cmake-developer(7)¶

Contents

Introduction

This manual is intended for reference by developers modifying the CMake source tree itself, and by those authoring externally-maintained modules.

Adding Compile Features

CMake reports an error if a compiler whose features are known does not report support for a particular requested feature. A compiler is considered to have known features if it reports support for at least one feature.

When adding a new compile feature to CMake, it is therefore necessary to list support for the feature for all CompilerIds which already have one or more feature supported, if the new feature is available for any version of the compiler.

When adding the first supported feature to a particular CompilerId, it is necessary to list support for all features known to cmake (See `CMAKE_C_COMPILE_FEATURES` and `CMAKE_CXX_COMPILE_FEATURES` as appropriate), where available for the compiler. Ensure that the `CMAKE_<LANG>_STANDARD_DEFAULT` is set to the computed internal variable `CMAKE_<LANG>_STANDARD_COMPUTED_DEFAULT` for compiler versions which should be supported.

It is sensible to record the features for the most recent version of a particular CompilerId first, and then work backwards. It is sensible to try to create a continuous range of versions of feature releases of the compiler. Gaps in the range indicate incorrect features recorded for intermediate releases.

Generally, features are made available for a particular version if the compiler vendor documents availability of the feature with that version. Note that sometimes partially implemented features appear to be functional in previous releases (such as `cxx_constexpr` in GNU 4.6, though availability is documented in GNU 4.7), and sometimes compiler vendors document availability of features, though supporting infrastructure is not available (such as `__has_feature(cxx_generic_lambdas)` indicating non-availability in Clang 3.4, though it is documented as available, and fixed in Clang 3.5). Similar cases for other compilers and versions need to be investigated when extending CMake to support them.

When a vendor releases a new version of a known compiler which supports a previously unsupported feature, and there are already known features for that compiler, the feature should be listed as supported in CMake for that version of the compiler as soon as reasonably possible.

Standard-specific/compiler-specific variables such `CMAKE_CXX98_COMPILE_FEATURES` are deliberately not documented. They only exist for the compiler-specific implementation of adding the `-std` compile flag for compilers which need that.

Help

The `Help` directory contains CMake help manual source files. They are written using the reStructuredText markup syntax and processed by Sphinx to generate the CMake help manuals.

Markup Constructs

In addition to using Sphinx to generate the CMake help manuals, we also use a C++-implemented document processor to print documents for the `--help-*` command-line help options. It supports a subset of reStructuredText markup. When authoring or modifying documents, please verify that the command-line help looks good in addition to the Sphinx-generated html and man pages.

The command-line help processor supports the following constructs defined by reStructuredText, Sphinx, and a CMake extension to Sphinx.

- CMake Domain directives

  Directives defined in the CMake Domain for defining CMake documentation objects are printed in command-line help output as if the lines were normal paragraph text with interpretation.

- CMake Domain interpreted text roles

Interpreted text roles defined in the CMake Domain for cross-referencing CMake documentation objects are replaced by their link text in command-line help output. Other roles are printed literally and not processed.

- `code-block` directive

  Add a literal code block without interpretation. The command-line help processor prints the block content without the leading directive line and with common indentation replaced by one space.

- `include` directive

  Include another document source file. The command-line help processor prints the included document inline with the referencing document.

- literal block after `::`

  A paragraph ending in `::` followed by a blank line treats the following indented block as literal text without interpretation. The command-line help processor prints the `::` literally and prints the block content with common indentation replaced by one space.

- `note` directive

  Call out a side note. The command-line help processor prints the block content as if the lines were normal paragraph text with interpretation.

- `parsed-literal` directive

  Add a literal block with markup interpretation. The command-line help processor prints the block content without the leading directive line and with common indentation replaced by one space.

- `productionlist` directive

  Render context-free grammar productions. The command-line help processor prints the block content as if the lines were normal paragraph text with interpretation.

- `replace` directive

  Define a `|substitution|` replacement. The command-line help processor requires a substitution replacement to be defined before it is referenced.

- `|substitution|` reference

  Reference a substitution replacement previously defined by the `replace` directive. The command-line help processor performs the substitution and replaces all newlines in the replacement text with spaces.

- `toctree` directive

  Include other document sources in the Table-of-Contents document tree. The command-line help processor prints the referenced documents inline as part of the referencing document.

Inline markup constructs not listed above are printed literally in the command-line help output. We prefer to use inline markup constructs that look correct in source form, so avoid use of -escapes in favor of inline literals when possible.

Explicit markup blocks not matching directives listed above are removed from command-line help output. Do not use them, except for plain `..` comments that are removed by Sphinx too.

Note that nested indentation of blocks is not recognized by the command-line help processor. Therefore:

- Explicit markup blocks are recognized only when not indented inside other blocks.

- Literal blocks after paragraphs ending in `::` but not at the top indentation level may consume all indented lines following them.

Try to avoid these cases in practice.

CMake Domain

CMake adds a Sphinx Domain called `cmake`, also called the “CMake Domain”. It defines several “object” types for CMake documentation:

- `command`

  A CMake language command.

- `generator`

  A CMake native build system generator. See the `cmake(1)` command-line tool’s `-G` option.

- `manual`

  A CMake manual page, like this `cmake-developer(7)` manual.

- `module`

  A CMake module. See the `cmake-modules(7)` manual and the `include()` command.

- `policy`

  A CMake policy. See the `cmake-policies(7)` manual and the `cmake_policy()` command.

- `prop_cache, prop_dir, prop_gbl, prop_sf, prop_inst, prop_test, prop_tgt`

  A CMake cache, directory, global, source file, installed file, test, or target property, respectively. See the `cmake-properties(7)` manual and the `set_property()` command.

- `variable`

  A CMake language variable. See the `cmake-variables(7)` manual and the `set()` command.

Documentation objects in the CMake Domain come from two sources. First, the CMake extension to Sphinx transforms every document named with the form `Help/<type>/<file-name>.rst` to a domain object with type `<type>`. The object name is extracted from the document title, which is expected to be of the form:

```
<object-name>
-------------
```

and to appear at or near the top of the `.rst` file before any other lines starting in a letter, digit, or `<`. If no such title appears literally in the `.rst` file, the object name is the `<file-name>`. If a title does appear, it is expected that `<file-name>` is equal to `<object-name>` with any `<` and `>` characters removed.

Second, the CMake Domain provides directives to define objects inside other documents:

```
.. command:: <command-name>

 This indented block documents <command-name>.

.. variable:: <variable-name>

 This indented block documents <variable-name>.
```

Object types for which no directive is available must be defined using the first approach above.

Cross-References

Sphinx uses reStructuredText interpreted text roles to provide cross-reference syntax. The CMake Domainprovides for each domain object type a role of the same name to cross-reference it. CMake Domain roles are inline markup of the forms:

```
:type:`name`
:type:`text <name>`
```

where `type` is the domain object type and `name` is the domain object name. In the first form the link text will be `name` (or `name()` if the type is `command`) and in the second form the link text will be the explicit `text`. For example, the code:

```
* The :command:`list` command.
* The :command:`list(APPEND)` sub-command.
* The :command:`list() command <list>`.
* The :command:`list(APPEND) sub-command <list>`.
* The :variable:`CMAKE_VERSION` variable.
* The :prop_tgt:`OUTPUT_NAME_<CONFIG>` target property.
```

produces:

- The `list()` command.
- The `list(APPEND)` sub-command.
- The `list() command`.
- The `list(APPEND) sub-command`.
- The `CMAKE_VERSION` variable.
- The `OUTPUT_NAME_` target property.

Note that CMake Domain roles differ from Sphinx and reStructuredText convention in that the form `a<b>`, without a space preceding `<`, is interpreted as a name instead of link text with an explicit target. This is necessary because we use `<placeholders>` frequently in object names like `OUTPUT_NAME_<CONFIG>`. The form `a <b>`, with a space preceding `<`, is still interpreted as a link text with an explicit target.

Style

Style: Section Headers

When marking section titles, make the section decoration line as long as the title text. Use only a line below the title, not above. For example:

```
Title Text
----------
```

Capitalize the first letter of each non-minor word in the title.

The section header underline character hierarchy is

- `#`: Manual group (part) in the master document
- `*`: Manual (chapter) title
- `=`: Section within a manual
- `-`: Subsection or CMake Domain object document title
- `^`: Subsubsection or CMake Domain object document section
- `"`: Paragraph or CMake Domain object document subsection

Style: Whitespace

Use two spaces for indentation. Use two spaces between sentences in prose.

Style: Line Length

Prefer to restrict the width of lines to 75-80 columns. This is not a hard restriction, but writing new paragraphs wrapped at 75 columns allows space for adding minor content without significant re-wrapping of content.

Style: Prose

Use American English spellings in prose.

Style: Starting Literal Blocks

Prefer to mark the start of literal blocks with `::` at the end of the preceding paragraph. In cases where the following block gets a `code-block` marker, put a single `:` at the end of the preceding paragraph.

Style: CMake Command Signatures

Command signatures should be marked up as plain literal blocks, not as cmake `code-blocks`.

Signatures are separated from preceding content by a section header. That is, use:

```
... preceding paragraph.

Normal Libraries
^^^^^^^^^^^^^^^^

::

  add_library(<lib> ...)

This signature is used for ...
```

Signatures of commands should wrap optional parts with square brackets, and should mark list of optional arguments with an ellipsis (`...`). Elements of the signature which are specified by the user should be specified with angle brackets, and may be referred to in prose using `inline-literal` syntax.

Style: Boolean Constants

Use "`OFF`" and "`ON`" for boolean values which can be modified by the user, such as `POSITION_INDEPENDENT_CODE`. Such properties may be "enabled" and "disabled". Use "`True`" and "`False`" for inherent values which can't be modified after being set, such as the `IMPORTED` property of a build target.

Style: Inline Literals

Mark up references to keywords in signatures, file names, and other technical terms with `inline-literal` syntax, for example:

```
If ``WIN32`` is used with :command:`add_executable`, the
:prop_tgt:`WIN32_EXECUTABLE` target property is enabled. That command
creates the file ``<name>.exe`` on Windows.
```

Style: Cross-References

Mark up linkable references as links, including repeats. An alternative, which is used by wikipedia (http://en.wikipedia.org/wiki/WP:REPEATLINK), is to link to a reference only once per article. That style is not used in CMake documentation.

Style: Referencing CMake Concepts

If referring to a concept which corresponds to a property, and that concept is described in a high-level manual, prefer to link to the manual section instead of the property. For example:

```
This command creates an :ref:`Imported Target <Imported Targets>`.
```

instead of:

```
This command creates an :prop_tgt:`IMPORTED` target.
```

The latter should be used only when referring specifically to the property.

References to manual sections are not automatically created by creating a section, but code such as:

```
.. _`Imported Targets`:
```

creates a suitable anchor. Use an anchor name which matches the name of the corresponding section. Refer to the anchor using a cross-reference with specified text.

Imported Targets need the `IMPORTED` term marked up with care in particular because the term may refer to a command keyword (`IMPORTED`), a target property (`IMPORTED`), or a concept (Imported Targets).

Where a property, command or variable is related conceptually to others, by for example, being related to the buildsystem description, generator expressions or Qt, each relevant property, command or variable should link to the primary manual, which provides high-level information. Only particular information relating to the command should be in the documentation of the command.

Style: Referencing CMake Domain Objects

When referring to CMake Domain objects such as properties, variables, commands etc, prefer to link to the target object and follow that with the type of object it is. For example:

```
Set the :prop_tgt:`AUTOMOC` target property to ``ON``.
```

Instead of

```
Set the target property :prop_tgt:`AUTOMOC` to ``ON``.
```

The `policy` directive is an exception, and the type us usually referred to before the link:

```
If policy :prop_tgt:`CMP0022` is set to ``NEW`` the behavior is ...
```

However, markup self-references with `inline-literal` syntax. For example, within the `add_executable()` command documentation, use

```
``add_executable``
```

not

```
:command:`add_executable`
```

which is used elsewhere.

Modules

The `Modules` directory contains CMake-language `.cmake` module files.

Module Documentation

To document CMake module `Modules/<module-name>.cmake`, modify `Help/manual/cmake-modules.7.rst` to reference the module in the `toctree` directive, in sorted order, as:

```
/module/<module-name>
```

Then add the module document file `Help/module/<module-name>.rst` containing just the line:

```
.. cmake-module:: ../../Modules/<module-name>.cmake
```

The `cmake-module` directive will scan the module file to extract reStructuredText markup from comment blocks that start in `.rst:`. At the top of `Modules/<module-name>.cmake`, begin with the following license notice:

```
# Distributed under the OSI-approved BSD 3-Clause License.  See accompanying
# file Copyright.txt or https://cmake.org/licensing for details.
```

After this notice, add a *BLANK* line. Then, add documentation using a Line Comment block of the form:

```
#.rst:
# <module-name>
# -------------
#
# <reStructuredText documentation of module>
```

or a Bracket Comment of the form:

```
#[[.rst:
<module-name>
-------------

<reStructuredText documentation of module>
#]]
```

Any number of `=` may be used in the opening and closing brackets as long as they match. Content on the line containing the closing bracket is excluded if and only if the line starts in `#`.

Additional such `.rst:` comments may appear anywhere in the module file. All such comments must start with `#` in the first column.

For example, a `Modules/Findxxx.cmake` module may contain:

```
# Distributed under the OSI-approved BSD 3-Clause License.  See accompanying
# file Copyright.txt or https://cmake.org/licensing for details.

#.rst:
# FindXxx
# -------
#
# This is a cool module.
# This module does really cool stuff.
# It can do even more than you think.
#
# It even needs two paragraphs to tell you about it.
# And it defines the following variables:
#
# * VAR_COOL: this is great isn't it?
# * VAR_REALLY_COOL: cool right?

<code>

#[========================================[.rst:
.. command:: xxx_do_something

 This command does something for Xxx::

  xxx_do_something(some arguments)
#]========================================]
macro(xxx_do_something)
```

```
  <code>
endmacro()
```

Test the documentation formatting by running `cmake --help-module <module-name>`, and also by enabling the `SPHINX_HTML` and `SPHINX_MAN` options to build the documentation. Edit the comments until generated documentation looks satisfactory. To have a .cmake file in this directory NOT show up in the modules documentation, simply leave out the `Help/module/<module-name>.rst` file and the `Help/manual/cmake-modules.7.rst` toctree entry.

Find Modules

A "find module" is a `Modules/Find<PackageName>.cmake` file to be loaded by the `find_package()` command when invoked for `<PackageName>`.

The primary task of a find module is to determine whether a package exists on the system, set the `<PackageName>_FOUND` variable to reflect this and provide any variables, macros and imported targets required to use the package. A find module is useful in cases where an upstream library does not provide a config file package.

The traditional approach is to use variables for everything, including libraries and executables: see the Standard Variable Names section below. This is what most of the existing find modules provided by CMake do.

The more modern approach is to behave as much like config file packages files as possible, by providing imported target. This has the advantage of propagating Transitive Usage Requirements to consumers.

In either case (or even when providing both variables and imported targets), find modules should provide backwards compatibility with old versions that had the same name.

A FindFoo.cmake module will typically be loaded by the command:

```
find_package(Foo [major[.minor[.patch[.tweak]]]]
             [EXACT] [QUIET] [REQUIRED]
             [[COMPONENTS] [components...]]
             [OPTIONAL_COMPONENTS components...]
             [NO_POLICY_SCOPE])
```

See the `find_package()` documentation for details on what variables are set for the find module. Most of these are dealt with by using `FindPackageHandleStandardArgs`.

Briefly, the module should only locate versions of the package compatible with the requested version, as described by the `Foo_FIND_VERSION` family of variables. If `Foo_FIND_QUIETLY` is set to true, it should avoid printing messages, including anything complaining about the package not being found. If `Foo_FIND_REQUIRED` is set to true, the module should issue a `FATAL_ERROR` if the package cannot be found. If neither are set to true, it should print a non-fatal message if it cannot find the package.

Packages that find multiple semi-independent parts (like bundles of libraries) should search for the components listed in `Foo_FIND_COMPONENTS` if it is set , and only set `Foo_FOUND` to true if for each searched-for component `<c>` that was not found, `Foo_FIND_REQUIRED_<c>` is not set to true. The `HANDLE_COMPONENTS` argument of `find_package_handle_standard_args()` can be used to implement this.

If `Foo_FIND_COMPONENTS` is not set, which modules are searched for and required is up to the find module, but should be documented.

For internal implementation, it is a generally accepted convention that variables starting with underscore are for temporary use only.

Like all modules, find modules should be properly documented. To add a module to the CMake documentation, follow the steps in the Module Documentation section above.

Standard Variable Names

For a `FindXxx.cmake` module that takes the approach of setting variables (either instead of or in addition to creating imported targets), the following variable names should be used to keep things consistent between find modules. Note that all variables start with `Xxx_` to make sure they do not interfere with other find modules; the same consideration applies to macros, functions and imported targets.

- `Xxx_INCLUDE_DIRS`

  The final set of include directories listed in one variable for use by client code. This should not be a cache entry.

- `Xxx_LIBRARIES`

  The libraries to link against to use Xxx. These should include full paths. This should not be a cache entry.

- `Xxx_DEFINITIONS`

  Definitions to use when compiling code that uses Xxx. This really shouldn’t include options such as `-DHAS_JPEG` that a client source-code file uses to decide whether to `#include <jpeg.h>`

- `Xxx_EXECUTABLE`

  Where to find the Xxx tool.

- `Xxx_Yyy_EXECUTABLE`

  Where to find the Yyy tool that comes with Xxx.

- `Xxx_LIBRARY_DIRS`

  Optionally, the final set of library directories listed in one variable for use by client code. This should not be a cache entry.

- `Xxx_ROOT_DIR`

  Where to find the base directory of Xxx.

- `Xxx_VERSION_Yy`

  Expect Version Yy if true. Make sure at most one of these is ever true.

- `Xxx_WRAP_Yy`

  If False, do not try to use the relevant CMake wrapping command.

- `Xxx_Yy_FOUND`

  If False, optional Yy part of Xxx system is not available.

- `Xxx_FOUND`

  Set to false, or undefined, if we haven’t found, or don’t want to use Xxx.

- `Xxx_NOT_FOUND_MESSAGE`

  Should be set by config-files in the case that it has set `Xxx_FOUND` to FALSE. The contained message will be printed by the `find_package()` command and by `find_package_handle_standard_args()` to inform the user about the problem.

- `Xxx_RUNTIME_LIBRARY_DIRS`

  Optionally, the runtime library search path for use when running an executable linked to shared libraries. The list should be used by user code to create the `PATH` on windows or `LD_LIBRARY_PATH` on UNIX. This should not be a cache entry.

- `Xxx_VERSION`

  The full version string of the package found, if any. Note that many existing modules provide `Xxx_VERSION_STRING` instead.

- `Xxx_VERSION_MAJOR`

  The major version of the package found, if any.

- `Xxx_VERSION_MINOR`

  The minor version of the package found, if any.

- `Xxx_VERSION_PATCH`

  The patch version of the package found, if any.

The following names should not usually be used in CMakeLists.txt files, but are typically cache variables for users to edit and control the behaviour of find modules (like entering the path to a library manually)

- `Xxx_LIBRARY`

  The path of the Xxx library (as used with `find_library()`, for example).

- `Xxx_Yy_LIBRARY`

  The path of the Yy library that is part of the Xxx system. It may or may not be required to use Xxx.

- `Xxx_INCLUDE_DIR`

  Where to find headers for using the Xxx library.

- `Xxx_Yy_INCLUDE_DIR`

  Where to find headers for using the Yy library of the Xxx system.

To prevent users being overwhelmed with settings to configure, try to keep as many options as possible out of the cache, leaving at least one option which can be used to disable use of the module, or locate a not-found library (e.g. `Xxx_ROOT_DIR`). For the same reason, mark most cache options as advanced. For packages which provide both debug and release binaries, it is common to create cache variables with a `_LIBRARY_<CONFIG>` suffix, such as `Foo_LIBRARY_RELEASE` and `Foo_LIBRARY_DEBUG`.

While these are the standard variable names, you should provide backwards compatibility for any old names that were actually in use. Make sure you comment them as deprecated, so that no-one starts using them.

A Sample Find Module

We will describe how to create a simple find module for a library `Foo`.

The first thing that is needed is a license notice.

```
# Distributed under the OSI-approved BSD 3-Clause License.  See accompanying
# file Copyright.txt or https://cmake.org/licensing for details.
```

Next we need module documentation. CMake's documentation system requires you to follow the license notice with a blank line and then with a documentation marker and the name of the module. You should follow this with a simple statement of what the module does.

```
#.rst:
# FindFoo
# -------
#
# Finds the Foo library
#
```

More description may be required for some packages. If there are caveats or other details users of the module should be aware of, you can add further paragraphs below this. Then you need to document what variables and imported targets are set by the module, such as

```
# This will define the following variables::
#
#   Foo_FOUND    - True if the system has the Foo library
#   Foo_VERSION  - The version of the Foo library which was found
#
# and the following imported targets::
#
#   Foo::Foo   - The Foo library
```

If the package provides any macros, they should be listed here, but can be documented where they are defined. See the Module Documentation section above for more details.

Now the actual libraries and so on have to be found. The code here will obviously vary from module to module (dealing with that, after all, is the point of find modules), but there tends to be a common pattern for libraries.

First, we try to use `pkg-config` to find the library. Note that we cannot rely on this, as it may not be available, but it provides a good starting point.

```
find_package(PkgConfig)
pkg_check_modules(PC_Foo QUIET Foo)
```

This should define some variables starting `PC_Foo_` that contain the information from the `Foo.pc` file.

Now we need to find the libraries and include files; we use the information from `pkg-config` to provide hints to CMake about where to look.

```
find_path(Foo_INCLUDE_DIR
  NAMES foo.h
  PATHS ${PC_Foo_INCLUDE_DIRS}
  PATH_SUFFIXES Foo
)
find_library(Foo_LIBRARY
  NAMES foo
  PATHS ${PC_Foo_LIBRARY_DIRS}
)
```

If you have a good way of getting the version (from a header file, for example), you can use that information to set `Foo_VERSION` (although note that find modules have traditionally used `Foo_VERSION_STRING`, so you may want to set both). Otherwise, attempt to use the information from `pkg-config`

```
set(Foo_VERSION ${PC_Foo_VERSION})
```

Now we can use `FindPackageHandleStandardArgs` to do most of the rest of the work for us

```
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(Foo
  FOUND_VAR Foo_FOUND
  REQUIRED_VARS
    Foo_LIBRARY
    Foo_INCLUDE_DIR
  VERSION_VAR Foo_VERSION
)
```

This will check that the `REQUIRED_VARS` contain values (that do not end in `-NOTFOUND`) and set `Foo_FOUND` appropriately. It will also cache those values. If `Foo_VERSION` is set, and a required version was passed to `find_package()`, it will check the requested version against the one in `Foo_VERSION`. It will also print messages as appropriate; note that if the package was found, it will print the contents of the first required variable to indicate where it was found.

At this point, we have to provide a way for users of the find module to link to the library or libraries that were found. There are two approaches, as discussed in the Find Modules section above. The traditional variable approach looks like

```
if(Foo_FOUND)
  set(Foo_LIBRARIES ${Foo_LIBRARY})
  set(Foo_INCLUDE_DIRS ${Foo_INCLUDE_DIR})
  set(Foo_DEFINITIONS ${PC_Foo_CFLAGS_OTHER})
endif()
```

If more than one library was found, all of them should be included in these variables (see the Standard Variable Names section for more information).

When providing imported targets, these should be namespaced (hence the `Foo::` prefix); CMake will recognize that values passed to `target_link_libraries()` that contain `::` in their name are supposed to be imported targets (rather than just library names), and will produce appropriate diagnostic messages if that target does not exist (see policy `CMP0028`).

```
if(Foo_FOUND AND NOT TARGET Foo::Foo)
  add_library(Foo::Foo UNKNOWN IMPORTED)
  set_target_properties(Foo::Foo PROPERTIES
    IMPORTED_LOCATION "${Foo_LIBRARY}"
    INTERFACE_COMPILE_OPTIONS "${PC_Foo_CFLAGS_OTHER}"
    INTERFACE_INCLUDE_DIRECTORIES "${Foo_INCLUDE_DIR}"
  )
endif()
```

One thing to note about this is that the `INTERFACE_INCLUDE_DIRECTORIES` and similar properties should only contain information about the target itself, and not any of its dependencies. Instead, those dependencies should also be targets, and CMake should be told that they are dependencies of this target. CMake will then combine all the necessary information automatically.

The type of the `IMPORTED` target created in the `add_library()` command can always be specified as `UNKNOWN` type. This simplifies the code in cases where static or shared variants may be found, and CMake will determine the type by inspecting the files.

If the library is available with multiple configurations, the `IMPORTED_CONFIGURATIONS` target property should also be populated:

```
if(Foo_FOUND)
  if (NOT TARGET Foo::Foo)
    add_library(Foo::Foo UNKNOWN IMPORTED)
  endif()
  if (Foo_LIBRARY_RELEASE)
    set_property(TARGET Foo::Foo APPEND PROPERTY
      IMPORTED_CONFIGURATIONS RELEASE
    )
    set_target_properties(Foo::Foo PROPERTIES
      IMPORTED_LOCATION_RELEASE "${Foo_LIBRARY_RELEASE}"
    )
  endif()
  if (Foo_LIBRARY_DEBUG)
    set_property(TARGET Foo::Foo APPEND PROPERTY
      IMPORTED_CONFIGURATIONS DEBUG
    )
    set_target_properties(Foo::Foo PROPERTIES
      IMPORTED_LOCATION_DEBUG "${Foo_LIBRARY_DEBUG}"
    )
  endif()
  set_target_properties(Foo::Foo PROPERTIES
    INTERFACE_COMPILE_OPTIONS "${PC_Foo_CFLAGS_OTHER}"
    INTERFACE_INCLUDE_DIRECTORIES "${Foo_INCLUDE_DIR}"
  )
endif()
```

The `RELEASE` variant should be listed first in the property so that the variant is chosen if the user uses a configuration which is not an exact match for any listed `IMPORTED_CONFIGURATIONS`.

Most of the cache variables should be hidden in the `ccmake` interface unless the user explicitly asks to edit them.

```
mark_as_advanced(
  Foo_INCLUDE_DIR
  Foo_LIBRARY
)
```

If this module replaces an older version, you should set compatibility variables to cause the least disruption possible.

```
# compatibility variables
set(Foo_VERSION_STRING ${Foo_VERSION})
```