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.