# Appendix D – Selected Modules

## CMake Modules

This is the documentation for the modules and scripts that come with CMake. Using these modules you can check the computer system for installed software packages, features of the compiler and the existance of headers to name just a few.

**AddFileDependencies**: ADD_FILE_DEPENDENCIES(source_file depend_files...)

Adds the given files as dependencies to source_file

**BundleUtilities**:

A collection of CMake utility functions useful for dealing with .app bundles on the Mac and bundle-like directories on any OS. The following functions are provided by this module:

```
get_bundle_main_executable
get_dotapp_dir
get_bundle_and_executable
get_bundle_all_executables
get_item_key
clear_bundle_keys
set_bundle_key_values
get_bundle_keys
```

```
copy_resolved_item_into_bundle
fixup_bundle_item
fixup_bundle
copy_and_fixup_bundle
verify_bundle_prerequisites
verify_bundle_symlinks
verify_app
```

Requires CMake 2.6 or greater because it uses function, break and PARENT_SCOPE. Also depends on GetPrerequisites.cmake.

**CMakeBackwardCompatibilityCXX**: define a bunch of backwards compatibility variables

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

**CMakeDependentOption**: Macro to provide an option dependent on other options.

This macro presents an option to the user only if a set of other conditions are true. When the option is not presented a default value is used, but any value set by the user is preserved for when the option is presented again. Example invocation:

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

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

**CMakeDetermineVSServicePack**: Includes a public function for assisting users in trying to determine the

Visual Studio service pack in use. Sets the passed in variable to one of the following values or an empty string if unknown.

```
vc80
vc80sp1
```

```
vc90
vc90sp1
```

Usage:

```
if(MSVC)
   include(CMakeDetermineVSServicePack)
   DetermineVSServicePack( my_service_pack )

   if( my_service_pack )
       message(STATUS "Detected: ${my_service_pack}")
   endif()
endif()
```

**CMakeFindFrameworks**: helper module to find OSX frameworks

**CMakeForceCompiler**:

This module defines macros intended for use by cross-compiling toolchain files when CMake is not able to automatically detect the compiler identification. The macro CMAKE_FORCE_C_COMPILER has the following signature:

```
CMAKE_FORCE_C_COMPILER(<compiler> <compiler-id>)
```

It sets CMAKE_C_COMPILER to the given compiler and the cmake internal variable CMAKE_C_COMPILER_ID to the given compiler-id. It also bypasses the check for working compiler and basic compiler information tests.

The macro CMAKE_FORCE_CXX_COMPILER has the following signature:

```
CMAKE_FORCE_CXX_COMPILER(<compiler> <compiler-id>)
```

It sets CMAKE_CXX_COMPILER to the given compiler and the cmake internal variable CMAKE_CXX_COMPILER_ID to the given compiler-id. It also bypasses the check for working compiler and basic compiler information tests.

So a simple toolchain file could look like this:

```
INCLUDE (CMakeForceCompiler)
SET(CMAKE_SYSTEM_NAME Generic)
CMAKE_FORCE_C_COMPILER   (chc12 MetrowerksHicross)
CMAKE_FORCE_CXX_COMPILER (chc12 MetrowerksHicross)
```

**CMakePrintSystemInformation**: print system information

This file can be used for diagnostic purposes just include it in a project to see various internal CMake variables.

**CPack**: Build binary and source package installers

The CPack module generates binary and source installers in a variety of formats using the cpack program. Inclusion of the CPack module adds two new targets to the resulting Makefiles, package and package_source, which build the binary and source installers, respectively. The generated binary installers contain everything installed via CMake's INSTALL command (and the deprecated INSTALL_FILES, INSTALL_PROGRAMS, and INSTALL_TARGETS commands).

For certain kinds of binary installers (including the graphical installers on Mac OS X and Windows), CPack generates installers that allow users to select individual application components to install. The contents of each of the components are identified by the COMPONENT argument of CMake's INSTALL command. These components can be annotated with user-friendly names and descriptions, inter-component dependencies, etc., and grouped in various ways to customize the resulting installer. See the cpack_add_* commands, described below, for more information about component-specific installations.

Before including the CPack module, there are a variety of variables that can be set to customize the resulting installers. The most commonly-used variables are:

**CPACK_PACKAGE_NAME**

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

**CPACK_PACKAGE_VENDOR**

The name of the package vendor (e.g. "Kitware").

**CPACK_PACKAGE_VERSION_MAJOR**

Package major Version

**CPACK_PACKAGE_VERSION_MINOR**

Package minor Version

**CPACK_PACKAGE_VERSION_PATCH**

Package patch Version

**CPACK_PACKAGE_DESCRIPTION_FILE**

A text file used to describe the project. Used, for example, the introduction screen of a CPack-generated Windows installer to describe the project.

**CPACK_PACKAGE_DESCRIPTION_SUMMARY**

Short description of the project (only a few words).

**CPACK_PACKAGE_FILE_NAME**

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

**CPACK_PACKAGE_INSTALL_DIRECTORY**

Installation directory on the target system, e.g., "CMake 2.5".

**CPACK_RESOURCE_FILE_LICENSE**

License file for the project, which will typically be displayed to the user (often with an explicit "Accept" button, for graphical installers) prior to installation.

**CPACK_RESOURCE_FILE_README**

ReadMe file for the project, which typically describes in some detail

**CPACK_RESOURCE_FILE_WELCOME**

Welcome file for the project, which welcomes users to this installer. Typically used in the graphical installers on Windows and Mac OS X.

**CPACK_MONOLITHIC_INSTALL**

Disables the component-based installation mechanism, so that all components are always installed.

**CPACK_GENERATOR**

List of CPack generators to use. If not specified, CPack will create a set of options (e.g., CPACK_BINARY_NSIS) allowing the user to enable/disable individual generators.

**CPACK_OUTPUT_CONFIG_FILE**

The name of the CPack configuration file for binary installers that will be generated by the CPack module. Defaults to CPackConfig.cmake.

**CPACK_PACKAGE_EXECUTABLES**

Lists each of the executables along with a text label, to be used to create Start Menu shortcuts on Windows. For example, setting this to the list ccmake;CMake will create a shortcut named "CMake" that will execute the installed executable ccmake.

**CPACK_STRIP_FILES**

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

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

**CPACK_SOURCE_PACKAGE_FILE_NAME**

The name of the source package, e.g., cmake-2.6.1

**CPACK_SOURCE_STRIP_FILES**

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

**CPACK_SOURCE_GENERATOR**

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

**CPACK_SOURCE_OUTPUT_CONFIG_FILE**

The name of the CPack configuration file for source installers that will be generated by the CPack module. Defaults to CPackSourceConfig.cmake.

**CPACK_SOURCE_IGNORE_FILES**

Pattern of files in the source tree that won't be packaged when building a source package. This is a list of patterns, e.g., /CVS/;/\\.svn/;\\.swp$;\\.#;/#;.*~;cscope.*

The following variables are specific to the graphical installers built on Windows using the Nullsoft Installation System.

**CPACK_PACKAGE_INSTALL_REGISTRY_KEY**

Registry key used when installing this project.

**CPACK_NSIS_MUI_ICON**

The icon file (.ico) for the generated install program.

**CPACK_NSIS_MUI_UNIICON**

The icon file (.ico) for the generated uninstall program.

**CPACK_PACKAGE_ICON**

A branding image that will be displayed inside the installer.

**CPACK_NSIS_EXTRA_INSTALL_COMMANDS**

Extra NSIS commands that will be added to the install Section.

**CPACK_NSIS_EXTRA_UNINSTALL_COMMANDS**

Extra NSIS commands that will be added to the uninstall Section.

**CPACK_NSIS_COMPRESSOR**

The arguments that will be passed to the NSIS SetCompressor command.

**CPACK_NSIS_MODIFY_PATH**

If this is set to "ON", then an extra page will appear in the installer that will allow the user to choose whether the program directory should be added to the system PATH variable.

**CPACK_NSIS_DISPLAY_NAME**

The display name string that appears in the Windows Add/Remove Program control panel

**CPACK_NSIS_PACKAGE_NAME**

The title displayed at the top of the installer.

**CPACK_NSIS_INSTALLED_ICON_NAME**

A path to the executable that contains the installer icon.

**CPACK_NSIS_HELP_LINK**

URL to a web site providing assistance in installing your application.

**CPACK_NSIS_URL_INFO_ABOUT**

URL to a web site providing more information about your application.

**CPACK_NSIS_CONTACT**

Contact information for questions and comments about the installation process.

**CPACK_NSIS_CREATE_ICONS_EXTRA**

Additional NSIS commands for creating start menu shortcuts.

**CPACK_NSIS_DELETE_ICONS_EXTRA**

Additional NSIS commands to uninstall start menu shortcuts.

The following variable is specific to installers build on Mac OS X using PackageMaker:

**CPACK_OSX_PACKAGE_VERSION**

The version of Mac OS X that the resulting PackageMaker archive should be compatible with. Different versions of Mac OS X support different features. For example, CPack can only build component-based installers for Mac OS X 10.4 or newer, and can only build installers that download component son-the-fly for Mac OS X 10.5 or newer. If left blank, this value will be set to the minimum version of Mac OS X that supports the requested features. Set this variable to some value (e.g., 10.4) only if you want to guarantee that your installer will work on that version of Mac OS X, and don't mind missing extra features available in the installer shipping with later versions of Mac OS X.

The following variables are for advanced uses of CPack:

**CPACK_CMAKE_GENERATOR**

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

**CPACK_INSTALL_CMAKE_PROJECTS**

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

**CPACK_SYSTEM_NAME**

System name, defaults to the value of ${CMAKE_SYSTEM_NAME}.

**CPACK_PACKAGE_VERSION**

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

**CPACK_TOPLEVEL_TAG**

Directory for the installed files.

**CPACK_INSTALL_COMMANDS**

Extra commands to install components.

**CPACK_INSTALL_DIRECTORIES**

Extra directories to install.

Component-specific installation allows users to select specific sets of components to install during the install process. Installation components are identified by the COMPONENT argument of CMake's INSTALL commands, and should be further described by the following CPack commands:

```
cpack_add_component - Describes a CPack installation component
   named by the COMPONENT argument to a CMake INSTALL command.

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

The cmake_add_component command describes an installation component, which the user can opt to install or remove as part of the graphical installation process. compname is the name of the component, as provided to the COMPONENT argument of one or more CMake INSTALL commands.

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

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

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

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

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

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

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

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

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

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

```
cpack_add_component_group - Describes a group of related CPack
   installation components.

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

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

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

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

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

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

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

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

cpack_add_install_type(typename [DISPLAY_NAME name])
```

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

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

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

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

The cpack_configure_downloads command configures installation-time downloads of selected components. For each downloadable component, CPack will create an archive

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

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

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

The site argument is a URL where the archives for downloadable components will reside, e.g., http://www.cmake.org/files/2.6.1/installer/ All of the archives produced by CPack should be uploaded to that location.

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

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

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

**CPackRPM**: The builtin (binary) CPack RPM generator (UNIX only)

CPackRPM may be used to create RPM package using CPack. CPackRPM is a CPack generator thus it uses the CPACK_* variables used by CPack. However CPackRPM has specific features which are controlled by the specifics CPACK_RPM_* variables.You'll find a detailed usage on the wiki:

```
http://www.cmake.org/Wiki/CMake:CPackPackageGenerators
```

**CheckCCompilerFlag**: Check whether the C compiler supports a given flag.

```
CHECK_C_COMPILER_FLAG(<flag> <var>)
  <flag> - the compiler flag
  <var>  - variable to store the result
```

This internally calls the check_c_source_compiles macro. See help for CheckCSourceCompiles for a listing of variables that can modify the build.

**CheckCSourceCompiles**: Check if the given C source code compiles.

```
CHECK_C_SOURCE_COMPILES(<code> <var> [FAIL_REGEX <fail-regex>])
  <code>     - source code to try to compile
  <var>      - variable to store whether the source code compiled
  <fail-regex> - fail if test output matches this regex
```

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

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES  = list of include directories
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

**CheckCSourceRuns**: Check if the given C source code compiles and runs.

```
CHECK_C_SOURCE_RUNS(<code> <var>)
  <code>    - source code to try to compile
  <var>     - variable to store the result
              (1 for success, empty for failure)
```

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

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES = list of include directories
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

**CheckCXXCompilerFlag**: Check whether the CXX compiler supports a given flag.

```
CHECK_CXX_COMPILER_FLAG(<flag> <var>)
  <flag> - the compiler flag
  <var>  - variable to store the result
```

This internally calls the check_cxx_source_compiles macro. See help for CheckCXXSourceCompiles for a listing of variables that can modify the build.

**CheckCXXSourceCompiles**: Check if the given C++ source code compiles.

```
CHECK_CXX_SOURCE_COMPILES(<code> <var>
                          [FAIL_REGEX <fail-regex>])
  <code>     - source code to try to compile
  <var>      - variable to store whether the source code compiled
  <fail-regex> - fail if test output matches this regex
```

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

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES = list of include directories
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

**CheckCXXSourceRuns**: Check if the given C++ source code compiles and runs.

```
CHECK_CXX_SOURCE_RUNS(<code> <var>)
  <code>    - source code to try to compile
  <var>     - variable to store the result
              (1 for success, empty for failure)
```

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

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES = list of include directories
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

**CheckFortranFunctionExists**: macro which checks if the Fortran function exists

```
CHECK_FORTRAN_FUNCTION_EXISTS(FUNCTION VARIABLE)
  FUNCTION - the name of the Fortran function
  VARIABLE - variable to store the result
```

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

```
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

**CheckIncludeFile**: macro which checks the include file exists.

```
CHECK_INCLUDE_FILE(INCLUDE VARIABLE)
  INCLUDE  - name of include file
  VARIABLE - variable to return result
```

an optional third argument is the CFlags to add to the compile line or you can use CMAKE_REQUIRED_FLAGS

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

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
  CMAKE_REQUIRED_INCLUDES = list of include directories
```

**CheckIncludeFileCXX**: Check if the include file exists.

```
CHECK_INCLUDE_FILE_CXX(INCLUDE VARIABLE)

  INCLUDE  - name of include file
  VARIABLE - variable to return result
```

An optional third argument is the CFlags to add to the compile line or you can use CMAKE_REQUIRED_FLAGS. The following variables may be set before calling this macro to modify the way the check is run:

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES = list of include directories
```

**CheckIncludeFiles**: Check if the files can be included

```
CHECK_INCLUDE_FILES(INCLUDE VARIABLE)
  INCLUDE  - list of files to include
  VARIABLE - variable to return result
```

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

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES = list of include directories
```

**CheckLibraryExists**: Check if the library exists with the specified function.

```
CHECK_LIBRARY_EXISTS (LIBRARY FUNCTION LOCATION VARIABLE)
  LIBRARY  - the name of the library you are looking for
  FUNCTION - the name of the function
  LOCATION - location where the library should be found
  VARIABLE - variable to store the result
```

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

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

**CheckStructHasMember**: Check if the given struct or class has the specified member variable

```
CHECK_STRUCT_HAS_MEMBER (STRUCT MEMBER HEADER VARIABLE)
  STRUCT - the name of the struct or class you are interested in
  MEMBER - the member which existence you want to check
  HEADER - the header(s) where the prototype should be declared
  VARIABLE - variable to store the result
```

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

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

Example:

```
CHECK_STRUCT_HAS_MEMBER("struct timeval" tv_sec sys/select.h
                        HAVE_TIMEVAL_TV_SEC)
```

**CheckSymbolExists**: Check if the symbol exists in include files

```
CHECK_SYMBOL_EXISTS(SYMBOL FILES VARIABLE)
  SYMBOL   - symbol
  FILES    - include files to check
  VARIABLE - variable to return result
```

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

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES = list of include directories
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

**CheckTypeSize**: Check sizeof a type

```
CHECK_TYPE_SIZE(TYPE VARIABLE [BUILTIN_TYPES_ONLY])
```

Check if the type exists and determine size of type. if the type exists, the size will be stored to the variable. This also calls check_include_file for sys/types.h stdint.h and stddef.h, setting HAVE_SYS_TYPES_H, HAVE_STDINT_H, and HAVE_STDDEF_H. This is because many types are stored in these include files.

```
VARIABLE           - variable to store size if the type exists.
HAVE_${VARIABLE}   - does the variable exists or not
BUILTIN_TYPES_ONLY - The third argument is optional and if
                     it is set to the string BUILTIN_TYPES_ONLY
                     this macro will not check for any header
                     files.
```

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

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_INCLUDES = list of include directories
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

**CheckVariableExists**: Check if the variable exists.

```
CHECK_VARIABLE_EXISTS(VAR VARIABLE)

  VAR      - the name of the variable
  VARIABLE - variable to store the result
```

This macro is only for C variables. The following variables may be set before calling this macro to modify the way the check is run:

```
CMAKE_REQUIRED_FLAGS = string of compile command line flags
CMAKE_REQUIRED_DEFINITIONS = list of macros to define
                             e.g. (-DFOO=bar)
CMAKE_REQUIRED_LIBRARIES = list of libraries to link
```

**ExternalProject**: Create custom targets to build projects in external trees

The 'ExternalProject_Add' function creates a custom target to drive download, update/patch, configure, build, install and test steps of an external project:

```
ExternalProject_Add(<name>     # Name for custom target
  [DEPENDS projects...]    # Targets on which the project depends
  [PREFIX dir]             # Root dir for entire project
  [LIST_SEPARATOR sep]     # Sep to be replaced by ; in cmd lines
  [TMP_DIR dir]            # Directory to store temporary files
  [STAMP_DIR dir]          # Directory to store step timestamps

  #--Download step--------------
  [DOWNLOAD_DIR dir]       # Directory to store downloaded files
  [DOWNLOAD_COMMAND cmd...] # Command to download source tree
  [CVS_REPOSITORY cvsroot]  # CVSROOT of CVS repository
  [CVS_MODULE mod]          # Module to checkout from CVS repo
  [CVS_TAG tag]             # Tag to checkout from CVS repo
  [SVN_REPOSITORY url]      # URL of Subversion repo
  [SVN_REVISION rev]        # Revision to checkout from SVN repo
  [URL /.../src.tgz]        # Full path or URL of source

  #--Update/Patch step----------
  [UPDATE_COMMAND cmd...]   # Source work-tree update command
  [PATCH_COMMAND cmd...]    # Command to patch downloaded source

  #--Configure step-------------
  [SOURCE_DIR dir]          # Source dir to be used for build
```

```
  [CONFIGURE_COMMAND cmd...] # Build tree configuration command
  [CMAKE_COMMAND /.../cmake] # Specify alternative cmake exec
  [CMAKE_GENERATOR gen]      # Specify generator for native build
  [CMAKE_ARGS args...]       # Arguments to CMake command line

  #--Build step-----------------
  [BINARY_DIR dir]           # Specify build dir location
  [BUILD_COMMAND cmd...]     # Command to drive the native build
  [BUILD_IN_SOURCE 1]        # Use source dir for build dir

  #--Install step---------------
  [INSTALL_DIR dir]          # Installation prefix
  [INSTALL_COMMAND cmd...]   # Command to drive install after
                               build


  #--Test step----------------
  [TEST_BEFORE_INSTALL 1]  # Add test step before install step
  [TEST_AFTER_INSTALL 1]   # Add test step after install step
  [TEST_COMMAND cmd...]    # Command to drive test
  )
```

The *_DIR options specify directories for the project, with default directories computed as follows. If the PREFIX option is given to ExternalProject_Add() or the EP_PREFIX directory property is set, then an external project is built and installed under the specified prefix:

```
TMP_DIR      = <prefix>/tmp
STAMP_DIR    = <prefix>/src/<name>-stamp
DOWNLOAD_DIR = <prefix>/src
SOURCE_DIR   = <prefix>/src/<name>
BINARY_DIR   = <prefix>/src/<name>-build
INSTALL_DIR  = <prefix>
```

Otherwise, if the EP_BASE directory property is set then components of an external project are stored under the specified base:

```
TMP_DIR      = <base>/tmp/<name>
STAMP_DIR    = <base>/Stamp/<name>
DOWNLOAD_DIR = <base>/Download/<name>
SOURCE_DIR   = <base>/Source/<name>
BINARY_DIR   = <base>/Build/<name>
INSTALL_DIR  = <base>/Install/<name>
```

If no PREFIX, EP_PREFIX, or EP_BASE is specified then the default is to set PREFIX to "<name>-prefix". Relative paths are interpreted with respect to the build directory corresponding to the source directory in which ExternalProject_Add is invoked.

If SOURCE_DIR is explicitly set to an existing directory the project will be built from it. Otherwise a download step must be specified using one of the DOWNLOAD_COMMAND, CVS_*, SVN_*, or URL options. The URL option may refer locally to a directory or source tarball, or refer to a remote tarball (e.g. http://.../src.tgz).

The 'ExternalProject_Add_Step' function adds a custom step to an external project:

```
ExternalProject_Add_Step(
    <name> <step>           # Names of project and custom step
    [COMMAND cmd...]        # Command line invoked by this step
    [COMMENT "text..."]     # Text printed when step executes
    [DEPENDEES steps...]    # Steps on which this step depends
    [DEPENDERS steps...]    # Steps that depend on this step
    [DEPENDS files...]      # Files on which this step depends
    [ALWAYS 1]              # No stamp file, step always runs
    [WORKING_DIRECTORY dir] # Working directory for command
    )
```

The command line, comment, and working directory of every standard and custom step is processed to replace tokens <SOURCE_DIR>, <BINARY_DIR>, <INSTALL_DIR>, and <TMP_DIR> with corresponding property values.

The 'ExternalProject_Get_Property' function retrieves external project target properties:

```
ExternalProject_Get_Property(<name> [prop1 [prop2 [...]]])
```

It stores property values in variables of the same name. Property names correspond to the keyword argument names of 'ExternalProject_Add'.

**FeatureSummary**: Macros for generating a summary of enabled/disabled features

PRINT_ENABLED_FEATURES() - Print a summary of all enabled features. By default all successfull FIND_PACKAGE() calls will appear here, except the ones which used the QUIET keyword. Additional features can be added by appending an entry to the global ENABLED_FEATURES property. If SET_FEATURE_INFO() is used for that feature, the output will be much more informative.

PRINT_DISABLED_FEATURES() - Same as PRINT_ENABLED_FEATURES(), but for disabled features. It can be extended the same way by adding to the global property DISABLED_FEATURES.

SET_FEATURE_INFO(NAME DESCRIPTION [URL [COMMENT] ] ) - Use this macro to set up information about the named feature, which will then be displayed by PRINT_ENABLED/DISABLED_FEATURES(). For Example:

```
SET_FEATURE_INFO(LibXml2 "XML processing library."
                 "http://xmlsoft.org/")
```

**FindALSA**: Find alsa

**FindASPELL**: Try to find ASPELL

**FindAVIFile**: Locate AVIFILE library and include paths

**FindBISON**: Find bison executable and provides macros to generate custom build rules

**FindBLAS**: Find BLAS library

**FindBZip2**: Try to find BZip2

**FindBoost**: Try to find Boost include dirs and libraries

Usage of this module is as follows. Using Header-Only libraries from within Boost:

```
find_package( Boost 1.36.0 )
if(Boost_FOUND)
      include_directories(${Boost_INCLUDE_DIRS})
      add_executable(foo foo.cc)
endif()
```

Using actual libraries from within Boost:

```
set(Boost_USE_STATIC_LIBS   ON)
set(Boost_USE_MULTITHREADED ON)
find_package( Boost 1.36.0 COMPONENTS date_time filesystem
system ... )
```

```
if(Boost_FOUND)
      include_directories(${Boost_INCLUDE_DIRS})
      add_executable(foo foo.cc)
      target_link_libraries(foo ${Boost_LIBRARIES})
endif()
```

The components list needs to contain actual names of boost libraries only, such as "date_time" for "libboost_date_time". If you're using parts of Boost that contain header files only (e.g. foreach) you do not need to specify COMPONENTS.

You should provide a minimum version number that should be used. If you provide this version number and specify the REQUIRED attribute, this module will fail if it can't find the specified or a later version. If you specify a version number this is automatically put into the considered list of version numbers and thus doesn't need to be specified in the Boost_ADDITIONAL_VERSIONS variable (see below).

NOTE for Visual Studio Users: Automatic linking is used on MSVC & Borland compilers by default when including things in Boost. It's important to note that setting Boost_USE_STATIC_LIBS to OFF is NOT enough to get you dynamic linking, should you need this feature. Automatic linking typically uses static libraries with a few exceptions (Boost.Python is one).

Please see the section below near Boost_LIB_DIAGNOSTIC_DEFINITIONS for more details. Adding a TARGET_LINK_LIBRARIES() as shown in the example above appears to cause VS to link dynamically if Boost_USE_STATIC_LIBS gets set to OFF. It is suggested you avoid automatic linking since it will make your application less portable.

### *Boost_ADDITIONAL_VERSIONS*

OK, so the Boost_ADDITIONAL_VERSIONS variable can be used to specify a list of boost version numbers that should be taken into account when searching for Boost. Unfortunately boost puts the version number into the actual filename for the libraries, so this variable will certainly be needed in the future when new Boost versions are released.

Currently this module searches for the following version numbers: 1.33, 1.33.0, 1.33.1, 1.34, 1.34.0, 1.34.1, 1.35, 1.35.0, 1.35.1, 1.36, 1.36.0, 1.36.1, 1.37, 1.37.0, 1.38, 1.38.0, 1.39, 1.39.0, 1.40, 1.40.0

NOTE: If you add a new major 1.x version in Boost_ADDITIONAL_VERSIONS you should add both 1.x and 1.x.0 as shown above. Official Boost include directories omit the 3rd version number from include paths if it is 0 although not all binary Boost releases do so.

```
set(Boost_ADDITIONAL_VERSIONS "0.99" "0.99.0" "1.78" "1.78.0")
```

Variables used by this module, they can change the default behaviour and need to be set before calling find_package:

**Boost_USE_MULTITHREADED**

Can be set to OFF to use the non-multithreaded boost libraries. If not specified, defaults to ON.

**Boost_USE_STATIC_LIBS**

Can be set to ON to force the use of the static boost libraries. Defaults to OFF.

Further documentation on other variables used by this module which you may want to set can be found in the module itself.

**FindBullet**: Try to find the Bullet physics engine

This module defines the following variables

```
BULLET_FOUND         - Was bullet found
BULLET_INCLUDE_DIRS  - the Bullet include directories
BULLET_LIBRARIES     - Link to this, by default it includes
                       all bullet components (Dynamics,
                       Collision, LinearMath, & SoftBody)
```

This module accepts the following variables:

```
BULLET_ROOT - Can be set to bullet install path or Windows build
path
```

**FindCABLE**: Find CABLE

**FindCUDA**: Tools for building CUDA C files: libraries and build dependencies.

This script locates the NVIDIA CUDA C tools. It should work on linux, windows, and mac and should be reasonably up to date with CUDA C releases. This script makes use of the standard find_package arguments of <VERSION>, REQUIRED and QUIET. CUDA_FOUND will report if an acceptable version of CUDA was found.

**FindCURL**: Find curl

**FindCVS**:

**FindCoin3D**: Find Coin3D (Open Inventor)

**FindCups**: Try to find the Cups printing system

**FindCurses**: Find the curses include file and library

**FindCxxTest**: Find CxxTest

Find the CxxTest suite and declare a helper macro for creating unit tests and integrating them with CTest. For more details on CxxTest see http://cxxtest.tigris.org

INPUT Variables

```
CXXTEST_USE_PYTHON
       If true, the CXXTEST_ADD_TEST macro will use
       the Python test generator instead of Perl.
```

OUTPUT Variables

```
CXXTEST_FOUND
       True if the CxxTest framework was found
CXXTEST_INCLUDE_DIR
       Where to find the CxxTest include directory
CXXTEST_PERL_TESTGEN_EXECUTABLE
       The perl-based test generator.
CXXTEST_PYTHON_TESTGEN_EXECUTABLE
       The python-based test generator.
```

MACROS for optional use by CMake users:

```
CXXTEST_ADD_TEST(<test_name> <gen_source_file>
                 <input_files_to_testgen...>)

Creates a CxxTest runner and adds it to the CTest testing suite
Parameters:
   test_name                The name of the test
   gen_source_file          The generated source filename to be
                            generated by CxxTest
   input_files_to_testgen   The list of header files containing
                            the CxxTest::TestSuite's to be
                            included in this runner
```

Sample usage:

```
find_package(CxxTest)
if(CXXTEST_FOUND)
   include_directories(${CXXTEST_INCLUDE_DIR})
   enable_testing()
   CXXTEST_ADD_TEST(unittest_foo foo_test.cc
                    ${CMAKE_CURRENT_SOURCE_DIR}/foo_test.h)
   target_link_libraries(unittest_foo foo) # as needed
endif()
```

This will (if CxxTest is found):

1. Invoke the testgen executable to autogenerate foo_test.cc in the binary tree from "foo_test.h" in the current source directory.
2. Create an executable and test called unittest_foo.

**FindCygwin**: this module looks for Cygwin

**FindDCMTK**: find DCMTK libraries

**FindDart**: Find DART

**FindDevIL**: This module locates the developer's image library.

**FindDoxygen**: This module looks for Doxygen and the path to Graphviz's dot

**FindEXPAT**: Find the native EXPAT headers and libraries.

**FindFLEX**: Find flex executable and provides a macro to generate custom build rules

**FindFLTK**: Find the native FLTK includes and library

**FindFLTK2**: Find the native FLTK2 includes and library

**FindFreetype**: Locate FreeType library

**FindGCCXML**: Find the GCC-XML front-end executable.

**FindGDAL**: Locate gdal

**FindGIF**:

**FindGLUT**: try to find glut library and include files

**FindGTK**: try to find GTK (and glib) and GTKGLArea

**FindGTK2**: FindGTK2.cmake

**FindGTest**: Locate the Google C++ Testing Framework.

**FindGettext**: Find GNU gettext tools

**FindGnuTLS**: Try to find the GNU Transport Layer Security library (gnutls)

**FindGnuplot**: this module looks for gnuplot

**FindHDF5**: Find HDF5, a library for reading and writing self describing array data.

This module invokes the HDF5 wrapper compiler that should be installed alongside HDF5. Depending upon the HDF5 Configuration, the wrapper compiler is called either h5cc or h5pcc. If this succeeds, the module will then call the compiler with the -show argument to see what flags are used when compiling an HDF5 client application.

**FindHSPELL**: Try to find HSPELL

**FindHTMLHelp**: This module looks for Microsoft HTML Help Compiler

**FindITK**: Find an ITK installation or build tree.

**FindImageMagick**: Find the ImageMagick binary suite.

**FindJNI**: Find JNI java libraries.

**FindJPEG**: Find the native JPEG includes and library.

**FindJasper**: Try to find the Jasper JPEG2000 library

**FindJava**: This module finds if Java is installed and determines where the include files and libraries are.

**FindKDE3**: Find the KDE3 include and library dirs, KDE preprocessors and define some macros.

**FindKDE4**: Find KDE4 and provide all necessary variables and macros to compile software for it.

**FindLAPACK**: Find the LAPACK library

**FindLATEX**: This module finds if Latex is installed and determines where the executables are.

**FindLibXml2**: Try to find LibXml2

**FindLibXslt**: Try to find LibXslt

**FindLua50**: Find Lua version 5.0

**FindLua51**: Find Lua version 5.1

**FindMFC**: Find MFC on Windows

**FindMPEG**: Find the native MPEG includes and library

**FindMPEG2**: Find the native MPEG2 includes and library

**FindMPI**: Message Passing Interface (MPI) module.

**FindMatlab**: this module looks for Matlab

**FindMotif**: Try to find Motif (or lesstif)

**FindOpenAL**:

**FindOpenGL**: Try to find OpenGL

**FindOpenMP**: Finds OpenMP support

**FindOpenSSL**: Try to find the OpenSSL encryption library

**FindOpenSceneGraph**: Comprehensive module to find OpenSceneGraph and all of its various parts.

**FindOpenThreads**: Find the OpenThreads, C++ based threading library.

**FindPHP4**: Find PHP4

**FindPNG**: Find the native PNG includes and library

**FindPackageHandleStandardArgs**:

```
FIND_PACKAGE_HANDLE_STANDARD_ARGS(NAME
   (DEFAULT_MSG|"Custom failure message") VAR1 ... )
```

This macro is intended to be used in FindXXX.cmake modules files. It handles the REQUIRED and QUIET argument to FIND_PACKAGE() and it also sets the <UPPERCASED_NAME>_FOUND variable. The package is found if all variables listed are TRUE. For example:

```
FIND_PACKAGE_HANDLE_STANDARD_ARGS(LibXml2 DEFAULT_MSG
                                  LIBXML2_LIBRARIES
                                  LIBXML2_INCLUDE_DIR)
```

LibXml2 is considered to be found, if both LIBXML2_LIBRARIES and LIBXML2_INCLUDE_DIR are valid. Then also LIBXML2_FOUND is set to TRUE. If it is not found and REQUIRED was used, it fails with FATAL_ERROR, independent whether QUIET was used or not. If it is found, the location is reported using the VAR1 argument, so here a message "Found LibXml2: /usr/lib/libxml2.so" will be printed out. If the second argument is DEFAULT_MSG, the message in the failure case will be "Could NOT find LibXml2", if you don't like this message you can specify your own custom failure message there.

**FindPackageMessage**:

```
FIND_PACKAGE_MESSAGE(<name> "message for user"
                     "find result details")
```

This macro is intended to be used in FindYYY.cmake modules files. It will print a message once for each unique find result. This is useful for telling the user where a package was found. The first argument specifies the name (YYY) of the package. The second argument specifies the message to display. The third argument lists details about the find result so that if they

change the message will be displayed again. The macro also obeys the QUIET argument to the find_package command. For example:

```
if(X11_FOUND)
  FIND_PACKAGE_MESSAGE(X11 "Found X11: ${X11_X11_LIB}"
      "[${X11_X11_LIB}][${X11_INCLUDE_DIR}]")
else()
   ...
endif ()
```

**FindPerl**: Find perl

**FindPerlLibs**: Find Perl libraries

**FindPhysFS**: Locate the PhysFS library

**FindPike**: This module finds PIKE and determines where the include files and libraries are.

**FindPkgConfig**: a pkg-config module for CMake

Usage:

```
pkg_check_modules(<PREFIX> [REQUIRED] <MODULE> [<MODULE>]*)
     checks for all the given modules
```

```
pkg_search_module(<PREFIX> [REQUIRED] <MODULE> [<MODULE>]*)
     checks for given modules and uses the first working one
```

When the 'REQUIRED' argument was set, macros will fail with an error when module(s) could not be found

It sets the following variables:

```
PKG_CONFIG_FOUND       ... true if pkg-config works on the system
PKG_CONFIG_EXECUTABLE  ... pathname of the pkg-config program
<PREFIX>_FOUND         ... set to 1 if module(s) exist
```

For the following variables two sets of values exist; first one is the common one and has the given PREFIX. The second set contains flags which are given out when pkgconfig was called with the '--static' option.

```
<XPREFIX>_LIBRARIES      ... only the libraries (w/o the '-l')
<XPREFIX>_LIBRARY_DIRS   ... the paths of the libraries
                             (w/o the '-L')
<XPREFIX>_LDFLAGS        ... all required linker flags
<XPREFIX>_LDFLAGS_OTHER  ... all other linker flags
<XPREFIX>_INCLUDE_DIRS   ... the '-I' preprocessor flags
                             (w/o the '-I')
<XPREFIX>_CFLAGS         ... all required cflags
<XPREFIX>_CFLAGS_OTHER   ... the other compiler flags

<XPREFIX> = <PREFIX>         for the common case
<XPREFIX> = <PREFIX>_STATIC for static linking
```

There are some special variables whose prefix depends on the count of given modules. When there is only one module, <PREFIX> stays unchanged. When there are multiple modules, the prefix will be changed to <PREFIX>_<MODNAME>:

```
<XPREFIX>_VERSION     ... version of the module
<XPREFIX>_PREFIX      ... prefix-directory of the module
<XPREFIX>_INCLUDEDIR  ... include-dir of the module
<XPREFIX>_LIBDIR      ... lib-dir of the module

<XPREFIX> = <PREFIX>  when |MODULES| == 1, else
<XPREFIX> = <PREFIX>_<MODNAME>
```

A <MODULE> parameter can have the following formats:

```
{MODNAME}              ... matches any version
{MODNAME}>={VERSION}   ... at least version <VERSION> is required
{MODNAME}={VERSION}    ... exactly version <VERSION> is required
{MODNAME}<={VERSION}   ... modules must not be newer than
                           <VERSION>
```

Examples

```
pkg_check_modules (GLIB2   glib-2.0)

pkg_check_modules (GLIB2   glib-2.0>=2.10)
     requires at least version 2.10 of glib2 and defines e.g.
       GLIB2_VERSION=2.10.3

pkg_check_modules (FOO     glib-2.0>=2.10 gtk+-2.0)
     requires both glib2 and gtk2, and defines e.g.
       FOO_glib-2.0_VERSION=2.10.3
       FOO_gtk+-2.0_VERSION=2.8.20

pkg_check_modules (XRENDER REQUIRED xrender)
     defines e.g.:
       XRENDER_LIBRARIES=Xrender;X11
       XRENDER_STATIC_LIBRARIES=Xrender;X11;pthread;Xau;Xdmcp

pkg_search_module (BAR     libxml-2.0 libxml2 libxml>=2)
```

**FindProducer**: Find the producer library

**FindProtobuf**: Locate and configure the Google Protocol Buffers library.

**FindPythonInterp**: Find python interpreter

**FindPythonLibs**: Find python libraries

**FindQt**: Searches for all installed versions of QT.

**FindQt3**: Locate Qt 3 include paths and libraries

**FindQt4**: This module can be used to find Qt4.

**FindQuickTime**: Locate QuickTime library and includes.

**FindRTI**: Try to find M&S HLA RTI libraries and includes.

**FindRuby**: This module finds Ruby and determines where the include files and libraries are.

**FindSDL**: Locate the SDL Library and includes.

**FindSDL_image**: Locate SDL_image library.

**FindSDL_mixer**: Locate SDL_mixer library.

**FindSDL_net**: Locate SDL_net library.

**FindSDL_sound**: Locates the SDL_sound library

**FindSDL_ttf**: Locate SDL_ttf library.

**FindSWIG**: Find the SWIG wrapper generator.

**FindSelfPackers**: Find upx

**FindSquish**: This module can be used to find Squish.

**FindSubversion**: Extract information from a subversion client

The module defines the following variables:

```
Subversion_SVN_EXECUTABLE - path to svn command line client
Subversion_VERSION_SVN - version of svn command line client
Subversion_FOUND - true if the command line client was found
```

If the command line client executable is found the macro

```
Subversion_WC_INFO(<dir> <var-prefix>)
```

is defined to extract information of a subversion working copy at a given location. The macro defines the following variables:

```
<var-prefix>_WC_URL - url of the repository (at <dir>)
<var-prefix>_WC_ROOT - root url of the repository
<var-prefix>_WC_REVISION - current revision
<var-prefix>_WC_LAST_CHANGED_AUTHOR - author of last commit
<var-prefix>_WC_LAST_CHANGED_DATE - date of last commit
<var-prefix>_WC_LAST_CHANGED_REV - revision of last commit
<var-prefix>_WC_LAST_CHANGED_LOG - last log of base revision
<var-prefix>_WC_INFO - output of command `svn info <dir>'
```

Example usage:

```
find_package(Subversion)
if(Subversion_FOUND)
  Subversion_WC_INFO(${PROJECT_SOURCE_DIR} Project)
  message ("Current revision is ${Project_WC_REVISION}")
  Subversion_WC_LOG(${PROJECT_SOURCE_DIR} Project)
  message ("Last changed log is ${Project_LAST_CHANGED_LOG}")
endif()
```

**FindTCL**: This module finds if Tcl is installed and locates the include files and libraries.

**FindTIFF**: Find the native TIFF includes and library.

**FindTclStub**: This module finds Tcl stub libraries.

**FindTclsh**: Find tclsh

**FindThreads**: This module determines the thread library of the system.

The following variables are set

```
CMAKE_THREAD_LIBS_INIT      - the thread library
CMAKE_USE_SPROC_INIT        - are we using sproc?
CMAKE_USE_WIN32_THREADS_INIT - using WIN32 threads?
CMAKE_USE_PTHREADS_INIT     - are we using pthreads
CMAKE_HP_PTHREADS_INIT      - are we using hp pthreads
```

**FindUnixCommands**: Find UNIX commands from cygwin

**FindVTK**: Find a VTK installation or build tree.

**FindWget**: Find wget

**FindWish**: Find wish installation

**FindX11**: Find X11 installation

**FindXMLRPC**: Find the native XMLRPC headers and libraries.

**FindZLIB**: Find the native ZLIB includes and library

**Findosg***: Find a specific part of open scene graph. See the FindOpenSceneGraph module as well.

**FindwxWidgets**: Find a wxWidgets (a.k.a., wxWindows) installation.

**FortranCInterface**: Fortran/C Interface Detection

This module automatically detects the API by which C and Fortran languages interact. Variables indicate if the mangling is found:

```
FortranCInterface_GLOBAL_FOUND
   = Global subroutines and functions
FortranCInterface_MODULE_FOUND
   = Module subroutines and functions
     (declared by "MODULE  PROCEDURE")
```

A function is provided to generate a C header file containing macros to mangle symbol names:

```
FortranCInterface_HEADER(<file>
                         [MACRO_NAMESPACE <macro-ns>]
                         [SYMBOL_NAMESPACE <ns>]
                         [SYMBOLS [<module>:]<function> ...])
```

It generates in <file> definitions of the following macros:

```
#define FortranCInterface_GLOBAL (name,NAME) ...
#define FortranCInterface_GLOBAL_(name,NAME) ...
#define FortranCInterface_MODULE (mod,name, MOD,NAME) ...
#define FortranCInterface_MODULE_(mod,name, MOD,NAME) ...
```

These macros mangle four categories of Fortran symbols, respectively:

```
- Global symbols without '_': call mysub()
- Global symbols with '_'   : call my_sub()
- Module symbols without '_': use mymod; call mysub()
- Module symbols with '_'   : use mymod; call my_sub()
```

If mangling for a category is not known, its macro is left undefined. All macros require raw names in both lower case and upper case. The MACRO_NAMESPACE option replaces the default "FortranCInterface_" prefix with a given namespace "<macro-ns>". The SYMBOLS option lists symbols to mangle automatically with C preprocessor definitions:

```
<function>          ==> #define <ns><function> ...
<module>:<function> ==> #define <ns><module>_<function> ...
```

If the mangling for some symbol is not known then no preprocessor definition is created, and a warning is displayed. The SYMBOL_NAMESPACE option prefixes all preprocessor definitions generated by the SYMBOLS option with a given namespace "<ns>". Example usage:

```
include(FortranCInterface)
FortranCInterface_HEADER(FC.h MACRO_NAMESPACE "FC_")
```

This creates a "FC.h" header that defines mangling macros FC_GLOBAL(), FC_GLOBAL_(), FC_MODULE(), and FC_MODULE_(). Another example:

```
include(FortranCInterface)
FortranCInterface_HEADER(FCMangle.h
                         MACRO_NAMESPACE "FC_"
                         SYMBOL_NAMESPACE "FC_"
                         SYMBOLS mysub mymod:my_sub)
```

This creates a "FC.h" header that defines the same FC_*() mangling macros as the previous example plus preprocessor symbols FC_mysub and FC_mymod_my_sub. Another function is provided to verify that the Fortran and C/C++ compilers work together:

```
FortranCInterface_VERIFY([CXX] [QUIET])
```

It tests whether a simple test executable using Fortran and C (and C++ when the CXX option is given) compiles and links successfully. The result is stored in the cache entry FortranCInterface_VERIFIED_C (or FortranCInterface_VERIFIED_CXX if CXX is given) as a boolean. If the check fails and QUIET is not given the function terminates with a FATAL_ERROR message describing the problem. The purpose of this check is to stop a build early for incompatible compiler combinations.

FortranCInterface is aware of possible GLOBAL and MODULE manglings for many Fortran compilers, but it also provides an interface to specify new possible manglings. Set the variables

```
FortranCInterface_GLOBAL_SYMBOLS
FortranCInterface_MODULE_SYMBOLS
```

before including FortranCInterface to specify manglings of the symbols "MySub", "My_Sub", "MyModule:MySub", and "My_Module:My_Sub". For example, the code:

```
set(FortranCInterface_GLOBAL_SYMBOLS mysub_ my_sub__ MYSUB_)
set(FortranCInterface_MODULE_SYMBOLS
        __mymodule_MOD_mysub __my_module_MOD_my_sub)
include(FortranCInterface)
```

tells FortranCInterface to try given GLOBAL and MODULE manglings. (The carets point at raw symbol names for clarity in this example but are not needed.)

**GetPrerequisites**: See section 4.11

**InstallRequiredSystemLibraries**: See section 4.11

By including this file, all files in the CMAKE_INSTALL_DEBUG_LIBRARIES, will be installed with INSTALL_PROGRAMS into /bin for WIN32 and /lib for non-win32. If CMAKE_SKIP_INSTALL_RULES is set to TRUE before including this file, then the INSTALL command is not called. The use can use the variable CMAKE_INSTALL_SYSTEM_RUNTIME_LIBS to use a custom install command and install them into any directory they want. If it is the MSVC compiler, then the microsoft run time libraries will be found add automatically added to the CMAKE_INSTALL_DEBUG_LIBRARIES, and installed. If

CMAKE_INSTALL_DEBUG_LIBRARIES is set and it is the MSVC compiler, then the debug libraries are installed when available. If CMAKE_INSTALL_MFC_LIBRARIES is set then the MFC run time libraries are installed as well as the CRT run time libraries.

**SelectLibraryConfigurations**:

```
select_library_configurations( basename )
```

This macro takes a library base name as an argument, and will choose good values for basename_LIBRARY, basename_LIBRARIES, basename_LIBRARY_DEBUG, and basename_LIBRARY_RELEASE depending on what has been found and set. If only basename_LIBRARY_RELEASE is defined, basename_LIBRARY, basename_LIBRARY_DEBUG, and basename_LIBRARY_RELEASE will be set to the release value. If only basename_LIBRARY_DEBUG is defined, then basename_LIBRARY, basename_LIBRARY_DEBUG and basename_LIBRARY_RELEASE will take the debug value.

If the generator supports configuration types, then basename_LIBRARY and basename_LIBRARIES will be set with debug and optimized flags specifying the library to be used for the given configuration. If no build type has been set or the generator in use does not support configuration types, then basename_LIBRARY and basename_LIBRARIES will take only the release values.

**TestBigEndian**: Define macro to determine endian type

```
TEST_BIG_ENDIAN(VARIABLE)
  VARIABLE - variable to store the result to
```

**TestCXXAcceptsFlag**: Test CXX compiler for a flag

Check if the CXX compiler accepts a flag

```
  Macro CHECK_CXX_ACCEPTS_FLAG(FLAGS VARIABLE) -
     checks if the flag is accepted
  FLAGS - the flags to try
  VARIABLE - variable to store the result
```

**TestForANSIForScope**: Check for ANSI for scope support

```
CMAKE_NO_ANSI_FOR_SCOPE - holds result
```

**TestForANSIStreamHeaders**: Test for compiler support of ANSI stream headers

Check if we they have the standard ansi stream files (without the .h)

```
CMAKE_NO_ANSI_STREAM_HEADERS - defined by the results
```

**TestForSSTREAM**:

```
CMAKE_NO_ANSI_STRING_STREAM - defined by the results
```

**TestForSTDNamespace**: Test for std:: namespace support

check if the compiler supports std:: on stl classes

```
CMAKE_NO_STD_NAMESPACE - defined by the results
```

**UseEcos**: This module defines variables and macros required to build eCos applications.

**UseQt4**: Use Module for QT4

Sets up C and C++ to use Qt 4. It is assumed that FindQt.cmake has already been loaded. See FindQt.cmake for information on how to load Qt 4 into your CMake project.

**UseSWIG**: SWIG module for CMake

**Use_wxWindows**:

**UsewxWidgets**: Convenience include for using wxWidgets library