion_SVN_EXECUTABLE - path to svn command line client
Subversion_VERSION_SVN - version of svn command line client
```

Subversion_FOUND - true if the command line client was found
 SUBVERSION_FOUND - same as Subversion_FOUND, set for compatibility reasons

The minimum required version of Subversion can be specified using the standard syntax, e.g. **find_package(Subversion 1.4)**.

If the command line client executable is found two macros are defined:

```
Subversion_WC_INFO(<dir> <var-prefix> [IGNORE_SVN_FAILURE])
Subversion_WC_LOG(<dir> <var-prefix>)
```

Subversion_WC_INFO extracts information of a subversion working copy at a given location. This macro defines the following variables if running Subversion's **info** command on **<dir>** succeeds; otherwise a **SEND_ERROR** message is generated.

Added in version 3.13: The error can be ignored by providing the **IGNORE_SVN_FAILURE** option, which causes these variables to remain undefined.

```
<var-prefix>_WC_URL - url of the repository (at <dir>)
<var-prefix>_WC_ROOT - root url of the repository
<var-prefix>_WC_REVISION - current revision
<var-prefix>_WC_LAST_CHANGED_AUTHOR - author of last commit
<var-prefix>_WC_LAST_CHANGED_DATE - date of last commit
<var-prefix>_WC_LAST_CHANGED_REV - revision of last commit
<var-prefix>_WC_INFO - output of command `svn info <dir>'
```

Subversion_WC_LOG retrieves the log message of the base revision of a subversion working copy at a given location. This macro defines the variable:

```
<var-prefix>_LAST_CHANGED_LOG - last log of base revision
```

Example usage:

```
find_package(Subversion)
if(SUBVERSION_FOUND)
  Subversion_WC_INFO(${PROJECT_SOURCE_DIR} Project)
  message("Current revision is ${Project_WC_REVISION}")
  Subversion_WC_LOG(${PROJECT_SOURCE_DIR} Project)
  message("Last changed log is ${Project_LAST_CHANGED_LOG}")
endif()
```

FindSWIG

Find the Simplified Wrapper and Interface Generator (SWIG) executable.

This module finds an installed SWIG and determines its version.

Added in version 3.18: If a **COMPONENTS** or **OPTIONAL_COMPONENTS** argument is given to the *find_package()* command, it will also determine supported target languages.

Added in version 3.19: When a version is requested, it can be specified as a simple value or as a range. For a detailed description of version range usage and capabilities, refer to the *find_package()* command.

The module defines the following variables:

SWIG_FOUND

Whether SWIG and any required components were found on the system.

SWIG_EXECUTABLE

Path to the SWIG executable.

SWIG_DIR

Path to the installed SWIG **Lib** directory (result of **swig -swiglib**).

SWIG_VERSION

SWIG executable version (result of **swig -version**).

SWIG_<lang>_FOUND

If **COMPONENTS** or **OPTIONAL_COMPONENTS** are requested, each available target language <lang> (lowercase) will be set to TRUE.

Any **COMPONENTS** given to **find_package** should be the names of supported target languages as provided to the LANGUAGE argument of **swig_add_library**, such as **python** or **perl5**. Language names *must* be lowercase.

All information is collected from the **SWIG_EXECUTABLE**, so the version to be found can be changed from the command line by means of setting **SWIG_EXECUTABLE**.

Example usage requiring SWIG 4.0 or higher and Python language support, with optional Fortran support:

```
find_package(SWIG 4.0 COMPONENTS python OPTIONAL_COMPONENTS fortran)
if(SWIG_FOUND)
  message("SWIG found: ${SWIG_EXECUTABLE}")
  if(NOT SWIG_fortran_FOUND)
    message(WARNING "SWIG Fortran bindings cannot be generated")
  endif()
endif()
```

FindTCL

TK_INTERNAL_PATH was removed.

This module finds if Tcl is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

TCL_FOUND	= Tcl was found
TK_FOUND	= Tk was found
TCLTK_FOUND	= Tcl and Tk were found
TCL_LIBRARY	= path to Tcl library (tcl tcl80)
TCL_INCLUDE_PATH	= path to where tcl.h can be found
TCL_TCLSH	= path to tclsh binary (tcl tcl80)
TK_LIBRARY	= path to Tk library (tk tk80 etc)
TK_INCLUDE_PATH	= path to where tk.h can be found
TK_WISH	= full path to the wish executable

In an effort to remove some clutter and clear up some issues for people who are not necessarily Tcl/Tk gurus/developers, some variables were moved or removed. Changes compared to CMake 2.4 are:

```
=> they were only useful for people writing Tcl/Tk extensions.
=> these libs are not packaged by default with Tcl/Tk distributions.
    Even when Tcl/Tk is built from source, several flavors of debug libs
    are created and there is no real reason to pick a single one
```

specifically (say, amongst tcl84g, tcl84gs, or tcl84sgx).
 Let's leave that choice to the user by allowing him to assign
 TCL_LIBRARY to any Tcl library, debug or not.

=> this ended up being only a Win32 variable, and there is a lot of
 confusion regarding the location of this file in an installed Tcl/Tk
 tree anyway (see 8.5 for example). If you need the internal path at
 this point it is safer you ask directly where the **source** tree is
 and dig from there.

FindTclsh

Find tclsh

This module finds if TCL is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

TCLSH_FOUND = TRUE if tclsh has been found
 TCL_TCLSH = the path to the tclsh executable

FindTclStub

TCL_STUB_LIBRARY_DEBUG and TK_STUB_LIBRARY_DEBUG were removed.

This module finds Tcl stub libraries. It first finds Tcl include files and libraries by calling FindTCL.cmake.
 How to Use the Tcl Stubs Library:

<http://tcl.activestate.com/doc/howto/stubs.html>

Using Stub Libraries:

<http://safari.oreilly.com/0130385603/ch48lev1sec3>

This code sets the following variables:

TCL_STUB_LIBRARY = path to Tcl stub library
 TK_STUB_LIBRARY = path to Tk stub library
 TTK_STUB_LIBRARY = path to ttk stub library

In an effort to remove some clutter and clear up some issues for people who are not necessarily Tcl/Tk gurus/developers, some variables were moved or removed. Changes compared to CMake 2.4 are:

=> these libs are not packaged by default with Tcl/Tk distributions.
 Even when Tcl/Tk is built from source, several flavors of debug libs
 are created and there is no real reason to pick a single one
 specifically (say, amongst tclstub84g, tclstub84gs, or tclstub84sgx).
 Let's leave that choice to the user by allowing him to assign
 TCL_STUB_LIBRARY to any Tcl library, debug or not.

FindThreads

This module determines the thread library of the system.

Imported Targets

Added in version 3.1.

This module defines the following *IMPORTED* target:

Threads::Threads

The thread library, if found.

Result Variables

The following variables are set:

Threads_FOUND

If a supported thread library was found.

CMAKE_THREAD_LIBS_INIT

The thread library to use. This may be empty if the thread functions are provided by the system libraries and no special flags are needed to use them.

CMAKE_USE_WIN32_THREADS_INIT

If the found thread library is the win32 one.

CMAKE_USE_PTHREADS_INIT

If the found thread library is pthread compatible.

CMAKE_HP_PTHREADS_INIT

If the found thread library is the HP thread library.

Variables Affecting Behavior**THREADS_PREFER_PTHREAD_FLAG**

Added in version 3.1.

If the use of the `-pthread` compiler and linker flag is preferred then the caller can set this variable to `TRUE`. The compiler flag can only be used with the imported target. Use of both the imported target as well as this switch is highly recommended for new code.

This variable has no effect if the system libraries provide the thread functions, i.e. when **CMAKE_THREAD_LIBS_INIT** will be empty.

FindTIFF

Find the TIFF library (**libtiff**, <https://libtiff.gitlab.io/libtiff/>).

Optional COMPONENTS

This module supports the optional component *CXX*, for use with the **COMPONENTS** argument of the *find_package()* command. This component has an associated imported target, as described below.

Imported Targets

Added in version 3.5.

This module defines the following *IMPORTED* targets:

TIFF::TIFF

The TIFF library, if found.

TIFF::CXX

Added in version 3.19.

The C++ wrapper `libtiffxx`, if requested by the *COMPONENTS CXX* option, if the compiler is not `MSVC` (which includes the C++ wrapper in `libtiff`), and if found.

Result variables

This module will set the following variables in your project:

TIFF_FOUND

true if the TIFF headers and libraries were found

TIFF_INCLUDE_DIR

the directory containing the TIFF headers

TIFF_INCLUDE_DIRS

the directory containing the TIFF headers

TIFF_LIBRARIES

TIFF libraries to be linked

Cache variables

The following cache variables may also be set:

TIFF_INCLUDE_DIR

the directory containing the TIFF headers

TIFF_LIBRARY_RELEASE

the path to the TIFF library for release configurations

TIFF_LIBRARY_DEBUG

the path to the TIFF library for debug configurations

TIFFXX_LIBRARY_RELEASE

the path to the TIFFXX library for release configurations

TIFFXX_LIBRARY_DEBUG

the path to the TIFFXX library for debug configurations

Added in version 3.4: Debug and Release variants are found separately.

FindVulkan

Added in version 3.7.

Find Vulkan, which is a low-overhead, cross-platform 3D graphics and computing API.

Optional COMPONENTS

Added in version 3.24.

This module respects several optional COMPONENTS. There are corresponding imported targets for each of these.

glslc The SPIR-V compiler.

glslangValidator

The **glslangValidator** tool.

glslang The SPIR-V generator library.

shaderc_combined

The static library for Vulkan shader compilation.

SPIRV-Tools

Tools to process SPIR-V modules.

MoltenVK

On macOS, an additional component **MoltenVK** is available.

dx Added in version 3.25.

The DirectX Shader Compiler.

The **glslc** and **glslangValidator** components are provided even if not explicitly requested (for backward

compatibility).

Imported Targets

This module defines *IMPORTED* targets if Vulkan has been found:

Vulkan::Vulkan

The main Vulkan library.

Vulkan::glslc

Added in version 3.19.

The GLSLC SPIR-V compiler, if it has been found.

Vulkan::Headers

Added in version 3.21.

Provides just Vulkan headers include paths, if found. No library is included in this target. This can be useful for applications that load Vulkan library dynamically.

Vulkan::glslangValidator

Added in version 3.21.

The glslangValidator tool, if found. It is used to compile GLSL and HLSL shaders into SPIR-V.

Vulkan::glslang

Added in version 3.24.

Defined if SDK has the Khronos-reference front-end shader parser and SPIR-V generator library (glslang).

Vulkan::shaderc_combined

Added in version 3.24.

Defined if SDK has the Google static library for Vulkan shader compilation (shaderc_combined).

Vulkan::SPIRV-Tools

Added in version 3.24.

Defined if SDK has the Khronos library to process SPIR-V modules (SPIRV-Tools).

Vulkan::MoltenVK

Added in version 3.24.

Defined if SDK has the Khronos library which implement a subset of Vulkan API over Apple Metal graphics framework. (MoltenVK).

Vulkan::volk

Added in version 3.25.

Defined if SDK has the Vulkan meta-loader (volk).

Vulkan::dxc_lib

Added in version 3.25.

Defined if SDK has the DirectX shader compiler library.

Vulkan::dxc_exe

Added in version 3.25.

Defined if SDK has the DirectX shader compiler CLI tool.

Result Variables

This module defines the following variables:

Vulkan_FOUND

set to true if Vulkan was found

Vulkan_INCLUDE_DIRS

include directories for Vulkan

Vulkan_LIBRARIES

link against this library to use Vulkan

Vulkan_VERSION

Added in version 3.23.

value from `vulkan/vulkan_core.h`

Vulkan_glslc_FOUND

Added in version 3.24.

True, if the SDK has the glslc executable.

Vulkan_glslangValidator_FOUND

Added in version 3.24.

True, if the SDK has the glslangValidator executable.

Vulkan_glslang_FOUND

Added in version 3.24.

True, if the SDK has the glslang library.

Vulkan_shaderc_combined_FOUND

Added in version 3.24.

True, if the SDK has the shaderc_combined library.

Vulkan_SPIRV-Tools_FOUND

Added in version 3.24.

True, if the SDK has the SPIRV-Tools library.

Vulkan_MoltenVK_FOUND

Added in version 3.24.

True, if the SDK has the MoltenVK library.

Vulkan_volk_FOUND

Added in version 3.25.

True, if the SDK has the volk library.

Vulkan_dxc_lib_FOUND

Added in version 3.25.

True, if the SDK has the DirectX shader compiler library.

Vulkan_dxc_exe_FOUND

Added in version 3.25.

True, if the SDK has the DirectX shader compiler CLI tool.

The module will also defines these cache variables:

Vulkan_INCLUDE_DIR

the Vulkan include directory

Vulkan_LIBRARY

the path to the Vulkan library

Vulkan_GLSLC_EXECUTABLE

the path to the GLSL SPIR-V compiler

Vulkan_GLSLANG_VALIDATOR_EXECUTABLE

the path to the glslangValidator tool

Vulkan_glslang_LIBRARY

Added in version 3.24.

Path to the glslang library.

Vulkan_shaderc_combined_LIBRARY

Added in version 3.24.

Path to the shaderc_combined library.

Vulkan_SPIRV-Tools_LIBRARY

Added in version 3.24.

Path to the SPIRV-Tools library.

Vulkan_MoltenVK_LIBRARY

Added in version 3.24.

Path to the MoltenVK library.

Vulkan_volk_LIBRARY

Added in version 3.25.

Path to the volk library.

Vulkan_dxc_LIBRARY

Added in version 3.25.

Path to the DirectX shader compiler library.

Vulkan_dxc_EXECUTABLE

Added in version 3.25.

Path to the DirectX shader compiler CLI tool.

Hints

Added in version 3.18.

The **VULKAN_SDK** environment variable optionally specifies the location of the Vulkan SDK root directory for the given architecture. It is typically set by sourcing the toplevel **setup-env.sh** script of the Vulkan SDK directory into the shell environment.

FindWget

Find wget

This module looks for wget. This module defines the following values:

WGET_EXECUTABLE: the full path to the wget tool.

WGET_FOUND: True if wget has been found.

FindWish

Find wish installation

This module finds if TCL is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

TK_WISH = the path to the wish executable

if UNIX is defined, then it will look for the cygwin version first

FindwxWidgets

Find a wxWidgets (a.k.a., wxWindows) installation.

This module finds if wxWidgets is installed and selects a default configuration to use. wxWidgets is a modular library. To specify the modules that you will use, you need to name them as components to the package:

```
find_package(wxWidgets COMPONENTS core base ... OPTIONAL_COMPONENTS net ...)
```

Added in version 3.4: Support for *find_package()* version argument; **webview** component.

Added in version 3.14: **OPTIONAL_COMPONENTS** support.

There are two search branches: a windows style and a unix style. For windows, the following variables are searched for and set to defaults in case of multiple choices. Change them if the defaults are not desired (i.e., these are the only variables you should change to select a configuration):

- `wxWidgets_ROOT_DIR` - Base wxWidgets directory
 (e.g., `C:/wxWidgets-3.2.0`).
- `wxWidgets_LIB_DIR` - Path to wxWidgets libraries
 (e.g., `C:/wxWidgets-3.2.0/lib/vc_x64_lib`).
- `wxWidgets_CONFIGURATION` - Configuration to use
 (e.g., `msw`, `mswd`, `mswu`, `mswunivud`, etc.)
- `wxWidgets_EXCLUDE_COMMON_LIBRARIES`
 - Set to `TRUE` to exclude linking of
 commonly required libs (e.g., `png` `tiff`
 `jpeg` `zlib` `regex` `expat` `scintilla` `lexilla`).

For unix style it uses the `wx-config` utility. You can select between debug/release, unicode/ansi, universal/non-universal, and static/shared in the QtDialog or `ccmake` interfaces by turning ON/OFF the following variables:

```
wxWidgets_USE_DEBUG
wxWidgets_USE_UNICODE
wxWidgets_USE_UNIVERSAL
wxWidgets_USE_STATIC
```

There is also a `wxWidgets_CONFIG_OPTIONS` variable for all other options that need to be passed to the `wx-config` utility. For example, to use the base toolkit found in the `/usr/local` path, set the variable (before calling the `FIND_PACKAGE` command) as such:

```
set(wxWidgets_CONFIG_OPTIONS --toolkit=base --prefix=/usr)
```

The following are set after the configuration is done for both windows and unix style:

- `wxWidgets_FOUND` - Set to `TRUE` if wxWidgets was found.
- `wxWidgets_INCLUDE_DIRS` - Include directories for WIN32
 i.e., where to find `"wx/wx.h"` and
 `"wx/setup.h"`; possibly empty for unices.
- `wxWidgets_LIBRARIES` - Path to the wxWidgets libraries.
- `wxWidgets_LIBRARY_DIRS` - compile time link dirs, useful for
 `rpath` on UNIX. Typically an empty string
 in WIN32 environment.
- `wxWidgets_DEFINITIONS` - Contains defines required to compile/link
 against WX, e.g. `WXUSINGDLL`
- `wxWidgets_DEFINITIONS_DEBUG` - Contains defines required to compile/link
 against WX debug builds, e.g. `__WXDEBUG__`
- `wxWidgets_CXX_FLAGS` - Include dirs and compiler flags for
 unices, empty on WIN32. Essentially
 `"`wx-config --cxxflags`"`.
- `wxWidgets_USE_FILE` - Convenience include file.

Added in version 3.11: The following environment variables can be used as hints: **WX_CONFIG**, **WXRC_CMD**.

Sample usage:

```
# Note that for MinGW users the order of libs is important!
find_package(wxWidgets COMPONENTS gl core base OPTIONAL_COMPONENTS net)
if(wxWidgets_FOUND)
    include(${wxWidgets_USE_FILE})
    # and for each of your dependent executable/library targets:
    target_link_libraries(<YourTarget> ${wxWidgets_LIBRARIES})
endif()
```

If wxWidgets is required (i.e., not an optional part):

```
find_package(wxWidgets REQUIRED gl core base OPTIONAL_COMPONENTS net)
include(${wxWidgets_USE_FILE})
# and for each of your dependent executable/library targets:
target_link_libraries(<YourTarget> ${wxWidgets_LIBRARIES})
```

Imported Targets

Added in version 3.27.

This module defines the following *IMPORTED* targets:

wxWidgets::wxWidgets

An interface library providing usage requirements for the found components.

FindX11

Find X11 installation

Try to find X11 on UNIX systems. The following values are defined

```
X11_FOUND           - True if X11 is available
X11_INCLUDE_DIR     - include directories to use X11
X11_LIBRARIES       - link against these to use X11
```

and also the following more fine grained variables and targets:

Added in version 3.14: Imported targets.

X11_ICE_INCLUDE_PATH,	X11_ICE_LIB,	X11_ICE_FOUND,
X11_SM_INCLUDE_PATH,	X11_SM_LIB,	X11_SM_FOUND,
X11_X11_INCLUDE_PATH,	X11_X11_LIB,	
X11_Xaccessrules_INCLUDE_PATH,		
X11_Xaccessstr_INCLUDE_PATH,		X11_Xaccess_FOUND
X11_Xau_INCLUDE_PATH,	X11_Xau_LIB,	X11_Xau_FOUND,
X11_xcb_INCLUDE_PATH,	X11_xcb_LIB,	X11_xcb_FOUND,
X11_X11_xcb_INCLUDE_PATH,	X11_X11_xcb_LIB,	X11_X11_xcb_FOUND,
X11_xcb_composite_INCLUDE_PATH,	X11_xcb_composite_LIB,	X11_xcb_composite_FOUND,
X11_xcb_cursor_INCLUDE_PATH,	X11_xcb_cursor_LIB,	X11_xcb_cursor_FOUND,
X11_xcb_damage_INCLUDE_PATH,	X11_xcb_damage_LIB,	X11_xcb_damage_FOUND,
X11_xcb_dpms_INCLUDE_PATH,	X11_xcb_dpms_LIB,	X11_xcb_dpms_FOUND,
X11_xcb_dri2_INCLUDE_PATH,	X11_xcb_dri2_LIB,	X11_xcb_dri2_FOUND,
X11_xcb_dri3_INCLUDE_PATH,	X11_xcb_dri3_LIB,	X11_xcb_dri3_FOUND,
X11_xcb_errors_INCLUDE_PATH,	X11_xcb_errors_LIB,	X11_xcb_errors_FOUND,
X11_xcb_ewmh_INCLUDE_PATH,	X11_xcb_ewmh_LIB,	X11_xcb_ewmh_FOUND,
X11_xcb_glx_INCLUDE_PATH,	X11_xcb_glx_LIB,	X11_xcb_glx_FOUND,
X11_xcb_icccm_INCLUDE_PATH,	X11_xcb_icccm_LIB,	X11_xcb_icccm_FOUND,

X11_xcb_image_INCLUDE_PATH,	X11_xcb_image_LIB,	X11_xcb_image_FOUND,
X11_xcb_keysyms_INCLUDE_PATH,	X11_xcb_keysyms_LIB,	X11_xcb_keysyms_FOUND,
X11_xcb_present_INCLUDE_PATH,	X11_xcb_present_LIB,	X11_xcb_present_FOUND,
X11_xcb_randr_INCLUDE_PATH,	X11_xcb_randr_LIB,	X11_xcb_randr_FOUND,
X11_xcb_record_INCLUDE_PATH,	X11_xcb_record_LIB,	X11_xcb_record_FOUND,
X11_xcb_render_INCLUDE_PATH,	X11_xcb_render_LIB,	X11_xcb_render_FOUND,
X11_xcb_render_util_INCLUDE_PATH,	X11_xcb_render_util_LIB,	X11_xcb_render_util_FOUND,
X11_xcb_res_INCLUDE_PATH,	X11_xcb_res_LIB,	X11_xcb_res_FOUND,
X11_xcb_screensaver_INCLUDE_PATH,	X11_xcb_screensaver_LIB,	X11_xcb_screensaver_FOUND,
X11_xcb_shape_INCLUDE_PATH,	X11_xcb_shape_LIB,	X11_xcb_shape_FOUND,
X11_xcb_shm_INCLUDE_PATH,	X11_xcb_shm_LIB,	X11_xcb_shm_FOUND,
X11_xcb_sync_INCLUDE_PATH,	X11_xcb_sync_LIB,	X11_xcb_sync_FOUND,
X11_xcb_util_INCLUDE_PATH,	X11_xcb_util_LIB,	X11_xcb_util_FOUND,
X11_xcb_xf86dri_INCLUDE_PATH,	X11_xcb_xf86dri_LIB,	X11_xcb_xf86dri_FOUND,
X11_xcb_xfixes_INCLUDE_PATH,	X11_xcb_xfixes_LIB,	X11_xcb_xfixes_FOUND,
X11_xcb_xinerama_INCLUDE_PATH,	X11_xcb_xinerama_LIB,	X11_xcb_xinerama_FOUND,
X11_xcb_xinput_INCLUDE_PATH,	X11_xcb_xinput_LIB,	X11_xcb_xinput_FOUND,
X11_xcb_xkb_INCLUDE_PATH,	X11_xcb_xkb_LIB,	X11_xcb_xkb_FOUND,
X11_xcb_xrm_INCLUDE_PATH,	X11_xcb_xrm_LIB,	X11_xcb_xrm_FOUND,
X11_xcb_xtest_INCLUDE_PATH,	X11_xcb_xtest_LIB,	X11_xcb_xtest_FOUND,
X11_xcb_xvmc_INCLUDE_PATH,	X11_xcb_xvmc_LIB,	X11_xcb_xvmc_FOUND,
X11_xcb_xv_INCLUDE_PATH,	X11_xcb_xv_LIB,	X11_xcb_xv_FOUND,
X11_Xcomposite_INCLUDE_PATH,	X11_Xcomposite_LIB,	X11_Xcomposite_FOUND,
X11_Xcursor_INCLUDE_PATH,	X11_Xcursor_LIB,	X11_Xcursor_FOUND,
X11_Xdamage_INCLUDE_PATH,	X11_Xdamage_LIB,	X11_Xdamage_FOUND,
X11_Xdmcp_INCLUDE_PATH,	X11_Xdmcp_LIB,	X11_Xdmcp_FOUND,
X11_Xext_INCLUDE_PATH,	X11_Xext_LIB,	X11_Xext_FOUND,
X11_Xxf86misc_INCLUDE_PATH,	X11_Xxf86misc_LIB,	X11_Xxf86misc_FOUND,
X11_Xxf86vm_INCLUDE_PATH,	X11_Xxf86vm_LIB,	X11_Xxf86vm_FOUND,
X11_Xfixes_INCLUDE_PATH,	X11_Xfixes_LIB,	X11_Xfixes_FOUND,
X11_Xft_INCLUDE_PATH,	X11_Xft_LIB,	X11_Xft_FOUND,
X11_Xi_INCLUDE_PATH,	X11_Xi_LIB,	X11_Xi_FOUND,
X11_Xinerama_INCLUDE_PATH,	X11_Xinerama_LIB,	X11_Xinerama_FOUND,
X11_Xkb_INCLUDE_PATH,		X11_Xkb_FOUND,
X11_Xkblib_INCLUDE_PATH,		X11_Xkbcommon_FOUND,
X11_xkbcommon_INCLUDE_PATH,	X11_xkbcommon_LIB,	X11_xkbcommon_X11_FOUND,
X11_xkbcommon_X11_INCLUDE_PATH,	X11_xkbcommon_X11_LIB,	X11_xkbcommon_X11_FOUND,
X11_xkbfile_INCLUDE_PATH,	X11_xkbfile_LIB,	X11_xkbfile_FOUND,
X11_Xmu_INCLUDE_PATH,	X11_Xmu_LIB,	X11_Xmu_FOUND,
X11_Xpm_INCLUDE_PATH,	X11_Xpm_LIB,	X11_Xpm_FOUND,
X11_Xpresent_INCLUDE_PATH,	X11_Xpresent_LIB,	X11_Xpresent_FOUND,
X11_Xtst_INCLUDE_PATH,	X11_Xtst_LIB,	X11_Xtst_FOUND,
X11_Xrandr_INCLUDE_PATH,	X11_Xrandr_LIB,	X11_Xrandr_FOUND,
X11_Xrender_INCLUDE_PATH,	X11_Xrender_LIB,	X11_Xrender_FOUND,
X11_XRes_INCLUDE_PATH,	X11_XRes_LIB,	X11_XRes_FOUND,
X11_Xss_INCLUDE_PATH,	X11_Xss_LIB,	X11_Xss_FOUND,
X11_Xt_INCLUDE_PATH,	X11_Xt_LIB,	X11_Xt_FOUND,
X11_Xutil_INCLUDE_PATH,		X11_Xutil_FOUND,
X11_Xv_INCLUDE_PATH,	X11_Xv_LIB,	X11_Xv_FOUND,
X11_dpms_INCLUDE_PATH,	(in X11_Xext_LIB),	X11_dpms_FOUND
X11_Xdbe_INCLUDE_PATH,	(in X11_Xext_LIB),	X11_Xdbe_FOUND
X11_XShm_INCLUDE_PATH,	(in X11_Xext_LIB),	X11_XShm_FOUND
X11_Xshape_INCLUDE_PATH,	(in X11_Xext_LIB),	X11_Xshape_FOUND
X11_XSync_INCLUDE_PATH,	(in X11_Xext_LIB),	X11_XSync_FOUND

X11_Xaw_INCLUDE_PATH,

X11_Xaw_LIB

X11_Xaw_FOUND

Added in version 3.14: Renamed **Xxf86misc**, **X11_Xxf86misc**, **X11_Xxf86vm**, **X11_xkbfile**, **X11_Xtst**, and **X11_Xss** libraries to match their file names. Deprecated the **X11_Xinput** library. Old names are still available for compatibility.

Added in version 3.14: Added the **X11_Xext_INCLUDE_PATH** variable.

Added in version 3.18: Added the **xcb**, **X11-xcb**, **xcb-icccm**, **xcb-xkb**, **xkbcommon**, and **xkbcommon-X11** libraries.

Added in version 3.19: Added the **Xaw**, **xcb_util**, and **xcb_xfixes** libraries.

Added in version 3.24: Added the **xcb_randr**, **xcb_xtext**, and **xcb_keysyms** libraries.

Added in version 3.27: Added the **xcb_composite**, **xcb_cursor**, **xcb_damage**, **xcb_dpms**, **xcb_dri2**, **xcb_dri3**, **xcb_errors**, **xcb_ewmh**, **xcb_glx**, **xcb_image**, **xcb_present**, **xcb_record**, **xcb_render**, **xcb_render_util**, **xcb_res**, **xcb_screensaver**, **xcb_shape**, **xcb_shm**, **xcb_sync**, **xcb_xf86dri**, **xcb_xinerama**, **xcb_xinput**, **xcb_xrm**, **xcb_xvnc**, and **xcb_xv** libraries.

Added in version 3.29: Added coverage of double buffer extension (variables **X11_Xdbe_INCLUDE_PATH** and **X11_Xdbe_FOUND**).

FindXalanC

Added in version 3.5.

Find the Apache Xalan-C++ XSL transform processor headers and libraries.

Imported Targets

This module defines the following *IMPORTED* targets:

XalanC::XalanC

The Xalan-C++ **xalan-c** library, if found.

Result variables

This module will set the following variables in your project:

XalanC_FOUND

true if the Xalan headers and libraries were found

XalanC_VERSION

Xalan release version

XalanC_INCLUDE_DIRS

the directory containing the Xalan headers; note **XercesC_INCLUDE_DIRS** is also required

XalanC_LIBRARIES

Xalan libraries to be linked; note **XercesC_LIBRARIES** is also required

Cache variables

The following cache variables may also be set:

XalanC_INCLUDE_DIR

the directory containing the Xalan headers

XalanC_LIBRARY

the Xalan library

FindXCTest

Added in version 3.3.

Functions to help creating and executing XCTest bundles.

An XCTest bundle is a CFBundle with a special product-type and bundle extension. The Mac Developer Library provides more information in the *Testing with Xcode* document.

Module Functions**xctest_add_bundle**

The **xctest_add_bundle** function creates a XCTest bundle named <target> which will test the target <testee>. Supported target types for testee are Frameworks and App Bundles:

```
xctest_add_bundle(
    <target> # Name of the XCTest bundle
    <testee> # Target name of the testee
)
```

xctest_add_test

The **xctest_add_test** function adds an XCTest bundle to the project to be run by *ctest(1)*. The test will be named <name> and tests <bundle>:

```
xctest_add_test(
    <name> # Test name
    <bundle> # Target name of XCTest bundle
)
```

Module Variables

The following variables are set by including this module:

XCTest_FOUND

True if the XCTest Framework and executable were found.

XCTest_EXECUTABLE

The path to the xctest command line tool used to execute XCTest bundles.

XCTest_INCLUDE_DIRS

The directory containing the XCTest Framework headers.

XCTest_LIBRARIES

The location of the XCTest Framework.

FindXercesC

Added in version 3.1.

Find the Apache Xerces-C++ validating XML parser headers and libraries.

Imported Targets

Added in version 3.5.

This module defines the following *IMPORTED* targets:

XercesC::XercesC

The Xerces-C++ **xerces-c** library, if found.

Result variables

This module will set the following variables in your project:

XercesC_FOUND

true if the Xerces headers and libraries were found

XercesC_VERSION

Xerces release version

XercesC_INCLUDE_DIRS

the directory containing the Xerces headers

XercesC_LIBRARIES

Xerces libraries to be linked

Cache variables

The following cache variables may also be set:

XercesC_INCLUDE_DIR

the directory containing the Xerces headers

XercesC_LIBRARY

the Xerces library

Added in version 3.4: Debug and Release variants are found separately.

FindXMLRPC

Find xmlrpc

Find the native XMLRPC headers and libraries.

XMLRPC_INCLUDE_DIRS	- where to find xmlrpc.h, etc.
XMLRPC_LIBRARIES	- List of libraries when using xmlrpc.
XMLRPC_FOUND	- True if xmlrpc found.

XMLRPC modules may be specified as components for this find module. Modules may be listed by running "xmlrpc-c-config". Modules include:

c++	C++ wrapper code
libwww-client	libwww-based client
cgi-server	CGI-based server
abyss-server	ABYSS-based server

Typical usage:

```
find_package(XMLRPC REQUIRED libwww-client)
```

FindZLIB

Find the native ZLIB includes and library.

Imported Targets

Added in version 3.1.

This module defines *IMPORTED* target **ZLIB::ZLIB**, if ZLIB has been found.

Result Variables

This module defines the following variables:

ZLIB_INCLUDE_DIRS

where to find zlib.h, etc.

ZLIB_LIBRARIES

List of libraries when using zlib.

ZLIB_FOUND

True if zlib found.

ZLIB_VERSION

Added in version 3.26: the version of Zlib found.

See also legacy variable **ZLIB_VERSION_STRING**.

Added in version 3.4: Debug and Release variants are found separately.

Legacy Variables

The following variables are provided for backward compatibility:

ZLIB_VERSION_MAJOR

The major version of zlib.

Changed in version 3.26: Superseded by **ZLIB_VERSION**.

ZLIB_VERSION_MINOR

The minor version of zlib.

Changed in version 3.26: Superseded by **ZLIB_VERSION**.

ZLIB_VERSION_PATCH

The patch version of zlib.

Changed in version 3.26: Superseded by **ZLIB_VERSION**.

ZLIB_VERSION_TWEAK

The tweak version of zlib.

Changed in version 3.26: Superseded by **ZLIB_VERSION**.

ZLIB_VERSION_STRING

The version of zlib found (x.y.z)

Changed in version 3.26: Superseded by **ZLIB_VERSION**.

ZLIB_MAJOR_VERSION

The major version of zlib. Superseded by **ZLIB_VERSION_MAJOR**.

ZLIB_MINOR_VERSION

The minor version of zlib. Superseded by **ZLIB_VERSION_MINOR**.

ZLIB_PATCH_VERSION

The patch version of zlib. Superseded by **ZLIB_VERSION_PATCH**.

Hints

A user may set **ZLIB_ROOT** to a zlib installation root to tell this module where to look.

Added in version 3.24: Set **ZLIB_USE_STATIC_LIBS** to **ON** to look for static libraries. Default is **OFF**.

DEPRECATED MODULES**Deprecated Utility Modules****AddFileDependencies**

Deprecated since version 3.20.

Add dependencies to a source file.

```
add_file_dependencies(<source> <files>...)
```

Adds the given **<files>** to the dependencies of file **<source>**.

Do not use this command in new code. It is just a wrapper around:

```
set_property(SOURCE <source> APPEND PROPERTY OBJECT_DEPENDS <files>...)
```

Instead use the *set_property()* command to append to the *OBJECT_DEPENDS* source file property directly.

CMakeDetermineVSServicePack

Deprecated since version 3.0: Do not use.

The functionality of this module has been superseded by the *CMAKE_<LANG>_COMPILER_VERSION* variable that contains the compiler version number.

Determine the Visual Studio service pack of the 'cl' in use.

Usage:

```
if(MSVC)
  include(CMakeDetermineVSServicePack)
  DetermineVSServicePack(my_service_pack)
  if(my_service_pack)
    message(STATUS "Detected: ${my_service_pack}")
  endif()
endif()
```

Function DetermineVSServicePack sets the given variable to one of the following values or an empty string if unknown:

```
vc80, vc80sp1
vc90, vc90sp1
vc100, vc100sp1
vc110, vc110sp1, vc110sp2, vc110sp3, vc110sp4
```

CMakeExpandImportedTargets

Deprecated since version 3.4: Do not use.

This module was once needed to expand imported targets to the underlying libraries they reference on disk for use with the *try_compile()* and *try_run()* commands. These commands now support imported libraries in their **LINK_LIBRARIES** options (since CMake 2.8.11 for *try_compile()* and since CMake 3.2 for *try_run()*).

This module does not support the policy *CMP0022* **NEW** behavior or use of the *INTERFACE_LINK_LIBRARIES* property because *generator expressions* cannot be evaluated during configuration.

```
CMAKE_EXPAND_IMPORTED_TARGETS(<var> LIBRARIES lib1 lib2...libN
                                [CONFIGURATION <config>])
```

CMAKE_EXPAND_IMPORTED_TARGETS() takes a list of libraries and replaces all imported targets contained in this list with their actual file paths of the referenced libraries on disk, including the libraries from their link interfaces. If a *CONFIGURATION* is given, it uses the respective configuration of the imported targets if it exists. If no *CONFIGURATION* is given, it uses the first configuration from *_\${CMAKE_CONFIGURATION_TYPES}* if set, otherwise *_\${CMAKE_BUILD_TYPE}*.

```
cmake_expand_imported_targets(expandedLibs
    LIBRARIES ${CMAKE_REQUIRED_LIBRARIES}
    CONFIGURATION "${CMAKE_TRY_COMPILE_CONFIGURATION}" )
```

CMakeFindFrameworks

Deprecated since version 3.31: This module does nothing, unless policy *CMP0173* is set to **OLD**.

helper module to find OSX frameworks

This module reads hints about search locations from variables:

```
CMAKE_FIND_FRAMEWORK_EXTRA_LOCATIONS - Extra directories
```

CMakeForceCompiler

Deprecated since version 3.6: Do not use.

The macros provided by this module were once intended for use by cross-compiling toolchain files when CMake was not able to automatically detect the compiler identification. Since the introduction of this module, CMake's compiler identification capabilities have improved and can now be taught to recognize any compiler. Furthermore, the suite of information CMake detects from a compiler is now too extensive to be provided by toolchain files using these macros.

One common use case for this module was to skip CMake's checks for a working compiler when using a cross-compiler that cannot link binaries without special flags or custom linker scripts. This case is now supported by setting the *CMAKE_TRY_COMPILE_TARGET_TYPE* variable in the toolchain file instead.

Macro **CMAKE_FORCE_C_COMPILER** has the following signature:

```
CMAKE_FORCE_C_COMPILER(<compiler> <compiler-id>)
```

It sets *CMAKE_C_COMPILER* to the given compiler and the *cmake* internal variable

CMAKE_C_COMPILER_ID to the given compiler-id. It also bypasses the check for working compiler and basic compiler information tests.

Macro **CMAKE_FORCE_CXX_COMPILER** has the following signature:

```
CMAKE_FORCE_CXX_COMPILER(<compiler> <compiler-id>)
```

It sets *CMAKE_CXX_COMPILER* to the given compiler and the cmake internal variable *CMAKE_CXX_COMPILER_ID* to the given compiler-id. It also bypasses the check for working compiler and basic compiler information tests.

Macro **CMAKE_FORCE_Fortran_COMPILER** has the following signature:

```
CMAKE_FORCE_Fortran_COMPILER(<compiler> <compiler-id>)
```

It sets *CMAKE_Fortran_COMPILER* to the given compiler and the cmake internal variable *CMAKE_Fortran_COMPILER_ID* to the given compiler-id. It also bypasses the check for working compiler and basic compiler information tests.

So a simple toolchain file could look like this:

```
include (CMakeForceCompiler)
set(CMAKE_SYSTEM_NAME Generic)
CMAKE_FORCE_C_COMPILER (chc12 MetrowerksHicross)
CMAKE_FORCE_CXX_COMPILER (chc12 MetrowerksHicross)
```

CMakeParseArguments

Deprecated since version 3.5.

This module once implemented the *cmake_parse_arguments()* command that is now implemented natively by CMake. It is now an empty placeholder for compatibility with projects that include it to get the command from CMake 3.4 and lower.

Dart

Deprecated since version 3.27: This module is available only if policy *CMP0145* is not set to **NEW**. Do not use it in new code. Use the *CTest* module instead.

Configure a project for testing with CTest or old Dart Tcl Client

This file is the backwards-compatibility version of the CTest module. It supports using the old Dart 1 Tcl client for driving dashboard submissions as well as testing with CTest. This module should be included in the CMakeLists.txt file at the top of a project. Typical usage:

```
include(Dart)
if(BUILD_TESTING)
  # ... testing related CMake code ...
endif()
```

The BUILD_TESTING option is created by the Dart module to determine whether testing support should be enabled. The default is ON.

Documentation

Deprecated since version 3.18: This module does nothing, unless policy *CMP0106* is set to **OLD**.

This module provides support for the VTK documentation framework. It relies on several tools (Doxygen, Perl, etc).

GetPrerequisites

Deprecated since version 3.16: Use `file(GET_RUNTIME_DEPENDENCIES)` instead.

Functions to analyze and list executable file prerequisites.

This module provides functions to list the .dll, .dylib or .so files that an executable or shared library file depends on. (Its prerequisites.)

It uses various tools to obtain the list of required shared library files:

```
dumpbin (Windows)
objdump (MinGW on Windows)
ldd (Linux/Unix)
otool (Mac OSX)
```

Changed in version 3.16: The tool specified by the `CMAKE_OBJDUMP` variable will be used, if set.

The following functions are provided by this module:

```
get_prerequisites
list_prerequisites
list_prerequisites_by_glob
gp_append_unique
is_file_executable
gp_item_default_embedded_path
    (projects can override with gp_item_default_embedded_path_override)
gp_resolve_item
    (projects can override with gp_resolve_item_override)
gp_resolved_file_type
    (projects can override with gp_resolved_file_type_override)
gp_file_type
```

```
GET_PREREQUISITES(<target> <prerequisites_var> <exclude_system> <recurse>
                  <exepath> <dirs> [<rpaths>])
```

Get the list of shared library files required by <target>. The list in the variable named <prerequisites_var> should be empty on first entry to this function. On exit, <prerequisites_var> will contain the list of required shared library files.

<target> is the full path to an executable file. <prerequisites_var> is the name of a CMake variable to contain the results. <exclude_system> must be 0 or 1 indicating whether to include or exclude "system" prerequisites. If <recurse> is set to 1 all prerequisites will be found recursively, if set to 0 only direct prerequisites are listed. <exepath> is the path to the top level executable used for @executable_path replacement on the Mac. <dirs> is a list of paths where libraries might be found: these paths are searched first when a target without any path info is given. Then standard system locations are also searched: PATH, Framework locations, /usr/lib...

Added in version 3.14: The variable `GET_PREREQUISITES_VERBOSE` can be set to true to enable verbose output.

```
LIST_PREREQUISITES(<target> [<recurse> [<exclude_system> [<verbose>]]])
```

Print a message listing the prerequisites of <target>.

<target> is the name of a shared library or executable target or the full path to a shared library or executable file. If <recurse> is set to 1 all prerequisites will be found recursively, if set to 0 only direct prerequisites are listed. <exclude_system> must be 0 or 1 indicating whether to include or exclude "system" prerequisites. With <verbose> set to 0 only the full path names of the prerequisites are printed, set to 1 extra information will be displayed.

```
LIST_PREREQUISITES_BY_GLOB(<glob_arg> <glob_exp>)
```

Print the prerequisites of shared library and executable files matching a globbing pattern. <glob_arg> is GLOB or GLOB_RECURSE and <glob_exp> is a globbing expression used with "file(GLOB" or "file(GLOB_RECURSE" to retrieve a list of matching files. If a matching file is executable, its prerequisites are listed.

Any additional (optional) arguments provided are passed along as the optional arguments to the list_prerequisites calls.

```
GP_APPEND_UNIQUE(<list_var> <value>)
```

Append <value> to the list variable <list_var> only if the value is not already in the list.

```
IS_FILE_EXECUTABLE(<file> <result_var>)
```

Return 1 in <result_var> if <file> is a binary executable, 0 otherwise.

```
GP_ITEM_DEFAULT_EMBEDDED_PATH(<item> <default_embedded_path_var>)
```

Return the path that others should refer to the item by when the item is embedded inside a bundle.

Override on a per-project basis by providing a project-specific gp_item_default_embedded_path_override function.

```
GP_RESOLVE_ITEM(<context> <item> <exepath> <dirs> <resolved_item_var>
                [<rpaths>])
```

Resolve an item into an existing full path file.

Override on a per-project basis by providing a project-specific gp_resolve_item_override function.

```
GP_RESOLVED_FILE_TYPE(<original_file> <file> <exepath> <dirs> <type_var>
                    [<rpaths>])
```

Return the type of <file> with respect to <original_file>. String describing type of prerequisite is returned in variable named <type_var>.

Use <exepath> and <dirs> if necessary to resolve non-absolute <file> values -- but only for non-embedded items.

Possible types are:

```
system
```

local
embedded
other

Override on a per-project basis by providing a project-specific `gp_resolved_file_type_override` function.

`GP_FILE_TYPE(<original_file> <file> <type_var>)`

Return the type of `<file>` with respect to `<original_file>`. String describing type of prerequisite is returned in variable named `<type_var>`.

Possible types are:

system
local
embedded
other

MacroAddFileDependencies

Deprecated since version 3.14.

`MACRO_ADD_FILE_DEPENDENCIES(<source> <files>...)`

Do not use this command in new code. It is just a wrapper around:

`set_property(SOURCE <source> APPEND PROPERTY OBJECT_DEPENDS <files>...)`

Instead use the `set_property()` command to append to the `OBJECT_DEPENDS` source file property directly.

SquishTestScript

Deprecated since version 3.0.

This script launches a GUI test using Squish. You should not call the script directly; instead, you should access it via the `SQUISH_ADD_TEST` macro that is defined in `FindSquish.cmake`.

This script starts the Squish server, launches the test on the client, and finally stops the squish server. If any of these steps fail (including if the tests do not pass) then a fatal error is raised.

TestBigEndian

Deprecated since version 3.20: Superseded by the `CMAKE_<LANG>_BYTE_ORDER` variable.

Check if the target architecture is big endian or little endian.

test_big_endian

`test_big_endian(<var>)`

Stores in variable `<var>` either 1 or 0 indicating whether the target architecture is big or little endian.

TestCXXAcceptsFlag

Deprecated since version 3.0: See `CheckCXXCompilerFlag`.

Check if the CXX compiler accepts a flag.

```
CHECK_CXX_ACCEPTS_FLAG(<flags> <variable>)
<flags> the flags to try
<variable>
    variable to store the result
```

Use_wxWindows

Deprecated since version 2.8.10: Use **find_package(wxWidgets)** and **include(\${wxWidgets_USE_FILE})** instead.

This convenience include finds if wxWindows is installed and set the appropriate libs, incdirs, flags etc. author Jan Woetzel <jw-at-mip.informatik.uni-kiel.de> (07/2003)

USAGE:

```
just include Use_wxWindows.cmake
in your projects CMakeLists.txt

include( ${CMAKE_MODULE_PATH}/Use_wxWindows.cmake)

if you are sure you need GL then

set(WXWINDOWS_USE_GL 1)

*before* you include this file.
```

UseJavaClassFilelist

Changed in version 3.20: This module was previously documented by mistake and was never meant for direct inclusion by project code. See the *UseJava* module.

UseJavaSymlinks

Changed in version 3.20: This module was previously documented by mistake and was never meant for direct inclusion by project code. See the *UseJava* module.

UsePkgConfig

Deprecated since version 3.0: Use *FindPkgConfig* instead.

This module defines the following macro:

```
PKGCONFIG(package includedir libdir linkflags cflags)
```

Calling PKGCONFIG will fill the desired information into the 4 given arguments, e.g. PKGCONFIG(libart-2.0 LIBART_INCLUDE_DIR LIBART_LINK_DIR LIBART_LINK_FLAGS LIBART_CFLAGS) if pkg-config was NOT found or the specified software package doesn't exist, the variable will be empty when the function returns, otherwise they will contain the respective information

WriteBasicConfigVersionFile

Deprecated since version 3.0: Use the identical command *write_basic_package_version_file()* from module *CMakePackageConfigHelpers*.

```
WRITE_BASIC_CONFIG_VERSION_FILE(filename
```

```

[VERSION major.minor.patch]
COMPATIBILITY (AnyNewerVersion|SameMajorVersion|SameMinorVersion|ExactVersion)
[ARCH_INDEPENDENT]
)

```

WriteCompilerDetectionHeader

Deprecated since version 3.20: This module is available only if policy *CMP0120* is not set to **NEW**. Do not use it in new code.

Added in version 3.1.

This module provides the function **write_compiler_detection_header()**.

This function can be used to generate a file suitable for preprocessor inclusion which contains macros to be used in source code:

```

write_compiler_detection_header(
    FILE <file>
    PREFIX <prefix>
    [OUTPUT_FILES_VAR <output_files_var> OUTPUT_DIR <output_dir>]
    COMPILERS <compiler> [...]
    FEATURES <feature> [...]
    [BARE_FEATURES <feature> [...]]
    [VERSION <version>]
    [PROLOG <prolog>]
    [EPILOG <epilog>]
    [ALLOW_UNKNOWN_COMPILERS]
    [ALLOW_UNKNOWN_COMPILER_VERSIONS]
)

```

This generates the file <file> with macros which all have the prefix <prefix>.

By default, all content is written directly to the <file>. The **OUTPUT_FILES_VAR** may be specified to cause the compiler-specific content to be written to separate files. The separate files are then available in the <output_files_var> and may be consumed by the caller for installation for example. The **OUTPUT_DIR** specifies a relative path from the main <file> to the compiler-specific files. For example:

```

write_compiler_detection_header(
    FILE climbingstats_compiler_detection.h
    PREFIX ClimbingStats
    OUTPUT_FILES_VAR support_files
    OUTPUT_DIR compilers
    COMPILERS GNU Clang MSVC Intel
    FEATURES cxx_variadic_templates
)
install(FILES
    ${CMAKE_CURRENT_BINARY_DIR}/climbingstats_compiler_detection.h
    DESTINATION include
)
install(FILES
    ${support_files}
    DESTINATION include/compilers
)

```

VERSION may be used to specify the API version to be generated. Future versions of CMake may introduce alternative APIs. A given API is selected by any **<version>** value greater than or equal to the version of CMake that introduced the given API and less than the version of CMake that introduced its succeeding API. The value of the *CMAKE_MINIMUM_REQUIRED_VERSION* variable is used if no explicit version is specified. (As of CMake version 4.0.0-rc1 there is only one API version.)

PROLOG may be specified as text content to write at the start of the header. **EPILOG** may be specified as text content to write at the end of the header

At least one **<compiler>** and one **<feature>** must be listed. Compilers which are known to CMake, but not specified are detected and a preprocessor **#error** is generated for them. A preprocessor macro matching **<PREFIX>_COMPILER_IS_<compiler>** is generated for each compiler known to CMake to contain the value **0** or **1**.

Possible compiler identifiers are documented with the *CMAKE_<LANG>_COMPILER_ID* variable. Available features in this version of CMake are listed in the *CMAKE_C_KNOWN_FEATURES* and *CMAKE_CXX_KNOWN_FEATURES* global properties. See the *cmake-compile-features(7)* manual for information on compile features.

Added in version 3.2: Added **MSVC** and **AppleClang** compiler support.

Added in version 3.6: Added **Intel** compiler support.

Changed in version 3.8: The **{c,cxx}_std_*** meta-features are ignored if requested.

Added in version 3.8: **ALLOW_UNKNOWN_COMPILERS** and **ALLOW_UNKNOWN_COMPILER_VERSIONS** cause the module to generate conditions that treat unknown compilers as simply lacking all features. Without these options the default behavior is to generate a **#error** for unknown compilers and versions.

Added in version 3.12: **BARE_FEATURES** will define the compatibility macros with the name used in newer versions of the language standard, so the code can use the new feature name unconditionally.

Feature Test Macros

For each compiler, a preprocessor macro is generated matching **<PREFIX>_COMPILER_IS_<compiler>** which has the content either **0** or **1**, depending on the compiler in use. Preprocessor macros for compiler version components are generated matching **<PREFIX>_COMPILER_VERSION_MAJOR** **<PREFIX>_COMPILER_VERSION_MINOR** and **<PREFIX>_COMPILER_VERSION_PATCH** containing decimal values for the corresponding compiler version components, if defined.

A preprocessor test is generated based on the compiler version denoting whether each feature is enabled. A preprocessor macro matching **<PREFIX>_COMPILER_<FEATURE>**, where **<FEATURE>** is the upper-case **<feature>** name, is generated to contain the value **0** or **1** depending on whether the compiler in use supports the feature:

```
write_compiler_detection_header(
  FILE climbingstats_compiler_detection.h
  PREFIX ClimbingStats
  COMPILERS GNU Clang AppleClang MSVC Intel
  FEATURES cxx_variadic_templates
```

```

)

#if ClimbingStats_COMPILER_CXX_VARIADIC_TEMPLATES
template<typename... T>
void someInterface(T t...) { /* ... */ }
#else
// Compatibility versions
template<typename T1>
void someInterface(T1 t1) { /* ... */ }
template<typename T1, typename T2>
void someInterface(T1 t1, T2 t2) { /* ... */ }
template<typename T1, typename T2, typename T3>
void someInterface(T1 t1, T2 t2, T3 t3) { /* ... */ }
#endif

```

Symbol Macros

Some additional symbol-defines are created for particular features for use as symbols which may be conditionally defined empty:

```

class MyClass ClimbingStats_FINAL
{
    ClimbingStats_CONSTEXPR int someInterface() { return 42; }
};

```

The **ClimbingStats_FINAL** macro will expand to **final** if the compiler (and its flags) support the **cxx_final** feature, and the **ClimbingStats_CONSTEXPR** macro will expand to **constexpr** if **cxx_constexpr** is supported.

If **BARE_FEATURES cxx_final** was given as argument the **final** keyword will be defined for old compilers, too.

The following features generate corresponding symbol defines and if they are available as **BARE_FEATURES**:

Feature	Define	Symbol	bare
c_restrict	<PREFIX>_RE- STRICT	restrict	yes
cxx_constexpr	<PREFIX>_CONST- EXPR	constexpr	yes
cxx_deleted_func- tions	<PRE- FIX>_DELETED_FUNC- TION	= delete	
cxx_extern_templates	<PREFIX>_EX- TERN_TEMPLATE	extern	
cxx_final	<PREFIX>_FINAL	final	yes
cxx_noexcept	<PREFIX>_NOEXCEPT	noexcept	yes
cxx_noexcept	<PREFIX>_NOEX- CEPT_EXPR(X)	noexcept(X)	
cxx_override	<PREFIX>_OVERRIDE	override	yes

Compatibility Implementation Macros

Some features are suitable for wrapping in a macro with a backward compatibility implementation if the compiler does not support the feature.

When the **cxx_static_assert** feature is not provided by the compiler, a compatibility implementation is

available via the `<PREFIX>_STATIC_ASSERT(COND)` and `<PREFIX>_STATIC_ASSERT_MSG(COND, MSG)` function-like macros. The macros expand to `static_assert` where that compiler feature is available, and to a compatibility implementation otherwise. In the first form, the condition is stringified in the message field of `static_assert`. In the second form, the message `MSG` is passed to the message field of `static_assert`, or ignored if using the backward compatibility implementation.

The `cxx_attribute_deprecated` feature provides a macro definition `<PREFIX>_DEPRECATED`, which expands to either the standard `[[deprecated]]` attribute or a compiler-specific decorator such as `__attribute__((deprecated))` used by GNU compilers.

The `cxx_alignas` feature provides a macro definition `<PREFIX>_ALIGNAS` which expands to either the standard `alignas` decorator or a compiler-specific decorator such as `__attribute__((aligned))` used by GNU compilers.

The `cxx_alignof` feature provides a macro definition `<PREFIX>_ALIGNOF` which expands to either the standard `alignof` decorator or a compiler-specific decorator such as `__alignof__` used by GNU compilers.

Feature	Define	Symbol	bare
<code>cxx_alignas</code>	<code><PREFIX>_ALIGNAS</code>	<code>alignas</code>	
<code>cxx_alignof</code>	<code><PREFIX>_ALIGNOF</code>	<code>alignof</code>	
<code>cxx_nullptr</code>	<code><PREFIX>_NULLPTR</code>	<code>nullptr</code>	yes
<code>cxx_static_assert</code>	<code><PREFIX>_STATIC_ASSERT</code>	<code>static_assert</code>	
<code>cxx_static_assert</code>	<code><PREFIX>_STATIC_ASSERT_MSG</code>	<code>static_assert</code>	
<code>cxx_attribute_deprecated</code>	<code><PREFIX>_DEPRECATED</code>	<code>[[deprecated]]</code>	
<code>cxx_attribute_deprecated</code>	<code><PREFIX>_DEPRECATED_MSG</code>	<code>[[deprecated]]</code>	
<code>cxx_thread_local</code>	<code><PREFIX>_THREAD_LOCAL</code>	<code>thread_local</code>	

A use-case which arises with such deprecation macros is the deprecation of an entire library. In that case, all public API in the library may be decorated with the `<PREFIX>_DEPRECATED` macro. This results in very noisy build output when building the library itself, so the macro may be defined to empty in that case when building the deprecated library:

```
add_library(compat_support ${srcs})
target_compile_definitions(compat_support
    PRIVATE
        CompatSupport_DEPRECATED=
)
```

Example Usage

NOTE:

This section was migrated from the *cmake-compile-features(7)* manual since it relies on the **WriteCompilerDetectionHeader** module which is removed by policy *CMP0120*.

Compile features may be preferred if available, without creating a hard requirement. For example, a library may provide alternative implementations depending on whether the `cxx_variadic_templates` feature is available:

```

#if Foo_COMPILER_CXX_VARIADIC_TEMPLATES
template<int I, int... Is>
struct Interface;

template<int I>
struct Interface<I>
{
    static int accumulate()
    {
        return I;
    }
};

template<int I, int... Is>
struct Interface
{
    static int accumulate()
    {
        return I + Interface<Is...>::accumulate();
    }
};
#else
template<int I1, int I2 = 0, int I3 = 0, int I4 = 0>
struct Interface
{
    static int accumulate() { return I1 + I2 + I3 + I4; }
};
#endif

```

Such an interface depends on using the correct preprocessor defines for the compiler features. CMake can generate a header file containing such defines using the *WriteCompilerDetectionHeader* module. The module contains the **write_compiler_detection_header** function which accepts parameters to control the content of the generated header file:

```

write_compiler_detection_header(
    FILE "${CMAKE_CURRENT_BINARY_DIR}/foo_compiler_detection.h"
    PREFIX Foo
    COMPILERS GNU
    FEATURES
        cxx_variadic_templates
)

```

Such a header file may be used internally in the source code of a project, and it may be installed and used in the interface of library code.

For each feature listed in **FEATURES**, a preprocessor definition is created in the header file, and defined to either **1** or **0**.

Additionally, some features call for additional defines, such as the **cxx_final** and **cxx_override** features. Rather than being used in **#ifdef** code, the **final** keyword is abstracted by a symbol which is defined to either **final**, a compiler-specific equivalent, or to empty. That way, C++ code can be written to unconditionally use the symbol, and compiler support determines what it is expanded to:

```

struct Interface {

```

```

    virtual void Execute() = 0;
};

struct Concrete Foo_FINAL {
    void Execute() Foo_OVERRIDE;
};

```

In this case, **Foo_FINAL** will expand to **final** if the compiler supports the keyword, or to empty otherwise.

In this use-case, the project code may wish to enable a particular language standard if available from the compiler. The *CXX_STANDARD* target property may be set to the desired language standard for a particular target, and the *CMAKE_CXX_STANDARD* variable may be set to influence all following targets:

```

write_compiler_detection_header(
    FILE "${CMAKE_CURRENT_BINARY_DIR}/foo_compiler_detection.h"
    PREFIX Foo
    COMPILERS GNU
    FEATURES
        cxx_final cxx_override
)

# Includes foo_compiler_detection.h and uses the Foo_FINAL symbol
# which will expand to 'final' if the compiler supports the requested
# CXX_STANDARD.
add_library(foo foo.cpp)
set_property(TARGET foo PROPERTY CXX_STANDARD 11)

# Includes foo_compiler_detection.h and uses the Foo_FINAL symbol
# which will expand to 'final' if the compiler supports the feature,
# even though CXX_STANDARD is not set explicitly. The requirement of
# cxx_constexpr causes CMake to set CXX_STANDARD internally, which
# affects the compile flags.
add_library(foo_impl foo_impl.cpp)
target_compile_features(foo_impl PRIVATE cxx_constexpr)

```

The **write_compiler_detection_header** function also creates compatibility code for other features which have standard equivalents. For example, the **cxx_static_assert** feature is emulated with a template and abstracted via the **<PREFIX>_STATIC_ASSERT** and **<PREFIX>_STATIC_ASSERT_MSG** function-macros.

Deprecated Find Modules

FindBoost

Changed in version 3.30: This module is available only if policy *CMP0167* is not set to **NEW**. Port projects to upstream Boost's **BoostConfig.cmake** package configuration file, for which **find_package(Boost)** now searches.

Find Boost include dirs and libraries

Use this module by invoking *find_package()* with the form:

```

find_package(Boost
    [version] [EXACT]      # Minimum or EXACT version e.g. 1.67.0
    [REQUIRED]             # Fail with error if Boost is not found
    [COMPONENTS <libs>...] # Boost libraries by their canonical name

```

```

                                # e.g. "date_time" for "libboost_date_time"
[OPTIONAL_COMPONENTS <libs>...]
                                # Optional Boost libraries by their canonical name)
)                                # e.g. "date_time" for "libboost_date_time"

```

This module finds headers and requested component libraries OR a CMake package configuration file provided by a "Boost CMake" build. For the latter case skip to the *Boost CMake* section below.

Added in version 3.7: **bzip2** and **zlib** components (Windows only).

Added in version 3.11: The **OPTIONAL_COMPONENTS** option.

Added in version 3.13: **stacktrace_*** components.

Added in version 3.19: **bzip2** and **zlib** components on all platforms.

Result Variables

This module defines the following variables:

Boost_FOUND

True if headers and requested libraries were found.

Boost_INCLUDE_DIRS

Boost include directories.

Boost_LIBRARY_DIRS

Link directories for Boost libraries.

Boost_LIBRARIES

Boost component libraries to be linked.

Boost_<COMPONENT>_FOUND

True if component <COMPONENT> was found (<COMPONENT> name is upper-case).

Boost_<COMPONENT>_LIBRARY

Libraries to link for component <COMPONENT> (may include *target_link_libraries()* debug/optimized keywords).

Boost_VERSION_MACRO

BOOST_VERSION value from **boost/version.hpp**.

Boost_VERSION_STRING

Boost version number in **X.Y.Z** format.

Boost_VERSION

Boost version number in **X.Y.Z** format (same as **Boost_VERSION_STRING**).

Changed in version 3.15: In previous CMake versions, this variable used the raw version string from the Boost header (same as **Boost_VERSION_MACRO**). See policy *CMP0093*.

Boost_LIB_VERSION

Version string appended to library filenames.

Boost_VERSION_MAJOR, Boost_MAJOR_VERSION

Boost major version number (**X** in **X.Y.Z**).

Boost_VERSION_MINOR, Boost_MINOR_VERSION

Boost minor version number (**Y** in **X.Y.Z**).

Boost_VERSION_PATCH, Boost_SUBMINOR_VERSION

Boost subminor version number (**Z** in **X.Y.Z**).

Boost_VERSION_COUNT

Amount of version components (3).

Boost_LIB_DIAGNOSTIC_DEFINITIONS (Windows-specific)

Pass to *add_definitions()* to have diagnostic information about Boost's automatic linking displayed during compilation

Added in version 3.15: The **Boost_VERSION_<PART>** variables.

Cache variables

Search results are saved persistently in CMake cache entries:

Boost_INCLUDE_DIR

Directory containing Boost headers.

Boost_LIBRARY_DIR_RELEASE

Directory containing release Boost libraries.

Boost_LIBRARY_DIR_DEBUG

Directory containing debug Boost libraries.

Boost_<COMPONENT>_LIBRARY_DEBUG

Component **<COMPONENT>** library debug variant.

Boost_<COMPONENT>_LIBRARY_RELEASE

Component **<COMPONENT>** library release variant.

Added in version 3.3: Per-configuration variables **Boost_LIBRARY_DIR_RELEASE** and **Boost_LIBRARY_DIR_DEBUG**.

Hints

This module reads hints about search locations from variables:

BOOST_ROOT, BOOSTROOT

Preferred installation prefix.

BOOST_INCLUDEDIR

Preferred include directory e.g. **<prefix>/include**.

BOOST_LIBRARYDIR

Preferred library directory e.g. **<prefix>/lib**.

Boost_NO_SYSTEM_PATHS

Set to **ON** to disable searching in locations not specified by these hint variables. Default is **OFF**.

Boost_ADDITIONAL_VERSIONS

List of Boost versions not known to this module. (Boost install locations may contain the version).

Users may set these hints or results as **CACHE** entries. Projects should not read these entries directly but instead use the above result variables. Note that some hint names start in upper-case **BOOST**. One may specify these as environment variables if they are not specified as CMake variables or cache entries.

This module first searches for the Boost header files using the above hint variables (excluding **BOOST_LIBRARYDIR**) and saves the result in **Boost_INCLUDE_DIR**. Then it searches for requested component

libraries using the above hints (excluding **BOOST_INCLUDEDIR** and **Boost_ADDITIONAL_VERSIONS**), "lib" directories near **Boost_INCLUDE_DIR**, and the library name configuration settings below. It saves the library directories in **Boost_LIBRARY_DIR_DEBUG** and **Boost_LIBRARY_DIR_RELEASE** and individual library locations in **Boost_<COMPONENT>_LIBRARY_DEBUG** and **Boost_<COMPONENT>_LIBRARY_RELEASE**. When one changes settings used by previous searches in the same build tree (excluding environment variables) this module discards previous search results affected by the changes and searches again.

Imported Targets

Added in version 3.5.

This module defines the following *IMPORTED* targets:

Boost::boost

Target for header-only dependencies. (Boost include directory).

Boost::headers

Added in version 3.15: Alias for **Boost::boost**.

Boost::<component>

Target for specific component dependency (shared or static library); **<component>** name is lower-case.

Boost::diagnostic_definitions

Interface target to enable diagnostic information about Boost's automatic linking during compilation (adds **-DBOOST_LIB_DIAGNOSTIC**).

Boost::disable_autolinking

Interface target to disable automatic linking with MSVC (adds **-DBOOST_ALL_NO_LIB**).

Boost::dynamic_linking

Interface target to enable dynamic linking with MSVC (adds **-DBOOST_ALL_DYN_LINK**).

Implicit dependencies such as **Boost::filesystem** requiring **Boost::system** will be automatically detected and satisfied, even if system is not specified when using *find_package()* and if **Boost::system** is not added to *target_link_libraries()*. If using **Boost::thread**, then **Threads::Threads** will also be added automatically.

It is important to note that the imported targets behave differently than variables created by this module: multiple calls to *find_package(Boost)* in the same directory or sub-directories with different options (e.g. static or shared) will not override the values of the targets created by the first call.

Other Variables

Boost libraries come in many variants encoded in their file name. Users or projects may tell this module which variant to find by setting variables:

Boost_USE_DEBUG_LIBS

Added in version 3.10.

Set to **ON** or **OFF** to specify whether to search and use the debug libraries. Default is **ON**.

Boost_USE_RELEASE_LIBS

Added in version 3.10.

Set to **ON** or **OFF** to specify whether to search and use the release libraries. Default is **ON**.

Boost_USE_MULTITHREADED

Set to OFF to use the non-multithreaded libraries ("mt" tag). Default is **ON**.

Boost_USE_STATIC_LIBS

Set to ON to force the use of the static libraries. Default is **OFF**.

Boost_USE_STATIC_RUNTIME

Set to **ON** or **OFF** to specify whether to use libraries linked statically to the C++ runtime ("s" tag). Default is platform dependent.

Boost_USE_DEBUG_RUNTIME

Set to **ON** or **OFF** to specify whether to use libraries linked to the MS debug C++ runtime ("g" tag). Default is **ON**.

Boost_USE_DEBUG_PYTHON

Set to **ON** to use libraries compiled with a debug Python build ("y" tag). Default is **OFF**.

Boost_USE_STLPORT

Set to **ON** to use libraries compiled with STLPort ("p" tag). Default is **OFF**.

Boost_USE_STLPORT_DEPRECATED_NATIVE_IOSTREAMS

Set to ON to use libraries compiled with STLPort deprecated "native iostreams" ("n" tag). Default is **OFF**.

Boost_COMPILER

Set to the compiler-specific library suffix (e.g. **-gcc43**). Default is auto-computed for the C++ compiler in use.

Changed in version 3.9: A list may be used if multiple compatible suffixes should be tested for, in decreasing order of preference.

Boost_LIB_PREFIX

Added in version 3.18.

Set to the platform-specific library name prefix (e.g. **lib**) used by Boost static libs. This is needed only on platforms where CMake does not know the prefix by default.

Boost_ARCHITECTURE

Added in version 3.13.

Set to the architecture-specific library suffix (e.g. **-x64**). Default is auto-computed for the C++ compiler in use.

Boost_THREADAPI

Suffix for **thread** component library name, such as **pthread** or **win32**. Names with and without this suffix will both be tried.

Boost_NAMESPACE

Alternate namespace used to build boost with e.g. if set to **myboost**, will search for **my-boost_thread** instead of **boost_thread**.

Other variables one may set to control this module are:

Boost_DEBUG

Set to **ON** to enable debug output from **FindBoost**. Please enable this before filing any bug report.

Boost_REALPATH

Set to **ON** to resolve symlinks for discovered libraries to assist with packaging. For example, the "system" component library may be resolved to **/usr/lib/libboost_system.so.1.67.0** instead of

`/usr/lib/libboost_system.so`. This does not affect linking and should not be enabled unless the user needs this information.

Boost_LIBRARY_DIR

Default value for **Boost_LIBRARY_DIR_RELEASE** and **Boost_LIBRARY_DIR_DEBUG**.

Boost_NO_WARN_NEW_VERSIONS

Added in version 3.20.

Set to **ON** to suppress the warning about unknown dependencies for new Boost versions.

On Visual Studio and Borland compilers Boost headers request automatic linking to corresponding libraries. This requires matching libraries to be linked explicitly or available in the link library search path. In this case setting **Boost_USE_STATIC_LIBS** to **OFF** may not achieve dynamic linking. Boost automatic linking typically requests static libraries with a few exceptions (such as **Boost.Python**). Use:

```
add_definitions(${Boost_LIB_DIAGNOSTIC_DEFINITIONS})
```

to ask Boost to report information about automatic linking requests.

Examples

Find Boost headers only:

```
find_package(Boost 1.36.0)
if(Boost_FOUND)
    include_directories(${Boost_INCLUDE_DIRS})
    add_executable(foo foo.cc)
endif()
```

Find Boost libraries and use imported targets:

```
find_package(Boost 1.56 REQUIRED COMPONENTS
    date_time filesystem iostreams)
add_executable(foo foo.cc)
target_link_libraries(foo Boost::date_time Boost::filesystem
    Boost::iostreams)
```

Find Boost Python 3.6 libraries and use imported targets:

```
find_package(Boost 1.67 REQUIRED COMPONENTS
    python36 numpy36)
add_executable(foo foo.cc)
target_link_libraries(foo Boost::python36 Boost::numpy36)
```

Find Boost headers and some *static* (release only) libraries:

```
set(Boost_USE_STATIC_LIBS      ON)  # only find static libs
set(Boost_USE_DEBUG_LIBS      OFF)  # ignore debug libs and
set(Boost_USE_RELEASE_LIBS     ON)  # only find release libs
set(Boost_USE_MULTITHREADED    ON)
set(Boost_USE_STATIC_RUNTIME   OFF)
find_package(Boost 1.66.0 COMPONENTS date_time filesystem system ...)
if(Boost_FOUND)
    include_directories(${Boost_INCLUDE_DIRS})
    add_executable(foo foo.cc)
```

```
target_link_libraries(foo ${Boost_LIBRARIES})
endif()
```

Boost CMake

If Boost was built using the boost-cmake project or from Boost 1.70.0 on it provides a package configuration file for use with find_package's config mode. This module looks for the package configuration file called **BoostConfig.cmake** or **boost-config.cmake** and stores the result in **CACHE** entry **Boost_DIR**. If found, the package configuration file is loaded and this module returns with no further action. See documentation of the Boost CMake package configuration for details on what it provides.

Set **Boost_NO_BOOST_CMAKE** to **ON**, to disable the search for boost-cmake.

FindCUDA

Changed in version 3.27: This module is available only if policy *CMP0146* is not set to **NEW**. Port projects to CMake's first-class **CUDA** language support.

Deprecated since version 3.10: Do not use this module in new code.

It is no longer necessary to use this module or call **find_package(CUDA)** for compiling CUDA code. Instead, list **CUDA** among the languages named in the top-level call to the *project()* command, or call the *enable_language()* command with **CUDA**. Then one can add CUDA (.cu) sources directly to targets similar to other languages.

Added in version 3.17: To find and use the CUDA toolkit libraries manually, use the *FindCUDAToolkit* module instead. It works regardless of the **CUDA** language being enabled.

Documentation of Deprecated Usage

Tools for building CUDA C files: libraries and build dependencies.

This script locates the NVIDIA CUDA C tools. It should work on Linux, Windows, and macOS and should be reasonably up to date with CUDA C releases.

Added in version 3.19: QNX support.

This script makes use of the standard *find_package()* arguments of **<VERSION>**, **REQUIRED** and **QUIET**. **CUDA_FOUND** will report if an acceptable version of CUDA was found.

The script will prompt the user to specify **CUDA_TOOLKIT_ROOT_DIR** if the prefix cannot be determined by the location of nvcc in the system path and **REQUIRED** is specified to *find_package()*. To use a different installed version of the toolkit set the environment variable **CUDA_BIN_PATH** before running cmake (e.g. **CUDA_BIN_PATH=/usr/local/cuda1.0** instead of the default **/usr/local/cuda**) or set **CUDA_TOOLKIT_ROOT_DIR** after configuring. If you change the value of **CUDA_TOOLKIT_ROOT_DIR**, various components that depend on the path will be relocated.

It might be necessary to set **CUDA_TOOLKIT_ROOT_DIR** manually on certain platforms, or to use a CUDA runtime not installed in the default location. In newer versions of the toolkit the CUDA library is included with the graphics driver -- be sure that the driver version matches what is needed by the CUDA runtime version.

Input Variables

The following variables affect the behavior of the macros in the script (in alphabetical order). Note that any of these flags can be changed multiple times in the same directory before calling **cuda_add_executable()**, **cuda_add_library()**, **cuda_compile()**, **cuda_compile_ptx()**, **cuda_compile_fatbin()**,

cuda_compile_cubin() or **cuda_wrap_srcs()**:

CUDA_64_BIT_DEVICE_CODE (Default: host bit size)

Set to **ON** to compile for 64 bit device code, **OFF** for 32 bit device code. Note that making this different from the host code when generating object or C files from CUDA code just won't work, because `size_t` gets defined by `nvcc` in the generated source. If you compile to PTX and then load the file yourself, you can mix bit sizes between device and host.

CUDA_ATTACH_VS_BUILD_RULE_TO_CUDA_FILE (Default: ON)

Set to **ON** if you want the custom build rule to be attached to the source file in Visual Studio. Turn **OFF** if you add the same cuda file to multiple targets.

This allows the user to build the target from the CUDA file; however, bad things can happen if the CUDA source file is added to multiple targets. When performing parallel builds it is possible for the custom build command to be run more than once and in parallel causing cryptic build errors. VS runs the rules for every source file in the target, and a source can have only one rule no matter how many projects it is added to. When the rule is run from multiple targets race conditions can occur on the generated file. Eventually everything will get built, but if the user is unaware of this behavior, there may be confusion. It would be nice if this script could detect the reuse of source files across multiple targets and turn the option off for the user, but no good solution could be found.

CUDA_BUILD_CUBIN (Default: OFF)

Set to **ON** to enable an extra compilation pass with the `-cubin` option in Device mode. The output is parsed and register, shared memory usage is printed during build.

CUDA_BUILD_EMULATION (Default: OFF for device mode)

Set to **ON** for Emulation mode. `-D_DEVICEEMU` is defined for CUDA C files when **CUDA_BUILD_EMULATION** is **TRUE**.

CUDA_LINK_LIBRARIES_KEYWORD (Default: "")

Added in version 3.9.

The `<PRIVATE|PUBLIC|INTERFACE>` keyword to use for internal `target_link_libraries()` calls. The default is to use no keyword which uses the old "plain" form of `target_link_libraries()`. Note that it matters because whatever is used inside the **FindCUDA** module must also be used outside – the two forms of `target_link_libraries()` cannot be mixed.

CUDA_GENERATED_OUTPUT_DIR (Default: CMAKE_CURRENT_BINARY_DIR)

Set to the path you wish to have the generated files placed. If it is blank output files will be placed in `CMAKE_CURRENT_BINARY_DIR`. Intermediate files will always be placed in `CMAKE_CURRENT_BINARY_DIR/CMakeFiles`.

CUDA_HOST_COMPILATION_CPP (Default: ON)

Set to **OFF** for C compilation of host code.

CUDA_HOST_COMPILER (Default: CMAKE_C_COMPILER)

Set the host compiler to be used by `nvcc`. Ignored if `-ccbin` or `--compiler-bindir` is already present in the **CUDA_NVCC_FLAGS** or **CUDA_NVCC_FLAGS_<CONFIG>** variables. For Visual Studio targets, the host compiler is constructed with one or more visual studio macros such as `$(VCInstallDir)`, that expands out to the path when the command is run from within VS.

Added in version 3.13: If the `CUDAHOSTCXX` environment variable is set it will be used as the default.

CUDA_NVCC_FLAGS, CUDA_NVCC_FLAGS_<CONFIG>

Additional NVCC command line arguments. NOTE: multiple arguments must be semi-colon delimited (e.g. `--compiler-options;-Wall`)

Added in version 3.6: Contents of these variables may use *generator expressions*.

CUDA_PROPAGATE_HOST_FLAGS (Default: ON)

Set to **ON** to propagate `CMAKE_{C,CXX}_FLAGS` and their configuration dependent counterparts (e.g. `CMAKE_C_FLAGS_DEBUG`) automatically to the host compiler through nvcc's `-Xcompiler` flag. This helps make the generated host code match the rest of the system better. Sometimes certain flags give nvcc problems, and this will help you turn the flag propagation off. This does not affect the flags supplied directly to nvcc via `CUDA_NVCC_FLAGS` or through the **OPTION** flags specified through `cuda_add_library()`, `cuda_add_executable()`, or `cuda_wrap_srcs()`. Flags used for shared library compilation are not affected by this flag.

CUDA_SEPARABLE_COMPILATION (Default: OFF)

If set this will enable separable compilation for all CUDA runtime object files. If used outside of `cuda_add_executable()` and `cuda_add_library()` (e.g. calling `cuda_wrap_srcs()` directly), `cuda_compute_separable_compilation_object_file_name()` and `cuda_link_separable_compilation_objects()` should be called.

CUDA_SOURCE_PROPERTY_FORMAT

Added in version 3.3.

If this source file property is set, it can override the format specified to `cuda_wrap_srcs()` (**OBJ**, **PTX**, **CUBIN**, or **FATBIN**). If an input source file is not a `.cu` file, setting this file will cause it to be treated as a `.cu` file. See documentation for `set_source_files_properties` on how to set this property.

CUDA_USE_STATIC_CUDA_RUNTIME (Default: ON)

Added in version 3.3.

When enabled the static version of the CUDA runtime library will be used in **CUDA_LIBRARIES**. If the version of CUDA configured doesn't support this option, then it will be silently disabled.

CUDA_VERBOSE_BUILD (Default: OFF)

Set to **ON** to see all the commands used when building the CUDA file. When using a Makefile generator the value defaults to **VERBOSE** (run `make VERBOSE=1` to see output), although setting `CUDA_VERBOSE_BUILD` to **ON** will always print the output.

Commands

The script creates the following functions and macros (in alphabetical order):

```
cuda_add_cufft_to_target(<cuda_target>)
```

Adds the cufft library to the target (can be any target). Handles whether you are in emulation mode or not.

```
cuda_add_cublas_to_target(<cuda_target>)
```

Adds the cublas library to the target (can be any target). Handles whether you are in emulation mode or not.

```
cuda_add_executable(<cuda_target> <file>...
                  [WIN32] [MACOSX_BUNDLE] [EXCLUDE_FROM_ALL] [OPTIONS ...])
```

Creates an executable `<cuda_target>` which is made up of the files specified. All of the non CUDA C files are compiled using the standard build rules specified by CMake and the CUDA files are compiled to object files using nvcc and the host compiler. In addition `CUDA_INCLUDE_DIRS` is added automatically to

include_directories(). Some standard CMake target calls can be used on the target after calling this macro (e.g. *set_target_properties()* and *target_link_libraries()*), but setting properties that adjust compilation flags will not affect code compiled by nvcc. Such flags should be modified before calling **cuda_add_executable()**, **cuda_add_library()** or **cuda_wrap_srcs()**.

```
cuda_add_library(<cuda_target> <file>...
                [STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL] [OPTIONS ...])
```

Same as **cuda_add_executable()** except that a library is created.

```
cuda_build_clean_target()
```

Creates a convenience target that deletes all the dependency files generated. You should make clean after running this target to ensure the dependency files get regenerated.

```
cuda_compile(<generated_files> <file>... [STATIC | SHARED | MODULE]
             [OPTIONS ...])
```

Returns a list of generated files from the input source files to be used with *add_library()* or *add_executable()*.

```
cuda_compile_ptx(<generated_files> <file>... [OPTIONS ...])
```

Returns a list of **PTX** files generated from the input source files.

```
cuda_compile_fatbin(<generated_files> <file>... [OPTIONS ...])
```

Added in version 3.1.

Returns a list of **FATBIN** files generated from the input source files.

```
cuda_compile_cubin(<generated_files> <file>... [OPTIONS ...])
```

Added in version 3.1.

Returns a list of **CUBIN** files generated from the input source files.

```
cuda_compute_separable_compilation_object_file_name(<output_file_var>
                                                    <cuda_target>
                                                    <object_files>)
```

Compute the name of the intermediate link file used for separable compilation. This file name is typically passed into **CUDA_LINK_SEPARABLE_COMPILATION_OBJECTS**. *output_file_var* is produced based on *cuda_target* the list of objects files that need separable compilation as specified by **<object_files>**. If the **<object_files>** list is empty, then **<output_file_var>** will be empty. This function is called automatically for **cuda_add_library()** and **cuda_add_executable()**. Note that this is a function and not a macro.

```
cuda_include_directories(path0 path1 ...)
```

Sets the directories that should be passed to nvcc (e.g. **nvcc -Ipath0 -Ipath1 ...**). These paths usually contain other **.cu** files.

```
cuda_link_separable_compilation_objects(<output_file_var> <cuda_target>
                                         <nvcc_flags> <object_files>)
```

Generates the link object required by separable compilation from the given object files. This is called automatically for **cuda_add_executable()** and **cuda_add_library()**, but can be called manually when using **cuda_wrap_srcs()** directly. When called from **cuda_add_library()** or **cuda_add_executable()** the **<nvcc_flags>** passed in are the same as the flags passed in via the **OPTIONS** argument. The only nvcc flag added automatically is the bitness flag as specified by **CUDA_64_BIT_DEVICE_CODE**. Note that this is a function instead of a macro.

```
cuda_select_nvcc_arch_flags(<out_variable> [<target_CUDA_architecture> ...])
```

Selects GPU arch flags for nvcc based on **target_CUDA_architecture**.

Values for **target_CUDA_architecture**:

- **Auto**: detects local machine GPU compute arch at runtime.
- **Common** and **All**: cover common and entire subsets of architectures.
- **<name>**: one of **Fermi**, **Kepler**, **Maxwell**, **Kepler+Tegra**, **Kepler+Tesla**, **Maxwell+Tegra**, **Pascal**.
- **<ver>**, **<ver>(<ver>)**, **<ver>+PTX**, where **<ver>** is one of **2.0**, **2.1**, **3.0**, **3.2**, **3.5**, **3.7**, **5.0**, **5.2**, **5.3**, **6.0**, **6.2**.

Returns list of flags to be added to **CUDA_NVCC_FLAGS** in **<out_variable>**. Additionally, sets **<out_variable>_readable** to the resulting numeric list.

Example:

```
cuda_select_nvcc_arch_flags(ARCH_FLAGS "3.0" "3.5+PTX" "5.2(5.0)" "Maxwell")
list(APPEND CUDA_NVCC_FLAGS ${ARCH_FLAGS})
```

More info on CUDA architectures: <https://en.wikipedia.org/wiki/CUDA>. Note that this is a function instead of a macro.

```
cuda_wrap_srcs(<cuda_target> <format> <generated_files> <file>...
               [STATIC | SHARED | MODULE] [OPTIONS ...])
```

This is where all the magic happens. **cuda_add_executable()**, **cuda_add_library()**, **cuda_compile()**, and **cuda_compile_ptx()** all call this function under the hood.

Given the list of files **<file>...** this macro generates custom commands that generate either PTX or linkable objects (use **PTX** or **OBJ** for the **<format>** argument to switch). Files that don't end with **.cu** or have the **HEADER_FILE_ONLY** property are ignored.

The arguments passed in after **OPTIONS** are extra command line options to give to nvcc. You can also specify per configuration options by specifying the name of the configuration followed by the options. General options must precede configuration specific options. Not all configurations need to be specified, only the ones provided will be used. For example:

```
cuda_add_executable(...
  OPTIONS -DFLAG=2 "-DFLAG_OTHER=space in flag"
  DEBUG -g
  RELEASE --use_fast_math
  RELWITHDEBINFO --use_fast_math;-g
  MINSIZEREL --use_fast_math)
```

For certain configurations (namely VS generating object files with **CUDA_ATTACH_VS_BUILD_RULE_TO_CUDA_FILE** set to **ON**), no generated file will be produced for the given cuda file. This is because when you add the cuda file to Visual Studio it knows that this file produces an object file and will link in the resulting object file automatically.

This script will also generate a separate cmake script that is used at build time to invoke nvcc. This is for several reasons:

- nvcc can return negative numbers as return values which confuses Visual Studio into thinking that the command succeeded. The script now checks the error codes and produces errors when there was a problem.
- nvcc has been known to not delete incomplete results when it encounters problems. This confuses build systems into thinking the target was generated when in fact an unusable file exists. The script now deletes the output files if there was an error.
- By putting all the options that affect the build into a file and then make the build rule dependent on the file, the output files will be regenerated when the options change.

This script also looks at optional arguments **STATIC**, **SHARED**, or **MODULE** to determine when to target the object compilation for a shared library. **BUILD_SHARED_LIBS** is ignored in **cuda_wrap_srcs()**, but it is respected in **cuda_add_library()**. On some systems special flags are added for building objects intended for shared libraries. A preprocessor macro, **<target_name>_EXPORTS** is defined when a shared library compilation is detected.

Flags passed into add_definitions with **-D** or **/D** are passed along to nvcc.

Result Variables

The script defines the following variables:

CUDA_VERSION_MAJOR

The major version of cuda as reported by nvcc.

CUDA_VERSION_MINOR

The minor version.

CUDA_VERSION, CUDA_VERSION_STRING

Full version in the **X.Y** format.

CUDA_HAS_FP16

Added in version 3.6: Whether a short float (**float16**, **fp16**) is supported.

CUDA_TOOLKIT_ROOT_DIR

Path to the CUDA Toolkit (defined if not set).

CUDA_SDK_ROOT_DIR

Path to the CUDA SDK. Use this to find files in the SDK. This script will not directly support finding specific libraries or headers, as that isn't supported by NVIDIA. If you want to change libraries when the path changes see the **FindCUDA.cmake** script for an example of how to clear these variables. There are also examples of how to use the **CUDA_SDK_ROOT_DIR** to locate headers or libraries, if you so choose (at your own risk).

CUDA_INCLUDE_DIRS

Include directory for cuda headers. Added automatically for **cuda_add_executable()** and **cuda_add_library()**.

CUDA_LIBRARIES

Cuda RT library.

CUDA_CUFFT_LIBRARIES

Device or emulation library for the Cuda FFT implementation (alternative to `cuda_add_cufft_to_target()` macro)

CUDA_CUBLAS_LIBRARIES

Device or emulation library for the Cuda BLAS implementation (alternative to `cuda_add_cublas_to_target()` macro).

CUDA_cudart_static_LIBRARY

Statically linkable cuda runtime library. Only available for CUDA version 5.5+.

CUDA_cudadevrt_LIBRARY

Added in version 3.7: Device runtime library. Required for separable compilation.

CUDA_cupti_LIBRARY

CUDA Profiling Tools Interface library. Only available for CUDA version 4.0+.

CUDA_curand_LIBRARY

CUDA Random Number Generation library. Only available for CUDA version 3.2+.

CUDA_cusolver_LIBRARY

Added in version 3.2: CUDA Direct Solver library. Only available for CUDA version 7.0+.

CUDA_cusparselib_LIBRARY

CUDA Sparse Matrix library. Only available for CUDA version 3.2+.

CUDA_npp_LIBRARY

NVIDIA Performance Primitives lib. Only available for CUDA version 4.0+.

CUDA_nppc_LIBRARY

NVIDIA Performance Primitives lib (core). Only available for CUDA version 5.5+.

CUDA_nppi_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 5.5 – 8.0.

CUDA_nppial_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 9.0.

CUDA_nppicc_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 9.0.

CUDA_nppicom_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 9.0 – 10.2. Replaced by `nvjpeg`.

CUDA_nppidei_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 9.0.

CUDA_nppif_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 9.0.

CUDA_nppig_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 9.0.

CUDA_nppim_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 9.0.

CUDA_nppist_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 9.0.

CUDA_nppisu_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 9.0.

CUDA_nppitc_LIBRARY

NVIDIA Performance Primitives lib (image processing). Only available for CUDA version 9.0.

CUDA_npps_LIBRARY

NVIDIA Performance Primitives lib (signal processing). Only available for CUDA version 5.5+.

CUDA_nvcuenc_LIBRARY

CUDA Video Encoder library. Only available for CUDA version 3.2+. Windows only.

CUDA_nvcuvid_LIBRARY

CUDA Video Decoder library. Only available for CUDA version 3.2+. Windows only.

CUDA_nvToolsExt_LIBRARY

Added in version 3.16: NVIDIA CUDA Tools Extension library. Available for CUDA version 5+.

CUDA_OpenCL_LIBRARY

Added in version 3.16: NVIDIA CUDA OpenCL library. Available for CUDA version 5+.

FindDart

Deprecated since version 3.27: This module is available only if policy *CMP0145* is not set to **NEW**.

Find DART

This module looks for the dart testing software and sets DART_ROOT to point to where it found it.

FindITK

This module no longer exists.

This module existed in versions of CMake prior to 3.1, but became only a thin wrapper around **find_package(ITK NO_MODULE)** to provide compatibility for projects using long-outdated conventions. Now **find_package(ITK)** will search for **ITKConfig.cmake** directly.

FindPythonInterp

Changed in version 3.27: This module is available only if policy *CMP0148* is not set to **NEW**.

Deprecated since version 3.12: Use *FindPython3*, *FindPython2* or *FindPython* instead.

Find python interpreter

This module finds if Python interpreter is installed and determines where the executables are. This code sets the following variables:

PYTHONINTERP_FOUND	- Was the Python executable found
PYTHON_EXECUTABLE	- path to the Python interpreter
PYTHON_VERSION_STRING	- Python version found e.g. 2.5.2
PYTHON_VERSION_MAJOR	- Python major version found e.g. 2
PYTHON_VERSION_MINOR	- Python minor version found e.g. 5
PYTHON_VERSION_PATCH	- Python patch version found e.g. 2

The `PYTHON_ADDITIONAL_VERSIONS` variable can be used to specify a list of version numbers that should be taken into account when searching for Python. You need to set this variable before calling `find_package(PythonInterp)`.

If calling both **find_package(PythonInterp)** and **find_package(PythonLibs)**, call **find_package(PythonInterp)** first to get the currently active Python version by default with a consistent version of PYTHON_LIBRARIES.

NOTE:

A call to **find_package(PythonInterp \${V})** for python version **V** may find a **python** executable with no version suffix. In this case no attempt is made to avoid python executables from other versions. Use *FindPython3*, *FindPython2* or *FindPython* instead.

FindPythonLibs

Changed in version 3.27: This module is available only if policy *CMP0148* is not set to **NEW**.

Deprecated since version 3.12: Use *FindPython3*, *FindPython2* or *FindPython* instead.

Find python libraries

This module finds if Python is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

PYTHONLIBS_FOUND	- have the Python libs been found
PYTHON_LIBRARIES	- path to the python library
PYTHON_INCLUDE_PATH	- path to where Python.h is found (deprecated)
PYTHON_INCLUDE_DIRS	- path to where Python.h is found
PYTHON_DEBUG_LIBRARIES	- path to the debug library (deprecated)
PYTHONLIBS_VERSION_STRING	- version of the Python libs found (since CMake 2.8.8)

The `Python_ADDITIONAL_VERSIONS` variable can be used to specify a list of version numbers that should be taken into account when searching for Python. You need to set this variable before calling `find_package(PythonLibs)`.

If you'd like to specify the installation of Python to use, you should modify the following cache variables:

PYTHON_LIBRARY	- path to the python library
PYTHON_INCLUDE_DIR	- path to where Python.h is found

If calling both **find_package(PythonInterp)** and **find_package(PythonLibs)**, call **find_package(PythonInterp)** first to get the currently active Python version by default with a consistent version of PYTHON_LIBRARIES.

FindQt

Deprecated since version 3.14: This module is available only if policy *CMP0084* is not set to **NEW**.

Searches for all installed versions of Qt3 or Qt4.

This module cannot handle Qt5 or any later versions. For those, see *cmake-qt(7)*.

This module should only be used if your project can work with multiple versions of Qt. If not, you should just directly use *FindQt4* or *FindQt3*. If multiple versions of Qt are found on the machine, then The user must set the option `DESIRED_QT_VERSION` to the version they want to use. If only one version of qt is found on the machine, then the `DESIRED_QT_VERSION` is set to that version and the matching *FindQt3* or *FindQt4* module is included. Once the user sets `DESIRED_QT_VERSION`, then the *FindQt3* or *FindQt4* module is included.

QT_REQUIRED if this is set to TRUE then if CMake can not find Qt4 or Qt3 an error is raised and a message is sent to the user.

DESIRED_QT_VERSION_OPTION is created
 QT4_INSTALLED is set to TRUE if qt4 is found.
 QT3_INSTALLED is set to TRUE if qt3 is found.

FindUnixCommands

Deprecated since version 3.26: Use `${CMAKE_COMMAND} -E subcommands` instead.

Find Unix commands, including the ones from Cygwin

This module looks for the Unix commands **bash**, **cp**, **gzip**, **mv**, **rm**, and **tar** and stores the result in the variables **BASH**, **CP**, **GZIP**, **MV**, **RM**, and **TAR**.

FindVTK

This module no longer exists.

This module existed in versions of CMake prior to 3.1, but became only a thin wrapper around **find_package(VTK NO_MODULE)** to provide compatibility for projects using long-outdated conventions. Now **find_package(VTK)** will search for **VTKConfig.cmake** directly.

FindwxWindows

Deprecated since version 3.0: Replaced by *FindwxWidgets*.

Find wxWindows (wxWidgets) installation

This module finds if wxWindows/wxWidgets is installed and determines where the include files and libraries are. It also determines what the name of the library is. This code sets the following variables:

```
WXWINDOWS_FOUND      = system has WxWindows
WXWINDOWS_LIBRARIES  = path to the wxWindows libraries
                      on Unix/Linux with additional
                      linker flags from
                      "wx-config --libs"
CMAKE_WXWINDOWS_CXX_FLAGS = Compiler flags for wxWindows,
                      essentially "`wx-config --cxxflags`"
                      on Linux
WXWINDOWS_INCLUDE_DIR = where to find "wx/wx.h" and "wx/setup.h"
WXWINDOWS_LINK_DIRECTORIES = link directories, useful for rpath on
                          Unix
WXWINDOWS_DEFINITIONS  = extra defines
```

OPTIONS If you need OpenGL support please

```
set(WXWINDOWS_USE_GL 1)
```

in your CMakeLists.txt *before* you include this file.

```
HAVE_ISYSTEM      - true required to replace -I by -isystem on g++
```

For convenience include `Use_wxWindows.cmake` in your project's `CMakeLists.txt` using `include(${CMAKE_CURRENT_LIST_DIR}/Use_wxWindows.cmake)`.

USAGE

```
set(WXWINDOWS_USE_GL 1)
find_package(wxWindows)
```

NOTES wxWidgets 2.6.x is supported for monolithic builds e.g. compiled in wx/build/msw dir as:

```
nmake -f makefile.vc BUILD=debug SHARED=0 USE_OPENGL=1 MONOLITHIC=1
```

DEPRECATED

```
CMAKE_WX_CAN_COMPILE
WXWINDOWS_LIBRARY
CMAKE_WX_CXX_FLAGS
WXWINDOWS_INCLUDE_PATH
```

AUTHOR Jan Woetzel (07/2003–01/2006)

Legacy CPack Modules

These modules used to be mistakenly exposed to the user, and have been moved out of user visibility. They are for CPack internal use, and should never be used directly.

CPackArchive

Added in version 3.9.

The documentation for the CPack Archive generator has moved here: *CPack Archive Generator*

CPackBundle

The documentation for the CPack Bundle generator has moved here: *CPack Bundle Generator*

CPackCygwin

The documentation for the CPack Cygwin generator has moved here: *CPack Cygwin Generator*

CPackDeb

The documentation for the CPack DEB generator has moved here: *CPack DEB Generator*

CPackDMG

The documentation for the CPack DragNDrop generator has moved here: *CPack DragNDrop Generator*

CPackFreeBSD

Added in version 3.10.

The documentation for the CPack FreeBSD generator has moved here: *CPack FreeBSD Generator*

CPackNSIS

The documentation for the CPack NSIS generator has moved here: *CPack NSIS Generator*

CPackNuGet

Added in version 3.12.

The documentation for the CPack NuGet generator has moved here: *CPack NuGet Generator*

CPackProductBuild

Added in version 3.7.

The documentation for the CPack productbuild generator has moved here: *CPack productbuild Generator*

CPackRPM

The documentation for the CPack RPM generator has moved here: *CPack RPM Generator*

CPackWIX

The documentation for the CPack WIX generator has moved here: *CPack WIX Generator*

COPYRIGHT

2000-2025 Kitware, Inc. and Contributors