NAME

cmake-packages − CMake Packages Reference

INTRODUCTION

Packages provide dependency information to CMake based buildsystems. Packages are found with the **find_package()** command. The result of using **find_package()** is either a set of **IMPORTED** targets, or a set of variables corresponding to build−relevant information.

USING PACKAGES

CMake provides direct support for two forms of packages, *Config−file Packages* and *Find−module Packages*. Indirect support for**pkg−config** packages is also pro vided via the **FindPkgConfig** module. In all cases, the basic form of **find_package()** calls is the same:

```
find_package(Qt4 4.7.0 REQUIRED) # CMake provides a Qt4 find−module
find_package(Qt5Core 5.1.0 REQUIRED) # Qt provides a Qt5 package config file.
find_package(LibXml2 REQUIRED) # Use pkg−config via the LibXml2 find−module
```

In cases where it is known that a package configuration file is provided by upstream, and only that should be used, the **CONFIG** keyword may be passed to **find_package()**:

```
find_package(Qt5Core 5.1.0 CONFIG REQUIRED)
find_package(Qt5Gui 5.1.0 CONFIG)
```

Similarly, the **MODULE** keyword says to use only a find−module:

```
find_package(Qt4 4.7.0 MODULE REQUIRED)
```

Specifying the type of package explicitly improves the error message shown to the user if it is not found.

Both types of packages also support specifying components of a package, either after the **REQUIRED** keyword:

```
find_package(Qt5 5.1.0 CONFIG REQUIRED Widgets Xml Sql)
```

or as a separate **COMPONENTS** list:

```
find_package(Qt5 5.1.0 COMPONENTS Widgets Xml Sql)
```

or as a separate **OPTIONAL_COMPONENTS** list:

```
find_package(Qt5 5.1.0 COMPONENTS Widgets
                       OPTIONAL_COMPONENTS Xml Sql
)
```

Handling of **COMPONENTS** and **OPTIONAL_COMPONENTS** is defined by the package.

By setting the **CMAKE_DISABLE_FIND_PACKAGE_<PackageName>** variable to **TRUE**, the **<PackageName>** package will not be searched, and will always be **NOTFOUND**. Likewise, setting the **CMAKE_REQUIRE_FIND_PACKAGE_<PackageName>** to **TRUE** will make the package REQUIRED.

Config−file Packages

A config−file package is a set of files provided by upstreams for downstreams to use. CMake searches in a number of locations for package configuration files, as described in the **find_package()** documentation. The most simple way for a CMake user to tell **cmake(1)** to search in a non−standard prefix for a package is to set the **CMAKE_PREFIX_PATH** cache variable.

Config−file packages are provided by upstream vendors as part of development packages, that is, they belong with the header files and any other files provided to assist downstreams in using the package.

A set of variables which provide package status information are also set automatically when using a config−file package. The **<PackageName>_FOUND** variable is set to true or false, depending on whether the package was found. The **<PackageName>_DIR** cache variable is set to the location of the package configuration file.

Find−module Packages

A find module is a file with a set of rules for finding the required pieces of a dependency, primarily header files and libraries. Typically, a find module is needed when the upstream is not built with CMake, or is not CMake−aware enough to otherwise provide a package configuration file. Unlike a package configuration file, it is not shipped with upstream, but is used by downstream to find the files by guessing locations of files with platform−specific hints.

Unlike the case of an upstream−provided package configuration file, no single point of reference identifies the package as being found, so the **<PackageName>_FOUND** variable is not automatically set by the **find_package()** command. It can still be expected to be set by convention however and should be set by the author of the Find−module. Similarly there is no **<PackageName>_DIR** variable, but each of the artifacts such as library locations and header file locations provide a separate cache variable.

See the **cmake−developer(7)** manual for more information about creating Find−module files.

PACKAGE LAYOUT

A config−file package consists of a *Package Configuration File* and optionally a *Package Version File* provided with the project distribution.

Package Configuration File

Consider a project **Foo** that installs the following files:

```
<prefix>/include/foo−1.2/foo.h
<prefix>/lib/foo−1.2/libfoo.a
```

It may also provide a CMake package configuration file:

```
<prefix>/lib/cmake/foo−1.2/FooConfig.cmake
```

with content defining **IMPORTED** targets, or defining variables, such as:

```
# ...
# (compute PREFIX relative to file location)
# ...
set(Foo_INCLUDE_DIRS ${PREFIX}/include/foo−1.2)
set(Foo_LIBRARIES ${PREFIX}/lib/foo−1.2/libfoo.a)
```

If another project wishes to use **Foo** it need only to locate the **FooConfig.cmake** file and load it to get all the information it needs about package content locations. Since the package configuration file is provided by the package installation it already knows all the file locations.

The **find_package()** command may be used to search for the package configuration file. This command constructs a set of installation prefixes and searches under each prefix in several locations. Given the name **Foo**, it looks for a file called **FooConfig.cmake** or **foo−config.cmake**. The full set of locations is specified in the **find_package()** command documentation. One place it looks is:

```
<prefix>/lib/cmake/Foo*/
```

where **Foo*** is a case−insensitive globbing expression. In our example the globbing expression will match **<prefix>/lib/cmake/foo−1.2** and the package configuration file will be found.

Once found, a package configuration file is immediately loaded. It, together with a package version file, contains all the information the project needs to use the package.

Package Version File

When the **find_package()** command finds a candidate package configuration file it looks next to it for a version file. The version file is loaded to test whether the package version is an acceptable match for the version requested. If the version file claims compatibility the configuration file is accepted. Otherwise it is ignored.

The name of the package version file must match that of the package configuration file but has either **−version** or **Version** appended to the name before the **.cmake** extension. For example, the files:

```
<prefix>/lib/cmake/foo−1.3/foo−config.cmake
<prefix>/lib/cmake/foo−1.3/foo−config−version.cmake
```

and:

```
<prefix>/lib/cmake/bar−4.2/BarConfig.cmake
<prefix>/lib/cmake/bar−4.2/BarConfigVersion.cmake
```

are each pairs of package configuration files and corresponding package version files.

When the **find_package()** command loads a version file it first sets the following variables:

PACKAGE_FIND_NAME
The **<PackageName>**

PACKAGE_FIND_VERSION
Full requested version string

PACKAGE_FIND_VERSION_MAJOR
Major version if requested, else 0

PACKAGE_FIND_VERSION_MINOR
Minor version if requested, else 0

PACKAGE_FIND_VERSION_PATCH
Patch version if requested, else 0

PACKAGE_FIND_VERSION_TWEAK
Tweak version if requested, else 0

PACKAGE_FIND_VERSION_COUNT
Number of version components, 0 to 4

The version file must use these variables to check whether it is compatible or an exact match for the requested version and set the following variables with results:

PACKAGE_VERSION
Full provided version string

PACKAGE_VERSION_EXACT
True if version is exact match

PACKAGE_VERSION_COMPATIBLE
True if version is compatible

PACKAGE_VERSION_UNSUITABLE
True if unsuitable as any version

Version files are loaded in a nested scope so they are free to set any variables they wish as part of their computation. The find_package command wipes out the scope when the version file has completed and it has checked the output variables. When the version file claims to be an acceptable match for the requested version the find_package command sets the following variables for use by the project:

<PackageName>_VERSION
Full provided version string

<PackageName>_VERSION_MAJOR
Major version if provided, else 0

<PackageName>_VERSION_MINOR
Minor version if provided, else 0

<PackageName>_VERSION_PATCH
Patch version if provided, else 0

<PackageName>_VERSION_TWEAK
Tweak version if provided, else 0

<PackageName>_VERSION_COUNT
Number of version components, 0 to 4

The variables report the version of the package that was actually found. The **<PackageName>** part of their name matches the argument given to the **find_package()** command.

CREATING PACKAGES

Usually, the upstream depends on CMake itself and can use some CMake facilities for creating the package files. Consider an upstream which provides a single shared library:

```
project(UpstreamLib)

set(CMAKE_INCLUDE_CURRENT_DIR ON)
set(CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE ON)

set(Upstream_VERSION 3.4.1)

include(GenerateExportHeader)

add_library(ClimbingStats SHARED climbingstats.cpp)
generate_export_header(ClimbingStats)
set_property(TARGET ClimbingStats PROPERTY VERSION ${Upstream_VERSION})
set_property(TARGET ClimbingStats PROPERTY SOVERSION 3)
set_property(TARGET ClimbingStats PROPERTY
  INTERFACE_ClimbingStats_MAJOR_VERSION 3)
set_property(TARGET ClimbingStats APPEND PROPERTY
  COMPATIBLE_INTERFACE_STRING ClimbingStats_MAJOR_VERSION
)

install(TARGETS ClimbingStats EXPORT ClimbingStatsTargets
  LIBRARY DESTINATION lib
  ARCHIVE DESTINATION lib
  RUNTIME DESTINATION bin
  INCLUDES DESTINATION include
)
install(
  FILES
    climbingstats.h
    "${CMAKE_CURRENT_BINARY_DIR}/climbingstats_export.h"
  DESTINATION
    include
  COMPONENT
    Devel
)

include(CMakePackageConfigHelpers)
write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/ClimbingStats/ClimbingStatsConfigVersion.cmake"
  VERSION ${Upstream_VERSION}
  COMPATIBILITY AnyNewerVersion
)

export(EXPORT ClimbingStatsTargets
  FILE "${CMAKE_CURRENT_BINARY_DIR}/ClimbingStats/ClimbingStatsTargets.cmake"
  NAMESPACE Upstream::
)
configure_file(cmake/ClimbingStatsConfig.cmake
  "${CMAKE_CURRENT_BINARY_DIR}/ClimbingStats/ClimbingStatsConfig.cmake"
  COPYONLY
)

set(ConfigPackageLocation lib/cmake/ClimbingStats)
install(EXPORT ClimbingStatsTargets
  FILE
    ClimbingStatsTargets.cmake
  NAMESPACE
    Upstream::
  DESTINATION
    ${ConfigPackageLocation}
)
install(
  FILES
    cmake/ClimbingStatsConfig.cmake
    "${CMAKE_CURRENT_BINARY_DIR}/ClimbingStats/ClimbingStatsConfigVersion.cmak
  DESTINATION
    ${ConfigPackageLocation}
  COMPONENT
    Devel
)
```

The **CMakePackageConfigHelpers** module provides a macro for creating a simple **ConfigVersion.cmake** file. This file sets the version of the package. It is read by CMake when **find_package()** is called to determine the compatibility with the requested version, and to set some version−specific variables **<PackageName>_VERSION**, **<PackageName>_VERSION_MAJOR**, **<PackageName>_VERSION_MINOR** etc. The**install(EXPOR T)** command is used to export the targets in the **ClimbingStatsTargets** export−set, defined previously by the **install(TARGETS)** command. This command generates the **ClimbingStatsTargets.cmake** file to contain **IMPORTED** targets, suitable for use by downstreams and arranges to install it to **lib/cmake/ClimbingStats**. The generated**ClimbingStatsConfigV ersion.cmake** and a **cmake/ClimbingStatsConfig.cmake** are installed to the same location, completing the package.

The generated **IMPORTED** targets have appropriate properties set to define their usage requirements, such as **INTERFACE_INCLUDE_DIRECTORIES**, **INTERFACE_COMPILE_DEFINITIONS** and other relevant built−in **INTERFACE_** properties. The **INTERFACE** variant of user−defined properties listed in **COMPATIBLE_INTERFACE_STRING** and other Compatible Interface Properties are also propagated to the generated **IMPORTED** targets. In the above case, **ClimbingStats_MAJOR_VERSION** is defined as a string which must be compatible among the dependencies of any depender. By setting this custom defined user property in this version and in the next version of **ClimbingStats**, **cmake(1)** will issue a diagnostic if there is an attempt to use version 3 together with version 4. Packages can choose to employ such a pattern if different major versions of the package are designed to be incompatible.

A **NAMESPACE** with double−colons is specified when exporting the targets for installation. This convention of double−colons gives CMake a hint that the name is an **IMPORTED** target when it is used by downstreams with the **target_link_libraries()** command. This way, CMake can issue a diagnostic if the package providing it has not yet been found.

In this case, when using **install(TARGETS)** the **INCLUDES DESTINATION** was specified. This causes the **IMPORTED** targets to have their **INTERFACE_INCLUDE_DIRECTORIES** populated with the **include** directory in the **CMAKE_INSTALL_PREFIX**. When the**IMPOR TED** target is used by downstream, it automatically consumes the entries from that property.

Creating a Package Configuration File

In this case, the **ClimbingStatsConfig.cmake** file could be as simple as:

```
include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStatsTargets.cmake")
```

As this allows downstreams to use the **IMPORTED** targets. If any macros should be provided by the **ClimbingStats** package, they should be in a separate file which is installed to the same location as the **ClimbingStatsConfig.cmake** file, and included from there.

This can also be extended to cover dependencies:

```
# ...
add_library(ClimbingStats SHARED climbingstats.cpp)
generate_export_header(ClimbingStats)

find_package(Stats 2.6.4 REQUIRED)
target_link_libraries(ClimbingStats PUBLIC Stats::Types)
```

As the **Stats::Types** target is a **PUBLIC** dependency of **ClimbingStats**, downstreams must also find the **Stats** package and link to the **Stats::Types** library. The**Stats** package should be found in the **ClimbingStatsConfig.cmake** file to ensure this. The **find_dependency** macro from the **CMakeFindDependencyMacro** helps with this by propagating whether the package is **REQUIRED**, or **QUIET** etc. All **REQUIRED** dependencies of a package should be found in the **Config.cmake** file:

```
include(CMakeFindDependencyMacro)
find_dependency(Stats 2.6.4)

include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStatsTargets.cmake")
include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStatsMacros.cmake")
```

The **find_dependency** macro also sets **ClimbingStats_FOUND** to **False** if the dependency is not found, along with a diagnostic that the **ClimbingStats** package can not be used without the **Stats** package.

If **COMPONENTS** are specified when the downstream uses **find_package()**, they are listed in the **<PackageName>_FIND_COMPONENTS** variable. If a particular component is non−optional, then the

<PackageName>_FIND_REQUIRED_<comp> will be true. This can be tested with logic in the package configuration file:

```
include(CMakeFindDependencyMacro)
find_dependency(Stats 2.6.4)

include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStatsTargets.cmake")
include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStatsMacros.cmake")

set(_supported_components Plot Table)

foreach(_comp ${ClimbingStats_FIND_COMPONENTS})
  if (NOT ";${_supported_components};" MATCHES ";${_comp};")
    set(ClimbingStats_FOUND False)
    set(ClimbingStats_NOT_FOUND_MESSAGE "Unsupported component: ${_comp}")
  endif()
  include("${CMAKE_CURRENT_LIST_DIR}/ClimbingStats${_comp}Targets.cmake")
endforeach()
```

Here, the **ClimbingStats_NOT_FOUND_MESSAGE** is set to a diagnosis that the package could not be found because an invalid component was specified. This message variable can be set for any case where the **_FOUND** variable is set to **False**, and will be displayed to the user.

Creating a Package Configuration File for the Build Tree

The **export(EXPORT)** command creates an **IMPORTED** targets definition file which is specific to the build−tree, and is not relocatable. This can similarly be used with a suitable package configuration file and package version file to define a package for the build tree which may be used without installation. Consumers of the build tree can simply ensure that the **CMAKE_PREFIX_PATH** contains the build directory, or set the **ClimbingStats_DIR** to **<build_dir>/ClimbingStats** in the cache.

Creating Relocatable Packages

A relocatable package must not reference absolute paths of files on the machine where the package is built that will not exist on the machines where the package may be installed.

Packages created by **install(EXPORT)** are designed to be relocatable, using paths relative to the location of the package itself. When defining the interface of a target for **EXPORT**, keep in mind that the include directories should be specified as relative paths which are relative to the **CMAKE_INSTALL_PREFIX**:

```
target_include_directories(tgt INTERFACE
  # Wrong, not relocatable:
  $<INSTALL_INTERFACE:${CMAKE_INSTALL_PREFIX}/include/TgtName>
)

target_include_directories(tgt INTERFACE
  # Ok, relocatable:
  $<INSTALL_INTERFACE:include/TgtName>
)
```

The **$<INSTALL_PREFIX> generator expression** may be used as a placeholder for the install prefix without resulting in a non−relocatable package. This is necessary if complex generator expressions are used:

```
target_include_directories(tgt INTERFACE
  # Ok, relocatable:
  $<INSTALL_INTERFACE:$<$<CONFIG:Debug>:$<INSTALL_PREFIX>/include/TgtName>>
)
```

This also applies to paths referencing external dependencies. It is not advisable to populate any properties which may contain paths, such as **INTERFACE_INCLUDE_DIRECTORIES** and **INTERFACE_LINK_LIBRARIES**, with paths relevant to dependencies. For example, this code may not work well for a relocatable package:

```
target_link_libraries(ClimbingStats INTERFACE
  ${Foo_LIBRARIES} ${Bar_LIBRARIES}
  )
target_include_directories(ClimbingStats INTERFACE
  "$<INSTALL_INTERFACE:${Foo_INCLUDE_DIRS};${Bar_INCLUDE_DIRS}>"
  )
```

The referenced variables may contain the absolute paths to libraries and include directories **as found on the machine the package was made on**. This would create a package with hard−coded paths to dependencies and not suitable for relocation.

Ideally such dependencies should be used through their own IMPORTED targets that have their own **IMPORTED_LOCATION** and usage requirement properties such as **INTERFACE_INCLUDE_DIRECTORIES** populated appropriately. Those imported targets may then be used with the **target_link_libraries()** command for **ClimbingStats**:

```
target_link_libraries(ClimbingStats INTERFACE Foo::Foo Bar::Bar)
```

With this approach the package references its external dependencies only through the names of IMPORTED targets. When a consumer uses the installed package, the consumer will run the appropriate **find_package()** commands (via the **find_dependency** macro described above) to find the dependencies and populate the imported targets with appropriate paths on their own machine.

Unfortunately many **modules** shipped with CMake do not yet provide IMPORTED targets because their development pre−dated this approach. This may improve incrementally over time. Workarounds to create relocatable packages using such modules include:

- When building the package, specify each **Foo_LIBRARY** cache entry as just a library name, e.g. **−DFoo_LIBRARY=foo**. This tells the corresponding find module to populate the**F oo_LIBRARIES** with just **foo** to ask the linker to search for the library instead of hard−coding a path.
- Or, after installing the package content but before creating the package installation binary for redistribution, manually replace the absolute paths with placeholders for substitution by the installation tool when the package is installed.

PACKAGE REGISTRY

CMake provides two central locations to register packages that have been built or installed anywhere on a system:

- *User Package Registry*
- *System Package Registry*

The registries are especially useful to help projects find packages in non−standard install locations or directly in their own build trees. A project may populate either the user or system registry (using its own means, see below) to refer to its location. In either case the package should store at the registered location a *Package Configuration File* (**<PackageName>Config.cmake**) and optionally a *Package Version File* (**<PackageName>ConfigVersion.cmake**).

The **find_package()** command searches the two package registries as two of the search steps specified in its documentation. If it has sufficient permissions it also removes stale package registry entries that refer to directories that do not exist or do not contain a matching package configuration file.

User Package Registry

The User Package Registry is stored in a per−user location. The **export(PACKAGE)** command may be used to register a project build tree in the user package registry. CMake currently provides no interface to add install trees to the user package registry. Installers must be manually taught to register their packages if desired.

On Windows the user package registry is stored in the Windows registry under a key in **HKEY_CURRENT_USER**.

A **<PackageName>** may appear under registry key:

```
HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\<PackageName>
```

as a **REG_SZ** value, with arbitrary name, that specifies the directory containing the package configuration file.

On UNIX platforms the user package registry is stored in the user home directory under **˜/.cmake/packages**. A**<P ackageName>** may appear under the directory:

```
˜/.cmake/packages/<PackageName>
```

as a file, with arbitrary name, whose content specifies the directory containing the package configuration file.

System Package Registry

The System Package Registry is stored in a system−wide location. CMake currently provides no interface to add to the system package registry. Installers must be manually taught to register their packages if desired.

On Windows the system package registry is stored in the Windows registry under a key in **HKEY_LOCAL_MACHINE**. A**<P ackageName>** may appear under registry key:

```
HKEY_LOCAL_MACHINE\Software\Kitware\CMake\Packages\<PackageName>
```

as a **REG_SZ** value, with arbitrary name, that specifies the directory containing the package configuration file.

There is no system package registry on non−Windows platforms.

Disabling the Package Registry

In some cases using the Package Registries is not desirable. CMake allows one to disable them using the following variables:

- The **export(PACKAGE)** command does not populate the user package registry when **CMP0090** is set to **NEW** unless the **CMAKE_EXPORT_PACKAGE_REGISTRY** variable explicitly enables it. When **CMP0090** is *not* set to **NEW** then **export(PACKAGE)** populates the user package registry unless the **CMAKE_EXPORT_NO_PACKAGE_REGISTRY** variable explicitly disables it.
- **CMAKE_FIND_USE_PACKAGE_REGISTRY** disables the User Package Registry in all the **find_package()** calls when set to **FALSE**.
- Deprecated **CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY** disables the User Package Registry in all the **find_package()** calls when set to **TRUE**. This variable is ignored when **CMAKE_FIND_USE_PACKAGE_REGISTRY** has been set.
- **CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY** disables the System Package Registry in all the **find_package()** calls.

Package Registry Example

A simple convention for naming package registry entries is to use content hashes. They are deterministic and unlikely to collide (**export(PACKAGE)** uses this approach). The name of an entry referencing a specific directory is simply the content hash of the directory path itself.

If a project arranges for package registry entries to exist, such as:

```
> reg query HKCU\Software\Kitware\CMake\Packages\MyPackage
HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\MyPackage
 45e7d55f13b87179bb12f907c8de6fc4 REG_SZ c:/Users/Me/Work/lib/cmake/MyPackage
 7b4a9844f681c80ce93190d4e3185db9 REG_SZ c:/Users/Me/Work/MyPackage−build
```

or:

```
$ cat ˜/.cmake/packages/MyPackage/7d1fb77e07ce59a81bed093bbee945bd
/home/me/work/lib/cmake/MyPackage
$ cat ˜/.cmake/packages/MyPackage/f92c1db873a1937f3100706657c63e07
/home/me/work/MyPackage−build
```

then the **CMakeLists.txt** code:

```
find_package(MyPackage)
```

will search the registered locations for package configuration files (**MyPackageConfig.cmake**). The search order among package registry entries for a single package is unspecified and the entry names (hashes in this example) have no meaning. Registered locations may contain package version files (**MyPackageConfigVersion.cmake**) to tell **find_package()** whether a specific location is suitable for the version requested.

Package Registry Ownership

Package registry entries are individually owned by the project installations that they reference. A package installer is responsible for adding its own entry and the corresponding uninstaller is responsible for removing it.

The **export(PACKAGE)** command populates the user package registry with the location of a project build tree. Build trees tend to be deleted by developers and have no "uninstall" event that could trigger removal of their entries. In order to keep the registries clean the **find_package()** command automatically removes stale entries it encounters if it has sufficient permissions. CMake provides no interface to remove an entry referencing an existing build tree once **export(PACKAGE)** has been invoked. However, if the project removes its package configuration file from the build tree then the entry referencing the location will be considered stale.

COPYRIGHT