Contents

CHAPTER 1 WHY CMAKE? 1

1.1 The History of CMake 3
1.2 Why Not Use Autoconf? 3
1.3 Why Not Use JAM, qmake, SCons, or ANT? 4
1.4 Why Not Script It Yourself? 4
1.5 On What Platforms Does CMake Run? 5

CHAPTER 2 GETTING STARTED 7

2.1 Getting and Installing CMake on Your Computer 7
UNIX and Mac Binary Installations *7*
Windows Binary Installation *7*
2.2 Building CMake Yourself 8
2.3 Basic CMake Usage and Syntax 8
2.4 Hello World for CMake 9
2.5 How to Run CMake? 10
Running CMake's Qt Interface *11*
Running the ccmake Curses Interface *13*
Running CMake from the Command Line *15*
Specifying the Compiler to CMake *15*
Dependency Analysis *16*
2.6 Editing CMakeLists Files 17
2.7 Setting Initial Values for CMake 17
2.8 Building Your Project 19

CHAPTER 3 KEY CONCEPTS 21

3.1 Main Structures 21
3.2 Targets 24
3.3 Source Files 25
3.4 Directories, Generators, Tests, and Properties 26
3.5 Variables and Cache Entries 27
3.6 Build Configurations 32

CHAPTER 4 WRITING CMAKELISTS FILES 33

4.1 CMake Syntax 33
4.2 Basic Commands 34
4.3 Flow Control 35
4.4 Regular Expressions 42

4.5	Checking Versions of CMake	44
4.6	Using Modules	45
	Using CMake with SWIG	*48*
	Using CMake with Qt	*49*
	Using CMake with FLTK	*50*
4.7	Policies	50
	Updating a Project For a New Version of CMake	*53*
4.8	Linking Libraries	57
	Specifying Optimized or Debug Libraries with a Target	*59*
4.9	Shared Libraries and Loadable Modules	59
4.10	Shared Library Versioning	64
4.11	Installing Files	66
	Installing Prerequisite Shared Libraries	*76*
4.12	Advanced Commands	82

CHAPTER 5 SYSTEM INSPECTION 85

5.1	Using Header Files and Libraries	85
5.2	System Properties	87
5.3	Finding Packages	92
5.4	Built-in Find Modules	93
5.5	How to Pass Parameters to a Compilation?	95
5.6	How to Configure a Header File	97
5.7	Creating CMake Package Configuration Files	99

CHAPTER 6 CUSTOM COMMANDS AND TARGETS 103

6.1	Portable Custom Commands	103
6.2	Using add_custom_command on a Target	105
	How to Copy an Executable Once it is Built?	*106*
6.3	Using add_custom_command to Generate a File	107
	Using an Executable to Build a Source File	*107*
6.4	Adding a Custom Target	108
6.5	Specifying Dependencies and Outputs	111
6.6	When There Isn't One Rule For One Output	112
	A Single Command Producing Multiple Outputs	*112*
	Having One Output That Can Be Generated By Different Commands	*112*

CHAPTER 7 CONVERTING EXISTING SYSTEMS TO CMAKE 115

7.1	Source Code Directory Structures	115
7.2	Build Directories	117
7.3	Useful CMake Commands When Converting Projects	119
7.4	Converting UNIX Makefiles	120

7.5 Converting Autoconf Based Projects 121
7.6 Converting Windows Based Workspaces 123

CHAPTER 8 CROSS COMPILING WITH CMAKE 125

8.1 Toolchain Files 126
Finding External Libraries, Programs and Other Files *128*
8.2 System Inspection 130
Using Compile Checks *131*
8.3 Running Executables Built in the Project 133
8.4 Cross Compiling Hello World 136
8.5 Cross Compiling for a Microcontroller 140
8.6 Cross Compiling an Existing Project 143
8.7 Cross Compiling a Complex Project - VTK 145
8.8 Some Tips and Tricks 147

CHAPTER 9 PACKAGING WITH CPACK 149

9.1 CPack Basics 149
Simple Example *150*
What Happens When CPack.cmake Is Included? *151*
Adding Custom CPack Options *152*
Options Added by CPack *153*
9.2 CPack Source Packages 154
9.3 CPack Installer Commands 154
9.4 CPack for Windows Installer NSIS 156
CPack Variables Used by CMake for NSIS *156*
Creating Windows Short Cuts in the Start Menu *161*
Advanced NSIS CPack Options *161*
Setting File Extension Associations With NSIS *162*
Installing Microsoft Run Time Libraries *163*
CPack Component Install Support *163*
9.5 CPack for Cygwin Setup 173
9.6 CPack for Mac OS X PackageMaker 176
9.7 CPack for Mac OS X Drag and Drop 178
9.8 CPack for Mac OS X X11 Applications 180
9.9 CPack for Debian Packages 182
9.10 CPack for RPM 183
9.11 CPack Files 183

CHAPTER 10 AUTOMATION & TESTING WITH CMAKE 185

10.1 Testing with CMake, CTest, and CDash 185
10.2 How Does CMake Facilitate Testing? 186

10.3 Additional Test Properties 187
10.4 Testing Using CTest 189
10.5 Using CTest to Drive Complex Tests 191
10.6 Handling a Large Number of Tests 192
10.7 Producing Test Dashboards 194
Adding CDash Dashboard Support to a Project *196*
Client Setup *199*
10.8 Customizing Dashboards for a Project 202
Dashboard Submissions Settings *202*
Filtering Errors and Warnings *203*
Adding Notes to a Dashboard *205*
10.9 Setting up Automated Dashboard Clients 206
Settings for Continuous Dashboards *210*
Variables Available in CTest Scripts *212*
10.10 Advanced CTest Scripting 212
Limitations of Traditional CTest Scripting *213*
Extended CTest Scripting *213*
10.11 Setting up a Dashboard Server 218
CDash Server *218*
Advanced Server Management *220*
Build Groups *223*
Email *225*
Sites *226*
Graphs *227*
Adding Notes to a Build *228*
Logging *229*
Test Timing *229*
Mobile Support *230*
Backing up CDash *230*
Upgrading CDash *231*
CDash Maintenance *232*
10.12 Subprojects 233
Using ctest_submit with PARTS and FILES *236*
Splitting Your Project into Multiple Subprojects *237*

CHAPTER 11 PORTING CMAKE TO NEW PLATFORMS AND LANGUAGES241

11.1 The Determine System Process 241
11.2 The Enable Language Process 242
11.3 Porting to a New Platform 244
11.4 Adding a New Language 246
11.5 Rule Variable Listing 247
General Tag Variables *247*
Language Specific Information *248*

11.6 C
C
E
11.7 E
C
U

CHAPTER 12

12.1 A
A
12.2 A
12.3 I
12.4 A
12.5 A
12.6 B
12.7 A

APPENDIX A

Variables Th
Variables Th
Variables fo
Variables Th
Variables Th

APPENDIX B

CMake Com
CMake Gene
CTest Comm
CPack Comm
CPack Gener

APPENDIX C

Current Com
Compatibilit

APPENDIX D

CMake Modu

187
189
191
192
194
196
199
202
202
203
205
206
210
212
212
213
213
218
218
220
223
225
226
227
228
229
229
230
230
231
232
233
236
237

GUAGES241

241
242
244
246
247
247
248

11.6 Compiler and Platform Examples 248
Como Compiler *248*
Borland Compiler *249*
11.7 Extending CMake 250
Creating a Loaded Command *250*
Using a Loaded Command *251*

CHAPTER 12 TUTORIALS 255

12.1 A Basic Starting Point (Step 1) 255
Adding a Version Number and Configured Header File *256*
12.2 Adding a Library (Step 2) 258
12.3 Installing and Testing (Step 3) 260
12.4 Adding System Introspection (Step 4) 262
12.5 Adding a Generated File and Generator (Step 5) 263
12.6 Building an Installer (Step 6) 267
12.7 Adding Support for a Dashboard (Step 7) 268

APPENDIX A - VARIABLES 269

Variables That Change Behavior 269
Variables That Describe the System 272
Variables for Languages 274
Variables That Control the Build 278
Variables That Provide Information 280

APPENDIX B – COMMAND LINE REFERENCE 287

CMake Command Line Options 287
CMake Generators 292
CTest Command Line Options 294
CPack Command Line Options 298
CPack Generators 299

APPENDIX C – LISTFILE COMMANDS 301

Current Commands 301
Compatibility Commands 366

APPENDIX D – SELECTED MODULES 373

CMake Modules 373

APPENDIX E - PROPERTIES **411**

Properties of Global Scope 411
Properties on Directories 414
Properties on Targets 417
Properties on Tests 431
Properties on Source Files 431
Properties on Cache Entries 434

APPENDIX F – CMAKE POLICIES **437**

INDEX **447**

Why CM

If you have ever m
be interested in C
allows developers
then used by CM
Development Env
UNIX, Linux, NM
building software
builds, in a simple
software systems.

For any project, a
system. Many pr
Microsoft Visual S
build systems up t
such as Borland o
bigger problem. Th
as including JPEG
consolidating these

If you have multip
software will have
software and custo
are that two compu
benefits for single

411

411
414
417
431
431
434

437

447

Chapter 1

Why CMake?

If you have ever maintained the build and installation process for a software package, you will be interested in CMake. CMake is an open source build manager for software projects that allows developers to specify build parameters in a simple portable text file format. This file is then used by CMake to generate project files for native build tools including Integrated Development Environments such as Microsoft Visual Studio or Apple's Xcode, as well as UNIX, Linux, NMake, and Borland style Makefiles. CMake handles the difficult aspects of building software such as cross platform builds, system introspection, and user customized builds, in a simple manner that allows users to easily tailor builds for complex hardware and software systems.

For any project, and especially cross platform projects, there is a need for a unified build system. Many projects today ship with both a UNIX Makefile (or Makefile.in) and a Microsoft Visual Studio workspace. This requires that developers constantly try to keep both build systems up to date and consistent with each other. To target additional build systems such as Borland or Xcode requires even more custom copies of these files, creating an even bigger problem. This problem is compounded if you try to support optional components, such as including JPEG support if libjpeg is available on the system. CMake solves this by consolidating these different operations into one simple easy to understand file format.

If you have multiple developers working on a project, or multiple target platforms, then the software will have to be built on more than one computer. Given the wide range of installed software and custom options that are involved with setting up a modern computer, the chances are that two computers running the same OS will be slightly different. CMake provides many benefits for single platform multi-machine development environments including:

- The ability to automatically search for programs, libraries, and header files that may be required by the software being built. This includes the ability to consider environment variables and Window's registry settings when searching.
- The ability to build in a directory tree outside of the source tree. This is a useful feature found on many UNIX platforms; CMake provides this feature on Windows as well. This allows a developer to remove an entire build directory without fear of removing source files.
- The ability to create complex custom commands for automatically generated files such as Qt's moc (qt.nokia.com), The Insight Toolkit's CABLE wrappers (public.kitware.com/Cable/HTML/Index.html) and SWIG (www.swig.org) wrapper generators. These commands are used to generate new source files during the build process that are in turn compiled into the software.
- The ability to select optional components at configuration time. For example, several of VTK's libraries are optional, and CMake provides an easy way for users to select which libraries are built.
- The ability to automatically generate workspaces and projects from a simple text file. This can be very handy for systems that have many programs or test cases, each of which requires a separate project file, typically a tedious manual process to create using an IDE.
- The ability to easily switch between static and shared builds. CMake knows how to create shared libraries and modules on all platforms supported. Complicated platform-specific linker flags are handled, and advanced features like built in run time search paths for shared libraries are supported on many UNIX systems.
- Automatic generation of file dependencies and support for parallel builds on most platforms.

When developing cross platform software, CMake provides a number of additional features:

- The ability to test for machine byte order and other hardware specific characteristics.
- A single set of build configuration files that work on all platforms. This avoids the problem of developers having to maintain the same information in several different formats inside a project.
- Support for building shared libraries on all platforms that support it.
- The ability to configure files with system dependent information such as the location of data files and other information. CMake can create header files that contain information such as paths to data files and other information in the form of #define macros. System specific flags can also be placed in configured header files. This has advantages over command line –D options to the compiler because it allows other build systems to use the CMake built library without having to specify the exact same command line options used during the build.

1.1 The History of CMake

CMake development began in 1999 as part of the Insight Toolkit (ITK, www.itk.org) funded by the US National Library of Medicine. ITK is a large software project that works on many platforms and can interact with many other software packages. To support this, a powerful, yet easy to use, build tool was required. Having worked with build systems for large projects in the past, the developers designed CMake to address these needs. Since then CMake has continuously grown in popularity, with many projects and developers adopting it for its ease of use and flexibility. Since 1999 CMake has been under active development and has matured to the point where it is a proven solution for a wide range of build issues. The most telling example of this is the successful adoption of CMake as the build system of the K Desktop Environment (KDE), arguably the largest open source software project in existence.

One of the recent additions to CMake is the inclusion of software testing support in the form of CTest. Part of the process of testing software involves building the software, possibly installing it, and determining what parts of the software are appropriate for the current system. This makes CTest a logical extension of CMake as it already has most of this information. In a similar vein, a new CMake feature is CPack, which is designed to support cross platform distribution of software. It provides a cross platform approach to creating native installations for your software, making use of existing popular packages such as NSIS, RPM, Cygwin, and PackageMaker.

Other recent additions to CMake include support for Apple's Xcode IDE and support for Microsoft's Visual Studio 10. With CMake, once you write your input files you get support for new compilers and build systems for free because the support for them is built into new releases of CMake, not tied to your software distribution. Another recent addition to CMake is support for cross compiling to other operating systems or embedded devices. Many commands in CMake now properly handle the differences between the host system and the target platform when cross compiling.

1.2 Why Not Use Autoconf?

Before developing CMake its authors had experience with the existing set of available tools. Autoconf combined with automake provides some of the same functionality as CMake, but to use these tools on a Windows platform requires the installation of many additional tools not found natively on a Windows box. In addition to requiring a host of tools, autoconf can be difficult to use or extend and impossible for some tasks that are easy in CMake. Even if you do get autoconf and its required environment running on your system, it generates Makefiles that will force users to the command line. CMake on the other hand provides a choice, allowing developers to generate project files that can be used directly from the IDE to which Windows and Xcode developers are accustomed.

While autoconf supports user specified options, it does not support dependent options where one option depends on some other property or selection. For example, in CMake you could have a user option to enable multithreading be dependent on first determining if the user's system has multithreading support. CMake provides an interactive user interface, making it easy for the user to see what options are available and how to set them.

For UNIX users, CMake also provides automated dependency generation that is not done directly by autoconf. CMake's simple input format is also easier to read and maintain than a combination of Makefile.in and configure.in files. The ability of CMake to remember and chain library dependency information has no equivalent in autoconf/automake.

1.3 Why Not Use JAM, qmake, SCons, or ANT?

Other tools such as ANT, qmake, SCons, and JAM have taken different approaches to solving these problems and they have helped us to shape CMake. Of the four, qmake, is the most similar to CMake although it lacks much of the system interrogation that CMake provides. Qmake's input format is more closely related to a traditional Makefile. ANT, JAM and SCons are also cross-platform although they do not support generating native project files. They do break away from the traditional Makefile oriented input with ANT using XML, JAM using its own language, and SCons using Python. A number of these tools run the compiler directly, as opposed to letting the system's build process perform that task. Many of these tools require other tools such as Python or Java to be installed before they will work.

1.4 Why Not Script It Yourself?

Some projects use existing scripting languages such as Perl or Python to configure build processes. Although similar functionality can be achieved with systems like this, over-use of tools can make the build process more of an Easter egg hunt than a simple-to-use build system. When building your software package users are forced to find and install version 4.3.2 of this, and 3.2.4 of that, before they can even start the build process. To avoid that problem, it was decided that CMake would require no more tools than the software it was being used to build would require. At a minimum using CMake requires a C compiler, that compiler's native build tools, and a CMake executable. CMake was written in C++, requires only a C++ compiler to build and precompiled binaries are available for most systems. Scripting it yourself also typically means you will not be generating native Xcode or Visual Studio workspaces, making Mac and Windows builds limited.

1.5 On What Platforms Does CMake Run?

CMake runs on a wide variety of platforms including Microsoft Windows, Apple Mac OS X, and most UNIX or UNIX-like platforms. At the time of the writing of this book CMake was tested nightly on the following platforms: Windows 98/2000/XP/Vista/7, AIX, HPUX, IRIX, Linux, Mac OS X, Solaris, OSF, QNX, CYGWIN, MinGW, and FreeBSD. You can check www.cmake.org for a current list of tested platforms.

Likewise, CMake supports most common compilers. It supports the GNU compiler on all CMake supported platforms. Other tested compilers include Visual Studio 6 through 10, Intel C, SGI CC, Mips Pro, Borland, Sun CC and HP aCC. CMake should work for most UNIX-style compilers out of the box. If the compiler takes arguments in a strange way, then see the section *Porting CMake to New Platform* on page 241 for information on how to customize CMake for a new compiler.

Chapter 2

Getting Started

2.1 Getting and Installing CMake on Your Computer

Before using CMake you will need to install or build the CMake binaries on your system. On many systems you may find that CMake is already installed, or is available for install with the standard package manager tool for the system. Cygwin, Debian, FreeBSD, Mac OS X Fink, and many others all have CMake distributions. If your system does not have a CMake package, you can find CMake precompiled for most common architectures at www.cmake.org. If you do not find binaries for your system precompiled, then you can build CMake from source. To build CMake you will need a modern C++ compiler.

UNIX and Mac Binary Installations

If your system provides CMake as one of its standard packages, follow your system's package installation instructions. If your system does not have CMake, or has an out of date version of CMake, you can download precompiled binaries from www.cmake.org. The binaries from www.cmake.org come in the form of a compressed tar file. The tar file contains a README file and an enclosed tar file. The README file contains a manifest of the files contained in the enclosed tar file, and some instructions. To install, simply extract the enclosed tar file into a destination directory (typically `/usr/local`). However, it can be any directory, and does not require root privileges for installation.

Windows Binary Installation

For Windows CMake has a NullSoft install file available for download from www.cmake.org. To install this file, simply run the executable on the windows machine on which you want to install CMake. You will be able to run CMake from the Start Menu after it is installed.

2.2 Building CMake Yourself

If binaries are not available for your system, or if binaries are not available for the version of CMake you wish to use, you can build CMake from the source code. You can obtain the CMake source code by following the instructions at www.cmake.org. Once you have the source code it can be built in two different ways. If you have a version of CMake on your system you can use it to build other versions of CMake. Generally the current development version of CMake can always be built from the previous release of CMake. This is how new versions of CMake are built on most Windows systems.

The second way to build CMake is by running its bootstrap build script. To do this you change directory into your CMake source directory and type

```
./bootstrap
make
make install
```

The make install step is optional since CMake can run directly from the build directory if desired. On UNIX, if you are not using the GNU C++ compiler, you need to tell the bootstrap script which compiler you want to use. This is done by setting the environment variable `CXX` before running bootstrap. If you need to use any special flags with your compiler, set the `CXXFLAGS` environment variable. For example, on the SGI with the 7.3X compiler, you would build CMake like this:

```
cd CMake
(setenv CXX CC; setenv CXXFLAGS "-LANG:std"; ./bootstrap)
make
make install
```

2.3 Basic CMake Usage and Syntax

Using CMake is simple. The build process is controlled by creating one or more CMakeLists files (actually CMakeLists.txt but this guide will leave off the extension in most cases) in each of the directories that make up a project. The CMakeLists files should contain the project description in CMake's simple language. The language is expressed as a series of commands. Each command is evaluated in the order that it appears in the CMakeLists file. The commands have the form

```
command (args...)
```