NAME

cmake-developer − CMake Developer Reference

INTRODUCTION

This manual is intended for reference by developers working with **cmake−language(7)** code, whether writing their own modules, authoring their own build systems, or working on CMake itself.

See *https://cmake.org/get−involved/* to get involved in development of CMake upstream. It includes links to contribution instructions, which in turn link to developer guides for CMake itself.

FIND MODULES

A "find module" is a **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 is available, 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 Target 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**F oo_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.

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_**, which (unless otherwise noted) must match exactly the name of the **FindXxx.cmake** file, including upper/lowercase. This prefix on the variable names ensures that they do not conflict with variables of other Find modules. The same pattern should also be followed for any macros, functions and imported targets defined by the Find module.

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 (note that this also means this variable should not be used as the result variable of a **find_path()** command − see **Xxx_INCLUDE_DIR** below for that).

Xxx_LIBRARIES
The libraries to use with the module. These may be CMake targets, full absolute paths to a library binary or the name of a library that the linker must find in its search path. This should not be a cache entry (note that this also means this variable should not be used as the result variable of a **find_library()** command − see **Xxx_LIBRARY** below for that).

Xxx_DEFINITIONS
The compile definitions to use when compiling code that uses the module. 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
The full absolute path to an executable. In this case,**Xxx** might not be the name of the module, it might be the name of the tool (usually converted to all uppercase), assuming that tool has such a well−known name that it is unlikely that another tool with the same name exists. It would be appropriate to use this as the result variable of a **find_program()** command.

Xxx_YYY_EXECUTABLE
Similar to **Xxx_EXECUTABLE** except here the **Xxx** is always the module name and **YYY** is the tool name (again, usually fully uppercase). Prefer this form if the tool name is not very widely known or has the potential to clash with another tool. For greater consistency, also prefer this form if the module provides more than one executable.

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 the module.

Xxx_VERSION_VV
Variables of this form specify whether the **Xxx** module being provided is version **VV** of the module. There should not be more than one variable of this form set to true for a given module. For example, a module **Barry** might have evolved over many years and gone through a number of different major versions. Version 3 of the **Barry** module might set the variable **Barry_VERSION_3** to true, whereas an older version of the module might set **Barry_VERSION_2** to true instead. It would be an error for both **Barry_VERSION_3** and **Barry_VERSION_2** to both be set to true.

Xxx_WRAP_YY
When a variable of this form is set to false, it indicates that the relevant wrapping command should not be used. The wrapping command depends on the module, it may be implied by the module name or it might be specified by the **YY** part of the variable.

Xxx_Yy_FOUND
For variables of this form, **Yy** is the name of a component for the module. It should match exactly one of the valid component names that may be passed to the **find_package()** command for the

module. If a variable of this form is set to false, it means that the **Yy** component of module **Xxx** was not found or is not available. Variables of this form would typically be used for optional components so that the caller can check whether an optional component is available.

Xxx_FOUND
When the **find_package()** command returns to the caller, this variable will be set to true if the module was deemed to have been found successfully.

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. Use this instead of calling **message()** directly to report a reason for failing to find the module or package.

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**P ATH** 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. They are intended for use by Find modules to specify and cache the locations of specific files or directories. Users are typically able to set and edit these variables to control the behavior of Find modules (like entering the path to a library manually):

Xxx_LIBRARY
The path of the library. Use this form only when the module provides a single library. It is appropriate to use this as the result variable in a **find_library()** command.

Xxx_Yy_LIBRARY
The path of library **Yy** provided by the module **Xxx**. Use this form when the module provides more than one library or where other modules may also provide a library of the same name. It is also appropriate to use this form as the result variable in a **find_library()** command.

Xxx_INCLUDE_DIR
When the module provides only a single library, this variable can be used to specify where to find headers for using the library (or more accurately, the path that consumers of the library should add to their header search path). It would be appropriate to use this as the result variable in a **find_path()** command.

Xxx_Yy_INCLUDE_DIR
If the module provides more than one library or where other modules may also provide a library of the same name, this form is recommended for specifying where to find headers for using library **Yy** provided by the module. Again, it would be appropriate to use this as the result variable in a **find_path()** command.

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**. The **SelectLibraryConfigurations** module can be helpful for such cases.

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 top of the module should begin with a license notice, followed by a blank line, and then followed by a Bracket Comment. The comment should begin with **.rst:** to indicate that the rest of its content is reStructuredText−format documentation. For example:

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

#[=======================================================================[.rst
FindFoo
−−−−−−−

Finds the Foo library.

Imported Targets
^^^^^^^^^^^^^^^^

This module provides the following imported targets, if found:

``Foo::Foo``
  The Foo library

Result Variables
^^^^^^^^^^^^^^^^

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.
``Foo_INCLUDE_DIRS``
  Include directories needed to use Foo.
``Foo_LIBRARIES``
  Libraries needed to link to Foo.

Cache Variables
^^^^^^^^^^^^^^^

The following cache variables may also be set:

``Foo_INCLUDE_DIR``
  The directory containing ``foo.h``.
``Foo_LIBRARY``
  The path to the Foo library.
```

```
#]=======================================================================]
```

The module documentation consists of:

- An underlined heading specifying the module name.
- A simple description of what the module finds. More description may be required for some packages. If there are caveats or other details users of the module should be aware of, specify them here.
- A section listing imported targets provided by the module, if any.
- A section listing result variables provided by the module.
- Optionally a section listing cache variables used by the module, if any.

If the package provides any macros or functions, they should be listed in an additional section, but can be documented by additional **.rst:** comment blocks immediately above where those macros or functions are defined.

The find module implementation may begin below the documentation block. 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}
)
```

Alternatively, if the library is available with multiple configurations, you can use **SelectLibraryConfigurations** to automatically set the **Foo_LIBRARY** variable instead:

```
find_library(Foo_LIBRARY_RELEASE
  NAMES foo
  PATHS ${PC_Foo_LIBRARY_DIRS}/Release
)
find_library(Foo_LIBRARY_DEBUG
  NAMES foo
  PATHS ${PC_Foo_LIBRARY_DIRS}/Debug
)

include(SelectLibraryConfigurations)
```

```
select_library_configurations(Foo)
```

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**F oo_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})
```