Introduction

The Cross-Platform Make facility (CMake) manages the build process—in a portable manner—across Windows, Unix and the Mac OSX platforms. CMake can be used to compile source code, create libraries, and build executables in arbitrary combinations. On Unix platforms, CMake produces makefiles that may be used with the standard make facility. In the Microsoft Visual C++ environment, CMake creates projects and workspaces that can be imported into MSVC.

CMake is designed to support complex directory hierarchies and applications dependent on several libraries. For example, CMake supports projects consisting of multiple toolkits (i.e., libraries), where each toolkit might contain several directories, and the application depends on the toolkits plus additional code. CMake can also handle situations where executables must be built in order to generate code that is then compiled and linked into a final application.

Using CMake is simple. The build process is controlled by creating a CMakeLists.txt file in each directory (including subdirectories) of a project. Each CMakeLists.txt file consists of one or more commands. Each command has the form COMMAND (args…) where COMMAND is the name of the command, and args is a white-space separated list of arguments. CMake provides many pre-defined commands, but if you need to, you can add your own commands. In addition, the advanced user can add other makefile generators for particular compiler/OS combinations.

Installing CMake

You can download CMake pre compiled binararies or CMake source code from the following link: http://www.cmake.org/CMake/HTML/Download.html.

From the source distribution you can build CMake on Windows by loading the CMake/Source/CMakeSetup.dsw file into Microsoft Visual Studio, then selecting CMakeSetup and the active project and building. On UNIX you can build and install CMake by running:

```
cd CMake
./configure
make
make install
```

[the make install step is optional, cmake can run directly from the build directory if you want.]

On UNIX, if you are not using the GNU C++ compiler, you need to tell configure which compiler you want to use. This is done by setting the environment variable CXX **before running configure**. If you need to use any special flags with your compiler use the CXXFLAGS variable.

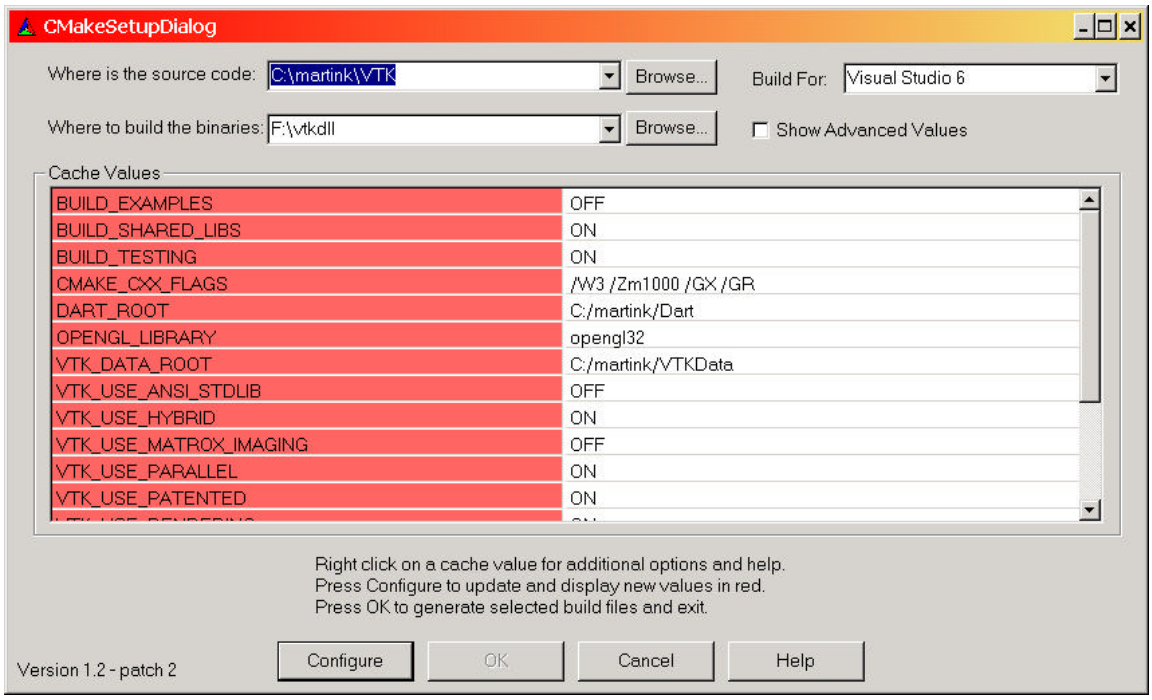
For example on the SGI with the 7.3X compiler, you build like this:
cd CMake
(setenv CXX CC; setenv CXXFLAGS "-LANG:std"; ./configure)
make
make install
[again make install is optional]

Running CMake

Once CMake has been installed on your system using it to build a project is easy. We will cover the process for Windows and then UNIX.

Running CMake for Windows / Microsoft Visual C++ (MSVC)

Run CMakeSetup.exe, which should be in your Start menu under Program Files, there may also be a shortcut on your desktop, or if you built from source, it will be in the build directory. A GUI will appear similar to what is shown below (but possibly different as CMake is still being developed). The top two entries are the source code and binary directories. They allow you to specify where the source code is for what you wanted to compile and where the resulting binaries should be placed. You should set these two values first. If the binary directory you specify does not exist, it will be created for you. The Build for option, allows you to select which type of build files are generated. Currently, on windows, visual studio 7 (.NET), visual studio 6, NMake makefiles and Borland makefiles are supported.

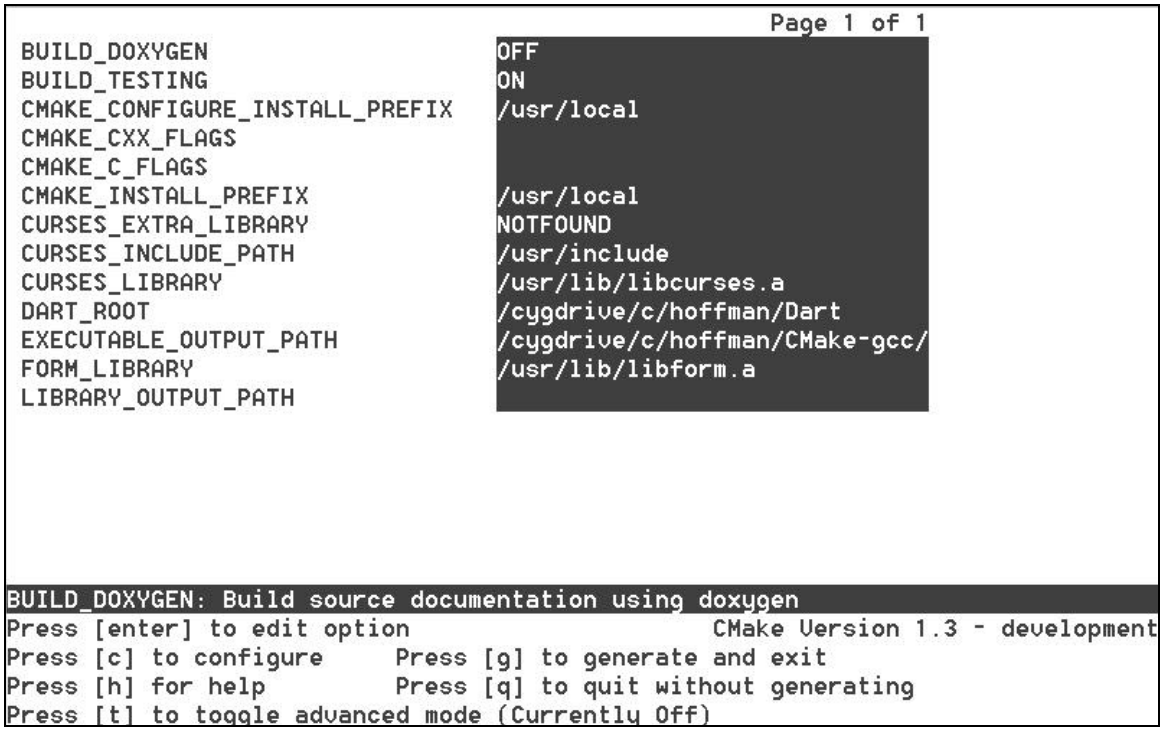


The cache values area is where you can specify different options for the build process. The example shown below is for VTK that has a large number of options. More obscure variables maybe hidden, but can be seen if you click on the “Show Advanced Values” button. Once you have specified the source code and binary directories you should click the Configure button. This will cause CMake to read in the CMakeLists.txt files from the source code directory and the cache area to be updated to display any new options for the project. Adjust your cache settings if desired and click the Configure button again. New values that were caused by the configure process will be colored red. To be sure you have seen all possible values you should click Configure until no values are red and you are happy with all the settings. Once you are done configuring, click the OK button, this will produce Microsoft Visual C++ workspaces and exit CMakeSetup.exe.

CMakeSetup.exe generates the build files in the binary directory you specified. If Visual Studio 6 or 7 was selected as the Build For option, a MSVC workspace file is created. Typically this file has the same name as what you are compiling (e.g. VTK.dsw, VTK.sln, ITK.dsw, ITK.sln etc). For the other Build For types, makefiles are generated.
The next step in this process is to open the workspace with MSVC. Once open, the project can be built in the normal manner of Microsoft Visual C++. The ALL_BUILD target can be used to build all of the libraries and executables in the package. If you are using a makefile build type, then you can follow the Unix instructions.

Running CMake on Unix

On most unix platforms, if the curses library is supported, cmake will build an executable called ccmake. This interface is a terminal based text application that is very similar to the windows GUI. To run ccmake, change directories into the directory where you want the binaries to be placed. This can be the same directory as the source code for what we call in-place builds (the binaries are in the same place as the source code) or it can be a new directory you create. Then run ccmake with either no arguments for an in-place-build, or with the path to the source directory on the command line. This will start the text interface that looks something like this:



If you hit the "c" key, it will configure the project. You should use that as you change values in the cache. To change values, use the arrow keys to select cache entries, and the enter key to edit them. Boolean values will toggle with the enter key. Once you have set all the values as you like, you can hit the 'G" key to generate the makefiles and exit. You can also hit "h" for help, "q" to quit, and "t" to toggle the viewing of advanced cache entries.

Two examples of CMake usage on the Unix platform follow for a hello world project called Hello. In the first example, and in-place build is performed, i.e., the binaries are placed in the same directory as the source code.

```
cd Hello
ccmake
make
```

In the second example, an out-of-place build is performed, i.e., the source code, libraries, and executables are produced in a directory separate from the source code directory(ies).

```
mkdir Hello-Linux
cd Hello-Linux
ccmake ../Hello
make
```

Running CMake from the command line

From the command line, cmake can be run as an interactive question and answer session or as a non-interactive program. To run in interactive mode, just pass the option "-i" to cmake. This will cause cmake to ask you to enter a value for each value in the cache file for the project. The process stops when there are no longer any more questions to ask.

Using CMake to build a project in non-interactive mode is a simple process if the project does not have many options. For larger projects like VTK, using ccmake, cmake –i, or CMakeSetup is recommended. This is because as you change options in the CMakeCache.txt file, cmake may add new entries to that file. It can be difficult to know when to stop the run cmake, edit the cache file cycle without the aid of an interface.

To build with just cmake change directory into where you want the binaries to be placed. For an in-place build you then run cmake and it will produce a CMakeCache.txt file that contains build options that you can adjust using any text editor. For non in-place builds the process is the same except you run cmake and provide the path to the source code as its argument. Once you have edited the CMakeCache.txt file you rerun cmake, repeat this process until you are happy with the cache settings. The type make and your project should compile. Some projects will have install targets as well so you can type make install to install them.

When running cmake from the command line, it is possible to specify command line options to cmake that will set values in the cache. This is done with a –DVARIABLE:TYPE=VALUE syntax on the command line. This is useful for non-interactive nightly test builds.

What is the CMake cache?

The cache is best thought of as a configuration file. Indeed Unix users could consider the cache as equivalent to the set of flags passed to the configure command. The first time CMake is run, it produces a CMakeCache.txt file. This file contains things like the existence and location of native JPEG library. The entries are added in response to certain CMake commands (e.g. FIND_LIBRARY) as they are processed anywhere in CMakeLists files anywhere in the source tree.

After CMake has been run, and created a CMakeCache.txt file - you may edit it. The CMake GUI, will allow you to edit the options easily, or you can edit the file directly. The main reason for editing the cache would be to give CMake the location of a native library such as JPEG, or to stop it from using a native library and use a version of the library in your source tree.

CMake will not alter an existing entry in the cache file itself. If your CMakeLists.txt files change significantly, you will need to remove the relevant entries from the cache file. If you have not already hand-edited the cache file, you could just delete it before re-running CMake.

Why do I have to edit the cache more than once for some projects?

Some projects are very complex and setting one value in the cache may cause new options to appear the next time the cache is built. For example, VTK supports the use of MPI for performing distributed computing. This requires the build process to determine where the MPI libraries and header files are and to let the user adjust their values. But MPI is only available if another option VTK_USE_PARALLEL is first turned on in VTK. So to avoid confusion for people who don't know what MPI is, we hide those options until VTK_USE_PARALLEL is turned on. So CMake shows the VTK_USE_PARALLEL option in the cache area, if the user turns that on and rebuilds the cache, new options will appear for MPI that they can then set. The rule is to keep building the cache until it doesn't change. For most projects this will be just once. For some complicated ones it will be twice.

Developer's Guide

This section describes how to use CMake from the software developer's point of view. That is, if your aim is to use CMake to manage your build process, read this section first. An Extension Guide follows later in this document to explain the internals of CMake, and how to setup the CMake environment. Read that section only if you plan to install, extend, or enhance the features of CMake. This section of the User's Guide begins with a description of the CMake inputs. Examples then follow to clarify these descriptions.

Input to CMake

CMake's input is the text file CMakeLists.txt in your source directory. This input file specifies the things that need to be built in the current directory. The CMakeLists.txt consists of one or more commands. Each command is of the form:

```
COMMAND(args…)
```

Where COMMAND is the name of the command, and args is a white-space separated list of arguments to the command. (Arguments with embedded white-space should be quoted.) Typically there will be a CMakeLists.txt file for each directory of the project. Let's start with a simple example. Consider building hello world. You would have a source tree with the following files:

Hello.c CMakeLists.txt

The CMakeLists.txt file would contain two lines:

```
PROJECT (Hello)
ADD_EXECUTABLE(Hello Hello.c)
```

To build the Hello executable you just follow the process described in **Running CMake** above to generate the makefiles or Microsoft project files. The PROJECT command indicates what the name of the resulting workspace should be and the ADD_EXECUTABLE command adds an executable target to the build process. That's all there is to it for this simple example. If your project requires a few files it is also quite easy, just modify the ADD_EXECUTABLE line as shown below.

```
ADD_EXECUTABLE(Hello Hello.c File2.c File3.c File4.c)
```

ADD_EXECUTABLE is just one of many commands available in CMake. Consider the more complicated example below.

```
PROJECT (HELLO)
SET(HELLO_SRCS Hello.c File2.c File3.c)
IF (WIN32)
  SET(HELLO_SRCS ${HELLO_SRCS} WinSupport.c)
ELSE (WIN32)
  SET(HELLO_SRCS ${HELLO_SRCS} UnixSupport.c)
ENDIF (WIN32)
ADD_EXECUTABLE (Hello ${HELLO_SRCS})

# look for the Tcl library
FIND_LIBRARY(TCL_LIBRARY NAMES tcl tcl84 tcl83 tcl82 tcl80
  PATHS  /usr/lib /usr/local/lib)
IF (TCL_LIBRARY)
  TARGET_ADD_LIBRARY (Hello TCL_LIBRARY)
ENDIF (TCL_LIBRARY)
```

In this example the SET command is used to group together source files into a list. The IF command is used to add either WinSupport.c or UnixSupport.c to this list. And finally the ADD_EXECUTABLE command is used to build the executable with the files listed in the source list HELLO_SRCS. The FIND_LIBRARY command looks for the Tcl library under a few different names and in a few different paths, and if it is found adds it to the link line for the Hello executable target. Note the use of the # character to denote a comment line.

CMake always defines some variables for use within CMakeList files. For example, WIN32 is always defined on windows systems and UNIX is always defined for UNIX systems. CMake defines a number of commands. A brief summary of the most commonly used commands

follows here. Later in the document an exhaustive list of all pre-defined commands is presented. (You may also add your own commands, see the Extension Guide for more information.)

A) Build Targets:

SET()
SUBDIRS()
ADD_LIBRARY()
ADD_EXECUTABLE()
AUX_SOURCE_DIRECTORY()
PROJECT()

CMake works recursively, descending from the current directory into any subdirectories listed in the SUBDIRS command. The command SET is used for setting a variable, in this case to a list of source files. (Note: currently only C and C++ code can be compiled.) ADD_LIBRARY adds a library to the list of targets this makefile will produce. ADD_EXECUTABLE adds an executable to the list of targets this makefile will produce. (Note: source code is compiled first, then libraries are built, and then executables are created.) The AUX_SOURCE_DIRECTORY is a directory where other source code, not in this directory, whose object code is to be inserted into the current LIBRARY. All source files in the AUX_SOURCE_DIRECTORY are compiled (e.g. *.c, *.cxx, *.cpp, etc.). PROJECT (PojectName) is a special variable used in the MSVC to create the project for the compiler, it also defines two useful variables for CMAKE: ProjectName_SOURCE_DIR and ProjectName_BINARY_DIR.

B) Build flags and options. In addition to the commands listed above, CMakeLists.txt often contain the following commands:

INCLUDE_DIRECTORIES()
LINK_DIRECTORIES()
LINK_LIBRARIES()
TARGET_LINK_LIBRARIES()

These commands define directories and libraries used to compile source code and build executables. An important feature of the commands listed above is that are inherited by any subdirectories. That is, as CMake descends through a directory hierarchy (defined by SUBDIRS()) these commands are expanded each time a definition for a command is encountered. For example, if in the top-level CMakeLists file has INCLUDE_DIRECTORIES(/usr/include), with SUBDIRS(./subdir1), and the file ./subdir1/CMakeLists.txt has INCLUDE_DIRECTORIES(/tmp/foobar), then the net result is

INCLUDE_DIRECTORIES(/usr/include /tmp/foobar)

C) CMake comes with a number of modules that look for commonly used packages such as OpenGL or Java. These modules save you from having to write all the CMake code to

find these packages yourself. Modules can be used by including them into your CMakeList file as shown below.

INCLUDE (${CMAKE_ROOT}/Modules/FindTCL.cmake)

CMAKE_ROOT is always defined in CMake and can be used to point to where CMake was installed. Looking through some of the files in the Modules subdirectory can provide good ideas on how to use some of the CMake commands.

Adding A New Directory to a project
A common way to extend a project is to add a new directory. This involves three steps:

1. Create the new directory somewhere in your source directory hierarchy.

2. Add the new directory to the SUBDIRS command in the parent directories CMakeLists.txt

3. Create a CMakeLists.txt in the new directory with the appropriate commands

CMake Commands

The following is an exhaustive list of pre-defined CMake commands, with brief descriptions.

- **ADD_CUSTOM_COMMAND** - Create new command within CMake.
  Usage: ADD_CUSTOM_COMMAND([SOURCE source] COMMAND command TARGET target [ARGS [args...]] [DEPENDS [depends...]] [OUTPUTS [outputs...]]) Add a custom command.

- **ADD_CUSTOM_TARGET** - Add an extra target to the build system that does not produce output, so it is run each time the target is built.
  Usage: ADD_CUSTOM_TARGET(Name [ALL] command arg arg arg ...) The ALL option is optional. If it is specified it indicates that this target should be added to the Build all target.

- **ADD_DEFINITIONS** - Add -D define flags to command line for environments.
  Usage: ADD_DEFINITIONS(-DFOO -DBAR ...) Add -D define flags to command line for environments.

- **ADD_DEPENDENCIES** - Add an dependency to a target
  Usage: ADD_DEPENDENCIES(target-name depend-target depend-target) Add a dependency to a target. This is only used to add dependencies between one executable and another. Regular build dependencies are handled automatically.

- **ADD_EXECUTABLE** - Add an executable to the project that uses the specified srclists
  Usage: ADD_EXECUTABLE(exename srclist srclist srclist ...)

ADD_EXECUTABLE(exename WIN32 srclist srclist srclist ...)This command adds an executable target to the current directory. The executable will be built from the source files / source lists specified. The second argument to this command can be WIN32 which indicates that the executable (when compiled on windows) is a windows app (using WinMain)not a console app (using main).

- **ADD_LIBRARY** - Add an library to the project that uses the specified srclists
  Usage: ADD_LIBRARY(libname [SHARED | STATIC | MODULE] srclist srclist ...) Adds a library target. SHARED, STATIC or MODULE keywords are used to set the library type. If the keywork MODULE appears, the library type is set to MH_BUNDLE on systems which use dyld. Systems without dyld MODULE is treated like SHARED. If no keywords appear as the second argument, the type defaults to the current value of BUILD_SHARED_LIBS. If this variable is not set, the type defaults to STATIC.

- **ADD_TEST** - Add a test to the project with the specified arguments.
  Usage: ADD_TEST(testname exename arg1 arg2 arg3 ...) If the ENABLE_TESTING command has been run, this command adds atest target to the current directory. If ENABLE_TESTING has notbeen run, this command does nothing. The tests are run by the testing subsystem by executing exename with the specified arguments. exename can be either an executable built by built by this project or an arbitrary executable on the system (like tclsh).

- **AUX_SOURCE_DIRECTORY** - Add all the source files found in the specified directory to the build as source list NAME.
  Usage: AUX_SOURCE_DIRECTORY(dir srcListName)

- **BUILD_COMMAND** - Determine the command line that will build this project.
  Usage: BUILD_COMMAND(NAME MAKECOMMAND) Within CMAKE set NAME to the command that will build this project from the command line using MAKECOMMAND.

- **BUILD_NAME** - Set a CMAKE variable to the build type.
  Usage: BUILD_NAME(NAME) Within CMAKE sets NAME to the build type.

- **CMAKE_MINIMUM_REQUIRED** - Set the minimum required version of cmake for a project.
  Usage: CMAKE_MINIMUM_REQUIRED(VERSION versionNumber) Let cmake know that the project requires a certain version of a cmake, or newer. CMake will also try to backwards compatible to the version of cmake specified, if a newer version of cmake is running.

- **CONFIGURE_FILE** - Create a file from an autoconf style file.in file.
  Usage: CONFIGURE_FILE(InputFile OutputFile [COPYONLY] [ESCAPE_QUOTES] [IMMEDIATE] [@ONLY]) The Input and Ouput files have to have full paths. They can also use variables like CMAKE_BINARY_DIR,CMAKE_SOURCE_DIR. This command replaces any variables in the input file with their values as determined by CMake. If a variables in not defined, it will be replaced with nothing. If COPYONLY is passed in, then

then no varible expansion will take place. If ESCAPE_QUOTES is passed in then any substitued quotes will be C style escaped. If IMMEDIATE is specified, then the file will be configured with the current values of CMake variables instead of waiting until the end of CMakeLists processing. If @ONLY is present, only variables of the form @var@ will be replaces and ${var} will be ignored. This is useful for configuring tcl scripts that use ${var}.

- **CREATE_TEST_SOURCELIST** - Create a test driver and source list for building test programs.
  Usage: CREATE_TEST_SOURCELIST(SourceListName DriverName test1 test2 test3 EXTRA_INCLUDE include.h FUNCTION function)The list of source files needed to build the testdriver will be in SourceListName. DriverName.cxx is the name of the test driver program. The rest of the arguments consist of a list of test source files, can be ; separated. Each test source file should have a function in it that is the same name as the file with no extension (foo.cxx should have int foo();) DriverName.cxx will be able to call each of the tests by name on the command line. If EXTRA_INCLUDE is specified, then the next argument is included into the generated file. If FUNCTION is specified, then the next argument is taken as a function name that is passed a pointer to ac and av. This can be used to add extra command line processing to each test.

- **ELSE** - starts the else portion of an if block
  Usage: ELSE(args), Note that the args for the ELSE clause must match those of the IF clause. See the IF command for more information.

- **ENABLE_TESTING** - Enable testing for this directory and below.
  Usage: ENABLE_TESTING() Enables testing for this directory and below. See also the ADD_TEST command. Note that Dart expects to find this file in the build directory root; therefore, this command should be in the source directory root too.

- **ENDFOREACH** - ends a foreach block
  Usage: ENDFOREACH(define)

- **ENDIF** - ends an if block
  Usage: ENDIF(define)

- **EXEC_PROGRAM** - Run and executable program during the processing of the CMakeList.txt file.
  Usage: EXEC_PROGRAM(Executable [Directory to run in] [ARGS arguments to executable])The executable is run in the optionally specified Directory. The executable can include arguments if it is double quoted, but it is better to use the optional ARGS argument to specify arguments to the program. This is because cmake will then be able to escape spaces in the Executable path.

- **FIND_FILE** - Find a file.
  Usage: FIND_FILE(NAME file extrapath extrapath ... [DOC docstring])Find a file in the system PATH or in any extra paths specified in the command.A cache entry called NAME is created to store the result. NOTFOUND is the value used if the file was not

found. If DOC is specified the next argument is the documentation string for the cache entry NAME.

- **FIND_LIBRARY** - Find a library.
  Usage: FIND_LIBRARY(DEFINE_PATH libraryName [NAMES] name1 name2 name3 [PATHS path1 path2 path3...] [DOC docstring]) If the library is found, then DEFINE_PATH is set to the full path where it was found. If DOC is specified the next argument is the documentation string for the cache entry NAME.

- **FIND_PATH** - Find a path for a file.
  Usage: FIND_PATH(PATH_DEFINE fileName path1 path2 path3...) If the file is found, then PATH_DEFINE is set to the path where it was found.If DOC is specified the next argument is the documentation string for the cache entry NAME.

- **FIND_PROGRAM** - Find an executable program.
  Usage: FIND_PROGRAM(NAME executable1 extrapath extrapath ... [DOC helpstring]) Find the executable in the system PATH or in any extra paths specified in the command.A cache entry called NAME is created to store the result. NOTFOUND is the value used if the program was not found. If DOC is specified the next argument is the documentation string for the cache entry NAME.

- **FLTK_WRAP_UI** - Create FLTK user interfaces Wrappers.
  Usage: FLTK_WRAP_UI(resultingLibraryName SourceList) Produce .h and .cxx files for all the .fl and .fld file listed in the SourceList. The .h files will be added to the library using the base name in source list. The .cxx files will be added to the library using the base name in source list.

- **FOREACH** - start a foreach loop
  Usage: FOREACH (define arg1 arg2 arg2) Starts a foreach block.

- **GET_FILENAME_COMPONENT** - Get a specific component of a full filename.
  Usage: GET_FILENAME_COMPONENT(VarName FileName PATH|NAME|EXT|NAME_WE [CACHE]) Set VarName to be the path (PATH), file name (NAME), file extension (EXT) or file name without extension (NAME_WE) of FileName. Note that the path is converted to Unix slashes format and has no trailing slashes. The longest file extension is always considered. Warning: as a utility command, the resulting value is not put in the cache but in the definition list, unless you add the optional CACHE parameter.

- **GET_SOURCE_FILE_PROPERTY** - Set attributes for a specific list of files.
  Usage: GET_SOURCE_FILE_PROPERTY(VAR file [ABSTRACT|WRAP_EXCLUDE|COMPILE_FLAGS]) Get a property from a source file. The value of the property is stored in the variable VAR.

- **IF** - start an if block
  Usage: IF (define) Starts an if block. Optionally it can be invoked using (NOT define) (def AND def2) (def OR def2) (def MATCHES def2) (COMMAND cmd) (EXISTS file)

MATCHES checks if def matches the regular expression def2. COMMAND checks if the cmake command cmd is in this cmake executable. EXISTS file checks if file exists

- **INCLUDE** - Basically identical to a C #include "somthing" command.
  Usage: INCLUDE(file1 [OPTIONAL]) If OPTIONAL is present, then do not complain if the file does not exist.

- **INCLUDE_DIRECTORIES** - Add include directories to the build.
  Usage: INCLUDE_DIRECTORIES([BEFORE] dir1 dir2 ...)

- **INCLUDE_EXTERNAL_MSPROJECT** - Include an external Microsoft project file in a workspace.
  Usage: INCLUDE_EXTERNAL_MSPROJECT(projectname location dep1 dep2 ...) Includes an external Microsoft project in the workspace file. Does nothing on UNIX currently

- **INCLUDE_REGULAR_EXPRESSION** - Set the regular expression used for dependency checking.
  Usage: INCLUDE_REGULAR_EXPRESSION(regex_match [regex_complain]) Set the regular expressions used in dependency checking. Only files matching regex_match will be traced as dependencies. Only files matching regex_complain will generate warnings if they cannot be found (standard header paths are not searched). The defaults are: regex_match = "^.*$" (match everything) regex_complain = "^$" (match empty string only)

- **INSTALL_FILES** - Create install rules for files
  Usage: INSTALL_FILES(path extension srclist file file srclist ...) INSTALL_FILES(path regexp) Create rules to install the listed files into the path. Path is relative to the variable CMAKE_INSTALL_PREFIX. There are two forms for this command. In the first the files can be specified explicitly or by referenceing source lists. All files must either have the extension specified or exist with the extension appended. A typical extension is .h etc... In the second form any files in the current directory that match the regular expression will be installed.

- **INSTALL_PROGRAMS** - Create install rules for programs
  Usage: INSTALL_PROGRAMS(path file file ...) INSTALL_PROGRAMS(path regexp) Create rules to install the listed programs into the path. Path is relative to the variable CMAKE_INSTALL_PREFIX. There are two forms for this command. In the first the programs can be specified explicitly. In the second form any program in the current directory that match the regular expression will be installed.

- **INSTALL_TARGETS** - Create install rules for targets
  Usage: INSTALL_TARGETS(path target target) Create rules to install the listed targets into the path. Path is relative to the variable PREFIX

- **LINK_DIRECTORIES** - Specify link directories.
  Usage: LINK_DIRECTORIES(directory1 directory2 ...) Specify the paths to the libraries

that will be linked in. The directories can use built in definitions like CMAKE_BINARY_DIR and CMAKE_SOURCE_DIR.

- **LINK_LIBRARIES** - Specify a list of libraries to be linked into executables or shared objects.
  Usage: LINK_LIBRARIES(library1 library2 ...) Specify a list of libraries to be linked into executables or shared objects. This command is passed down to all other commands. The debug and optimized strings may be used to indicate that the next library listed is to be used only for that specific type of build

- **LOAD_CACHE** - load in the values from another cache.
  Usage: LOAD_CACHE(pathToCacheFile [EXCLUDE entry1...] [INCLUDE_INTERNALS entry1...]) Load in the values from another cache. This is useful for a project that depends on another project built in a different tree.EXCLUDE option can be used to provide a list of entries to be excluded.INCLUDE_INTERNALS can be used to provide a list of internal entriesto be included. Normally, no internal entries are brougt in.

- **MAKE_DIRECTORY** - Create a directory in the build tree if it does not exist. Parent directories will be created if the do not exist..
  Usage: MAKE_DIRECTORY(directory)

- **MARK_AS_ADVANCED** - Mark a cmake varible as advanced.
  Usage: MARK_AS_ADVANCED([CLEAR|FORCE]VAR VAR2 VAR...) Mark the named variables as advanced. An advanced variable will not be displayed in any of the cmake GUIs, unless the show advanced option is on. If CLEAR is the first argument advanced variables are changed back to unadvanced.If FORCE is the first arguement, then the variable is made advanced.If neither FORCE or CLEAR is specified, new values will be marked as advanced, but if the variable already has an advanced state, it will not be changed.

- **MESSAGE** - Display a message to the user.
  Usage: MESSAGE([SEND_ERROR] "message to display"...) The arguments are messages to display. If the first argument is SEND_ERROR then an error is raised.

- **OPTION** - Provides an option that the user can optionally select
  Usage: OPTION(USE_MPI "help string decribing the option" [initial value]) Provide an option for the user to select

- **OUTPUT_REQUIRED_FILES** - Output a list of required source files for a specified source file.
  Usage: OUTPUT_REQUIRED_FILES(srcfile outputfile) Outputs a list of all the source files that are required by the specified srcfile. This list is written into outputfile. This is similar to writing out the dependencies for srcfile except that it jumps from .h files into .cxx, .c and .cpp files if possible.

- **PROJECT** - Set a name for the entire project. One argument.
  Usage: PROJECT(projectname [C++ C Java]) Sets the name of the project. This creates the variables projectname_BINARY_DIR and projectname_SOURCE_DIR. Optionally you can specify which languages your project supports. By default all languages are supported. If you do not have a C++ compiler, but want to build a c program with cmake, then use this option.

- **QT_WRAP_CPP** - Create QT Wrappers.
  Usage: QT_WRAP_CPP(resultingLibraryName DestName SourceLists ...) Produce moc files for all the .h file listed in the SourceLists. The moc files will be added to the library using the DestName source list.

- **QT_WRAP_UI** - Create QT user interfaces Wrappers.
  Usage: QT_WRAP_UI(resultingLibraryName HeadersDestName SourcesDestName SourceLists ...) Produce .h and .cxx files for all the .ui file listed in the SourceLists. The .h files will be added to the library using the HeadersDestName source list. The .cxx files will be added to the library using the SourcesDestName source list.

- **REMOVE** - Remove a value from a CMAKE variable
  Usage: REMOVE(VAR VALUE VALUE ...) Removes VALUE from the CMAke variable VAR. This is typically used to remove entries from a vector (e.g. semicolon separated list). VALUE is expanded.

- **SET** - Set a CMAKE variable to a value
  Usage: SET(VAR [VALUE] [CACHE TYPE DOCSTRING]) Within CMAKE sets VAR to the value VALUE. VALUE is expanded before VAR is set to it. If CACHE is present, then the VAR is put in the cache. TYPE and DOCSTRING are required. If TYPE is INTERNAL, then the VALUE is Always written into the cache, replacing any values existing in the cache. If it is not a CACHE VAR, then this always writes into the current makefile. An optional syntax is SET(VAR VALUE1 ... VALUEN). In this case VAR is set to a ; separated list of values.

- **SET_SOURCE_FILES_PROPERTIES** - Set attributes for a specific list of files.
  Usage: SET_SOURCE_FILES_PROPERTIES(file1 file2 .. filen [ABSTRACT|WRAP_EXCLUDE|GENERATED|COMPILE_FLAGS] [flags]) Set properties on a file. The syntax for the command is to list all the files you want to change, and then provide the values you want to set next.

- **SITE_NAME** - Set a CMAKE variable to the name of this computer.
  Usage: SITE_NAME(NAME) Within CMAKE sets NAME to the host name of the computer.

- **SOURCE_GROUP** - Define a grouping for sources in the makefile.
  Usage: SOURCE_GROUP(name regex) Defines a new source group. Any file whose name matches the regular expression will be placed in this group. The LAST regular expression of all defined SOURCE_GROUPs that matches the file will be selected.

- **SUBDIRS** - Add a list of subdirectories to the build.
  Usage: SUBDIRS(dir1 dir2 ...) Add a list of subdirectories to the build. This will cause any CMakeLists.txt files in the sub directories to be processed by CMake.

- **SUBDIR_DEPENDS** - Legacy command. Does nothing.
  Usage: SUBDIR_DEPENDS(subdir dep1 dep2 ...) Does not do anything. This command used to help projects order parallel builds correctly. This functionality is now automatic.

- **TARGET_LINK_LIBRARIES** - Specify a list of libraries to be linked into executables or shared objects.
  Usage: TARGET_LINK_LIBRARIES(target library1 library2 ...) Specify a list of libraries to be linked into the specified target The debug and optimized strings may be used to indicate that the next library listed is to be used only for that specific type of build

- **USE_MANGLED_MESA** - Create copies of mesa headers for use in combination with system gl.
  Usage: USE_MANGLED_MESA(PATH_TO_MESA OUTPUT_DIRECTORY) The path to mesa includes, should contain gl_mangle.h.

- **UTILITY_SOURCE** - Specify the source tree of a third-party utility.
  Usage: UTILITY_SOURCE(cache_entry executable_name path_to_source [file1 file2 ...]) When a third-party utility's source is included in the distribution, this command specifies its location and name. The cache entry will not be set unless the path_to_source and all listed files exist. It is assumed that the source tree of the utility will have been built before it is needed.

- **VARIABLE_REQUIRES** - Display an error message .
  Usage: VARIABLE_REQUIRES(TEST_VARIABLE RESULT_VARIABLE REQUIRED_VARIABLE1 REQUIRED_VARIABLE2 ...) The first argument (TEST_VARIABLE) is the name of the varible to be tested, if that varible is false nothing else is done. If TEST_VARIABLE is true, then the next arguemnt (RESULT_VARIABLE) is a vairable that is set to true if all the required variables are set.The rest of the arguments are varibles that must be true or not set to NOTFOUND to avoid an error.

- **VTK_MAKE_INSTANTIATOR** - Register classes for creation by vtkInstantiator
  Usage: VTK_MAKE_INSTANTIATOR(className outSourceList src-list1 [src-list2 ..] EXPORT_MACRO exportMacro [HEADER_LOCATION dir] [GROUP_SIZE groupSize] [INCLUDES [file1 file2 ..]]) Generates a new class with the given name and adds its files to the given outSourceList. It registers the classes from the other given source lists with vtkInstantiator when it is loaded. The output source list should be added to the library with the classes it registers. The EXPORT_MACRO argument must be given and followed by the export macro to use when generating the class (ex. VTK_COMMON_EXPORT). The HEADER_LOCATION option must be followed by a path. It specifies the directory in which to place the generated class's header file. The generated class implementation files always go in the build directory corresponding to the

CMakeLists.txt file containing the command. This is the default location for the header. The GROUP_SIZE option must be followed by a positive integer. As an implementation detail, the registered creation functions may be split up into multiple files. The groupSize option specifies the number of classes per file. Its default is 10. The INCLUDES option can be followed by a list of zero or more files. These files will be #included by the generated instantiator header, and can be used to gain access to the specified exportMacro in the C++ code.

- **VTK_WRAP_JAVA** - Create Java Wrappers.
  Usage: VTK_WRAP_JAVA(resultingLibraryName SourceListName SourceLists ...)
- **VTK_WRAP_PYTHON** - Create Python Wrappers.
  Usage: VTK_WRAP_PYTHON(resultingLibraryName SourceListName SourceLists ...)
- **VTK_WRAP_TCL** - Create Tcl Wrappers for VTK classes.
  Usage: VTK_WRAP_TCL(resultingLibraryName [SOURCES] SourceListName SourceLists ... [COMMANDS CommandName1 CommandName2 ...])

Extending CMake Guide

This section describes some of the internals of CMake. Read this section only if you intend to add new commands to the CMake executable or debug CMake. First you must download and install the source code for CMake as described in the **Installing CMake** section.

Adding a New Command

Commands can be added to CMake by deriving new commands from the class cmCommand (defined in CMake/Source/cmCommand.h/.cxx). Typically each command is implemented in a class called cmCommandNameCommand stored in cmCommandNameCommand.h and cmCommandNameCommand.cxx. If you want to create a rule the best bet is to take a look at some of the existing rules in CMake. They tend to be fairly short.

Adding a New Makefile Generator

From a conceptual point, adding a new generator is simple. You derive a class from cmMakefileGenerator, and override GenerateMakefile() and EnableLanguage(). The GenerateMakefile method can become quite complex. Its job is to translate all the internal values in the cmMakfile class into a build file. The developer must know how to create shared and static libraries, and executables. If you are interested in adding a new build type to cmake, please feel free to contact the cmake users list, and you most likely will find assistance for cmake developers.

Further Information

Much of the development of CMake was performed at Kitware http://www.kitware.com/. The developers can be reached at mailto:kitware@kitware.com. CMake was initially developed for the

NIH/NLM Insight Segmentation and Registration Toolkit, see the Web site at http://www.itk.org/Insight.html. CMake's web page can be found at http://www.cmake.org.