NAME

cmake-developer - CMake Developer Reference

INTRODUCTION

This manual is intended for reference by developers modifying the CMake source tree itself.

PERMITTED C++ SUBSET

CMake is required to build with ancient C++ compilers and standard library implementations. Some common C++ constructs may not be used in CMake in order to build with such toolchains.

std::vector::at

The **at()** member function of **std::vector** may not be used. Use **operator[]** instead:

```
std::vector<int> someVec = getVec();
int i1 = someVec.at(5); // Wrong
int i2 = someVec[5]; // Ok
```

std::string::append and std::string::clear

The **append()** and **clear()** member functions of **std::string** may not be used. Use **operator+=** and **operator=** instead:

```
std::string stringBuilder;
stringBuilder.append("chunk"); // Wrong
stringBuilder.clear(); // Wrong
stringBuilder += "chunk"; // Ok
stringBuilder = ""; // Ok
```

std::set const iterators

The **find()** member function of a **const std::set** instance may not be used in a comparison with the iterator returned by **end()**:

```
const std::set<cmStdString>& someSet = getSet();
if (someSet.find("needle") == someSet.end()) // Wrong
{
// ...
}
```

The return value of **find()** must be assigned to an intermediate **const_iterator** for comparison:

```
const std::set<cmStdString>& someSet;
const std::set<cmStdString>::const_iterator i = someSet.find("needle");
if (i != propSet.end()) // Ok
{
// ...
}
```

Char Array to string Conversions with Algorithms

In some implementations, algorithms operating on iterators to a container of **std::string** can not accept a **const char*** value:

```
const char* dir = /*...*/;
std::vector<std::string> vec;
// ...
std::binary_search(vec.begin(), vec.end(), dir); // Wrong
```

The **std::string** may need to be explicitly constructed:

```
const char* dir = /*...*/;
std::vector<std::string> vec;
// ...
std::binary_search(vec.begin(), vec.end(), std::string(dir)); // Ok
```

std::auto_ptr
Some implementations have a **std::auto_ptr** which can not be used as a return value from a function. **std::auto_ptr** may not be used. Use **cmsys::auto_ptr** instead.

std::vector::insert and std::set
Use of **std::vector::insert** with an iterator whose **element_type** requires conversion is not allowed:

```
std::set<cmStdString> theSet;
std::vector<std::string> theVector;
theVector.insert(theVector.end(), theSet.begin(), theSet.end()); // Wrong
```

A loop must be used instead:

```
std::set<cmStdString> theSet;
std::vector<std::string> theVector;
for(std::set<cmStdString>::iterator li = theSet.begin();
li != theSet.end(); ++li)
{
theVector.push_back(*li);
}
```

std::set::insert
Use of **std::set::insert** is not allowed with any source container:

```
std::set<cmTarget*> theSet;
theSet.insert(targets.begin(), targets.end()); // Wrong
```

A loop must be used instead:

```
ConstIterator it = targets.begin();
const ConstIterator end = targets.end();
for ( ; it != end; ++it)
{
theSet.insert(*it);
}
```

Template Parameter Defaults
On ancient compilers, C++ template must use template parameters in function arguments. If no parameter of that type is needed, the common workaround is to add a defaulted pointer to the type to the templated function. However, this does not work with other ancient compilers:

```
template<typename PropertyType>
PropertyType getTypedProperty(cmTarget* tgt, const char* prop,
PropertyType* = 0) // Wrong
{

}

template<typename PropertyType>
PropertyType getTypedProperty(cmTarget* tgt, const char* prop,
PropertyType*) // Ok
{

}
```

and invoke it with the value **0** explicitly in all cases.

std::min and std::max
min and **max** are defined as macros on some systems. **std::min** and **std::max** may not be used. Use **cmMinimum** and **cmMaximum** instead.

size_t
Various implementations have differing implementation of **size_t**. When assigning the result of **.size()** on a container for example, the result should not be assigned to an **unsigned int** or similar. **std::size_t** must not be used.

Templates
Some template code is permitted, but with some limitations. Member templates may not be used, and template friends may not be used.

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 tools **-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_test, prop_tgt
A CMake cache, directory, global, source 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 Domain* provides 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_<CONFIG>** 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

1. Command signatures should be marked up as plain literal blocks, not as cmake **code-blocks**.
2. Signatures are separated from preceding content by a horizontal line. That is, use:

```
... preceding paragraph.

------------------------------------------------------------------------
```

```
::

add_library(<lib> ...)

This signature is used for ...
```

3. 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 cant be modified after being set, such as the **IMPORTED** property of a build target.
4. Use two spaces for indentation. Use two spaces between sentences in prose.
5. 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.
6. 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.
7. Mark up 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.
8. Mark up all other 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.
9. Mark up references to keywords in signatures, file names, and other technical terms with **inline-literl** 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.
   ```

10. 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*).

11. 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.

12. 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.

13. When referring to 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 ...
    ```

14. 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.

15. Use American English spellings in prose.

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:**. Add to the top of**Modules/<module-name>.cmak e** a *Line Comment*bloc k 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:

```
#.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()
```

Find Modules

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

We would like all **FindXxx.cmake** files to produce consistent variable names. Please use the following consistent variable names for general use.

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 shouldnt 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 sytem is not available.

Xxx_FOUND
Set to false, or undefined, if we havent found, or dont 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_STRING
A human-readable string containing the version of the package found, if any.

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.

You do not have to provide all of the above variables. You should provide Xxx_FOUND under most circumstances. If Xxx is a library, then Xxx_LIBRARIES, should also be defined, and Xxx_INCLUDE_DIRS should usually be defined (I guess libm.a might be an exception)

The following names should not usually be used in CMakeLists.txt files, but they may be usefully modified in users CMake Caches to control stuff.

Xxx_LIBRARY
Name of Xxx Library. A User may set this and Xxx_INCLUDE_DIR to ignore to force non-use of Xxx.

Xxx_Yy_LIBRARY
Name of 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 xxx.h, etc. (Xxx_INCLUDE_PATH was considered bad because a path includes an actual filename.)

Xxx_Yy_INCLUDE_DIR
Where to find xxx_yy.h, etc.

For tidinesss sake, 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.

If you need other commands to do special things then it should still begin with **Xxx_**. This gives a sort of namespace effect and keeps things tidy for the user. You should put comments describing all the exported settings, plus descriptions of any the users can use to control stuff.

You really should also provide backwards compatibility any old settings that were actually in use. Make sure you comment them as deprecated, so that no-one starts using them.

To add a module to the CMake documentation, follow the steps in the *Module Documentation* section above. Test the documentation formatting by running **cmake --help-module FindXxx**, 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.

After the documentation, leave a *BLANK* line, and then add a copyright and licence notice block like this one:

```
#=============================================================================
# Copyright 2009-2011 Your Name
#
# Distributed under the OSI-approved BSD License (the "License");
# see accompanying file Copyright.txt for details.
#
# This software is distributed WITHOUT ANY WARRANTY; without even the
# implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
# See the License for more information.
#=============================================================================
# (To distribute this file outside of CMake, substitute the full
# License text for the above reference.)
```

The layout of the notice block is strictly enforced by the **ModuleNotices** test. Only the year range and name may be changed freely.

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

```
FIND_PACKAGE(Xxx [major[.minor[.patch[.tweak]]]] [EXACT]
[QUIET] [[REQUIRED|COMPONENTS] [components...]])
```

If any version numbers are given to the command it will set the following variables before loading the module:

Xxx_FIND_VERSION
: full requested version string

Xxx_FIND_VERSION_MAJOR
: major version if requested, else 0

Xxx_FIND_VERSION_MINOR
: minor version if requested, else 0

Xxx_FIND_VERSION_PATCH
: patch version if requested, else 0

Xxx_FIND_VERSION_TWEAK
: tweak version if requested, else 0

Xxx_FIND_VERSION_COUNT
: number of version components, 0 to 4

Xxx_FIND_VERSION_EXACT
true if EXACT option was given

If the find module supports versioning it should locate a version of the package that is compatible with the version requested. If a compatible version of the package cannot be found the module should not report success. The version of the package found should be stored in Xxx_VERSION... version variables documented by the module.

If the QUIET option is given to the command it will set the variable Xxx_FIND_QUIETLY to true before loading the FindXxx.cmake module. If this variable is set the module should not complain about not being able to find the package. If the REQUIRED option is given to the command it will set the variable Xxx_FIND_REQUIRED to true before loading the FindXxx.cmake module. If this variable is set the module should issue a FATAL_ERROR if the package cannot be found. If neither the QUIET nor REQUIRED options are given then the FindXxx.cmake module should look for the package and complain without error if the module is not found.

FIND_PACKAGE() will set the variable CMAKE_FIND_PACKAGE_NAME to contain the actual name of the package.

A package can provide sub-components. Those components can be listed after the COMPONENTS (or REQUIRED) or OPTIONAL_COMPONENTS keywords. The set of all listed components will be specified in a Xxx_FIND_COMPONENTS variable. For each package-specific component, say Yyy, a variable Xxx_FIND_REQUIRED_Yyy will be set to true if it listed after COMPONENTS and it will be set to false if it was listed after OPTIONAL_COMPONENTS. Using those variables a FindXxx.cmake module and also a XxxConfig.cmake package configuration file can determine whether and which components have been requested, and whether they were requested as required or as optional. For each of the requested components a Xxx_Yyy_FOUND variable should be set accordingly. The per-package Xxx_FOUND variable should be only set to true if all requested required components have been found. A missing optional component should not keep the Xxx_FOUND variable from being set to true. If the package provides Xxx_INCLUDE_DIRS and Xxx_LIBRARIES variables, the include dirs and libraries for all components which were requested and which have been found should be added to those two variables.

To get this behavior you can use the FIND_PACKAGE_HANDLE_STANDARD_ARGS() macro, as an example see FindJPEG.cmake.

For internal implementation, its a generally accepted convention that variables starting with underscore are for temporary use only. (variable starting with an underscore are not intended as a reserved prefix).

COPYRIGHT