

**CS 5301: Software Engineering Tools Lab  
Report**

# **CMakeQt: Cross-Platform Build System Automation**

Submitted by

Rohit Yadav (07020003)  
IDD Part V

DEPARTMENT OF COMPUTER ENGINEERING  
IT-BHU, VARANASI-221005  
NOVEMBER 2011



**DEPARTMENT OF COMPUTER ENGINEERING  
INSTITUTE OF TECHNOLOGY  
BANARAS HINDU UNIVERSITY  
VARANASI-221005, (U.P.) INDIA**

## **CERTIFICATE**

This is to certify that Rohit Yadav (07020003) and Rahul Jain (07020007), IDD Part V, Department of Computer Engineering, Institute of Technology, Banaras Hindu University, have worked on their project report for Software Engineering Tools Lab (CS 5301) entitled “CMakeQt: Cross-Platform Build System Automation” under my direct supervision during the period July 2011 – November 2011, the findings of which have been incorporated in this report. He has worked diligently, meticulously and methodically. The report has been found satisfactory and is approved for submission.

**Dr. Vinayak Srivastava,**  
CS 5301 - Software Engineering Tools Lab,  
Department of Computer Engineering,  
Institute of Technology,  
Banaras Hindu University.

# Table of Contents

1. Introduction.....	4
2. Build Automation.....	5
2.1 Makefiles.....	5
2.2 CMake.....	6
3. Cross-platform Build Automation .....	9
4. CMakeQt.....	10
4.1 Source Tree.....	10
4.2 Build Process.....	11
4.3 Testing and Conclusions.....	13
References.....	14

# 1. Introduction

Historically, developers used build automation to call compilers and linkers from inside a build script versus attempting to make the compiler calls from the command line. It is simple to use the command line to pass a single source module to a compiler and then to a linker to create the final deployable object. However, when attempting to compile and link many source code modules, in a particular order, using the command line process is not a reasonable solution. The make scripting language offered a better alternative. It allowed a build script to be written to call in a series, the needed compile and link steps to build a software application. [2]

GNU Make [3] was one of the first and most successful build automation tool still used today, it also offered additional features such as "makedepend" which allowed some source code dependency management as well as incremental build processing. This was the beginning of Build Automation. Its primary focus was on automating the calls to the compilers and linkers. As the build process grew more complex, developers began adding pre and post actions around the calls to the compilers such as a check-out from version control to the copying of deployable objects to a test location. The term "build automation" now includes managing the pre and post compile and link activities as well as the compile and link activities.

This report describe an on-demand cross-platform build automation tool called CMakeQt which uses pre-existing build automation tools such as *cmake*, *make* etc. and exclusively provides a solution for cross-platform software development requirements of a typical C/C++ Qt project.

## 2. Build Automation

Build automation [1] is the act of scripting or automating a wide variety of tasks that software developers do in their day-to-day activities including things like:

- compiling computer source code into binary code
- packaging binary code
- running tests
- deployment to production systems
- creating documentation and/or release notes

Build automation becomes really essential for medium to large projects where the build process applies to hundreds, thousands or even larger number of source files. Advantages of build system automation are as follows:

- Improve product quality
- Accelerate the compile and link processing
- Eliminate redundant tasks
- Minimize "bad builds"
- Eliminate dependencies on key personnel
- Have history of builds and releases in order to investigate issues
- Save time and money - because of the reasons listed above.

Based on usage, build automation tools also called build systems can be classified into following broad types:

1. On-Demand automation, such as a user running a script at the command line
2. Scheduled automation, such as a continuous integration server running a nightly build
3. Triggered automation, such as a continuous integration server running a build on every commit to a version control system.

### 2.1 Makefiles

Makefiles are simple text files which specify the rules and steps to derive a target program, and it is read by *Make* [4] which is a utility that automatically builds executable programs and libraries from source code by reading the *makefiles*. Though integrated development environments

and language-specific compiler features can also be used to manage the build process in modern systems, Make remains widely used, especially in Unix.

Following listing shows an example Makefile. To start this on-demand build process, **make all** is run on a terminal which targets to create the target by compiling C source files.

```
CC      = gcc
CFLAGS = -g

all: helloworld

helloworld: helloworld.o
    # Commands start with TAB not spaces
    $(CC) $(LDFLAGS) -o $@ $^

helloworld.o: helloworld.c
    $(CC) $(CFLAGS) -c -o $@ $<

clean:
    rm -f helloworld helloworld.o

# This is GNU makefile extension to notify that roughly means: 'clean' does
# not depend on any files in order to call it.
.PHONY: clean
```

One specific form of build automation is the automatic generation of Makefiles and which is also used to develop CMakeQt. This is accomplished by tools such as GNU Automake , CMake , qmake etc.

## 2.2 CMake

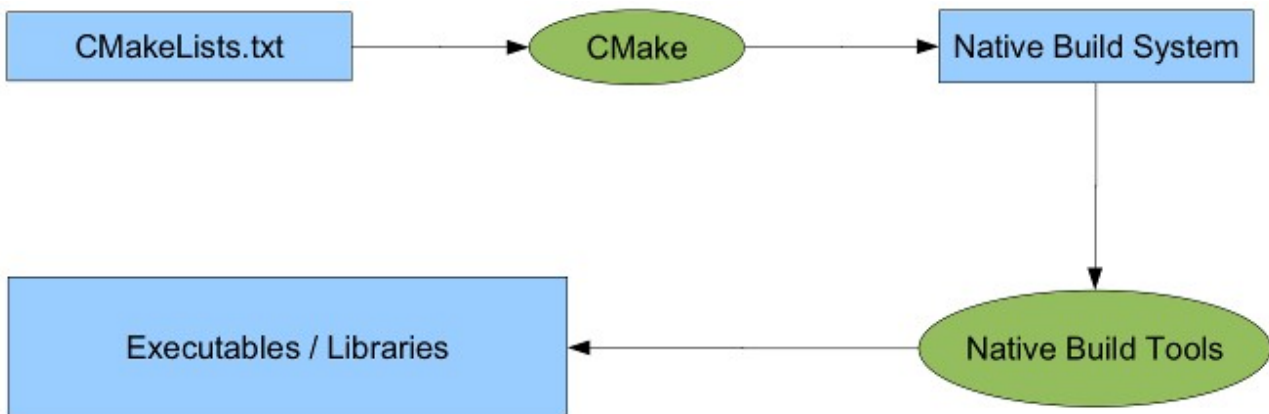
CMake [4] [5] is a cross-platform, open-source system for managing the build process of software using a compiler-independent method. It is designed to support directory hierarchies and applications that depend on multiple libraries, and for use in conjunction with native build environments such as Make.

Advantages of using CMake for *makefile* [5] generation are as follows:

- Cross-platform
- Very simple script language
- Dependency discovery is awesome: FIND\_PACKAGE
- Scales very well
- Creates a project files for Makefile, Visual Studio, Kdevelop, Eclipse, etc

- Users can use the tools they are used to

Typical build process with CMake consists of two stages. First, standard build files are created from configuration files. Then the platform's native build tools are used for the actual building. Following diagram shows the build process in action:



Each build project contains a CMakeLists.txt file in every directory that controls the build process. The CMakeLists.txt file has one or more commands in the form `COMMAND (args...)`, with `COMMAND` representing the name of each command and `args` the list of arguments, each separated by white space. While there are many built-in rules for compiling the software libraries (static and dynamic) and executables, there are also provisions for custom build rules. Some build dependencies can be determined automatically. Advanced users can also create and incorporate additional makefile generators to support their specific compiler and OS needs.

CMake can handle in-place and out-of-place builds, enabling several builds from the same source tree, and cross-compilation. The ability to build a directory tree outside the source tree is a key feature, ensuring that if a build directory is removed, the source file remains unaffected.

Another feature of CMake is the ability to generate a cache to be used with a graphical editor, which, when CMake is run, can locate executables, files and libraries. This information goes into the cache, which can then be tailored before generating the native build files.

Complicated directory hierarchies and applications that rely on several libraries are well supported by CMake. For instance, CMake is able to accommodate a project that has multiple toolkits, or libraries that each have multiple directories. In addition, CMake can work with projects that require executables to be created before generating code to be compiled for the final application. Its open-source, extensible design allows CMake to be adapted as necessary for specific projects.

CMake can generate makefiles for many platforms and IDEs including Unix, Windows, Mac OS X, MSVC, Cygwin, MinGW and Xcode.



### 3. Cross-platform Build Automation

In desktop software development, because of diversity of popular operating systems such as Windows, Mac OSX, Linux, BeOS, Haiku, BSD etc.; software developers are required to develop and publish cross-platform, or multi-platform software which is implemented and can inter-operate on multiple computer platforms.

Cross-platform software may be divided into two types; one requires individual building or compilation for each platform that it supports, and the other one can be directly run on any platform without special preparation, e.g., software written in an interpreted language or pre-compiled portable bytecode such as Java or Python applications.

Throughout the document by cross-platform software we mean the software which can be individually built or compiled for multiple platforms and we focus on the top three popular platforms or OS: Linux, Mac OSX and Windows. Most popular open source applications such as VLC, Firefox, Chrome, Blender, GIMP, are cross-platform in the way that they can be built on various platforms using the same source code.

While most desktop application are developed in C/C++, to solve the problem of supporting cross-platform applications Qt was created in 1992 and is the defacto cross-platform application framework. Qt is open source and is supported on Linux, Windows, Mac, Embedded Linux, Symbian, MeeGo/Maemo, Windows CE, Haiku, OpenSolaris, OS/2, webOS etc. And has a range of bindings for various languages. In our build system automation tool we've explicitly supported C/C++ based Qt projects.

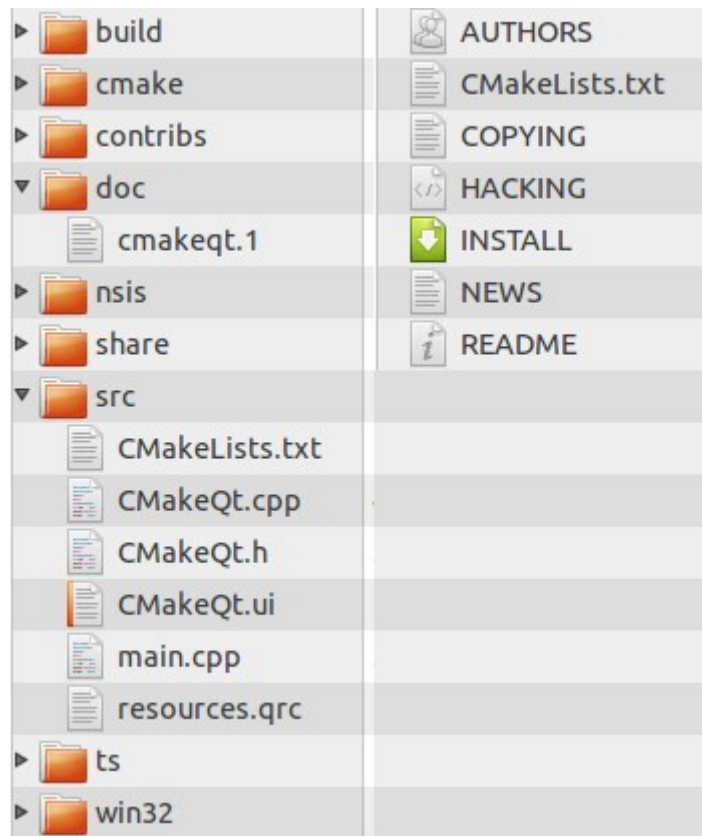
## 4. CMakeQt

CmakeQt is a cross-platform build automation tool and open-sourced as a build system template which any developer who wants to develop cross-platform desktop application can drag and drop the template, change certain variables and paths without the hassel to understand the build system or the build process thereby helping the developer to focus on the project. It simply requires CMake and build-tools (make and c/c++ compiler); and has following features:

1. Cross-platform, works on Linux, Mac OSX and Windows.
2. Asthetic terminal/text-based interface, which shows highlighted build activities and progress percentage.
3. Versioning/tagging support.
4. Automatic build tools detection to detect C/C++ compilers and Qt libraries.
5. Translation support, automatically compiles .ts files and links to compiled binary.
6. Cross-compilation for Windows on Linux using MINGW (Minimalistic GNU compiler for Linux) with Win32 contribs (precompiled Qt libraries and headers).
7. Documentation, *man pages* for Linux.
8. Processes application defined and dependent identifiers, icon and media files.
9. Packaging enabled; deb/rpm for Linux, bundled App or DMG for Mac OSX and NSIS [7] based installer for Windows.

### 4.1 Source Tree

The source tree can be seen in the following snapshot which lists various files and directories in the root folder:

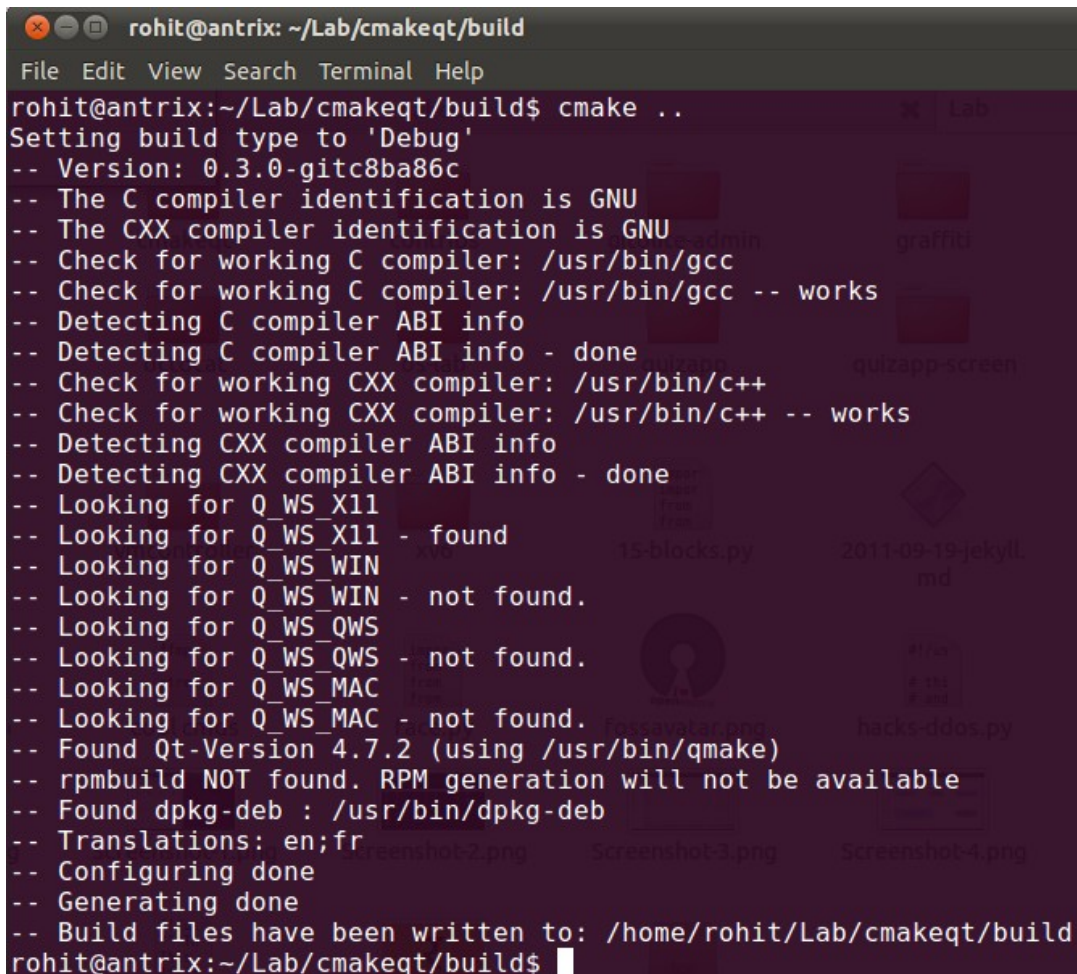


The *cmake* folder contains CMakeQt specific modules which has rules that enable rpm/deb packaging, automatic config.h module (for use in application to detect/set/get platform/os specific variables), rules for icon creation and cross-compilation MINGW32 rules for Windows on Linux. The *contribs* folder contains scripts to fetch Windows and Mac OSX Qt libraries and headers that are used to link them to the built application. The *doc* folder contains a simple man page template for providing application documentation on Linux. The *nsis* folder contains scripts and CMakeLists.txt file to packaging cross-compiled win32 application as a NSIS [7] installer for Windows. The *share* folder contains icons, media images and files used by the application. The *src* folder contains project source files. The *ts* folder contains translation files and rules. The *build* and *win32* folder are temporary folders where building takes place for Linux/Mac and Windows respectively.

## 4.2 Build Process

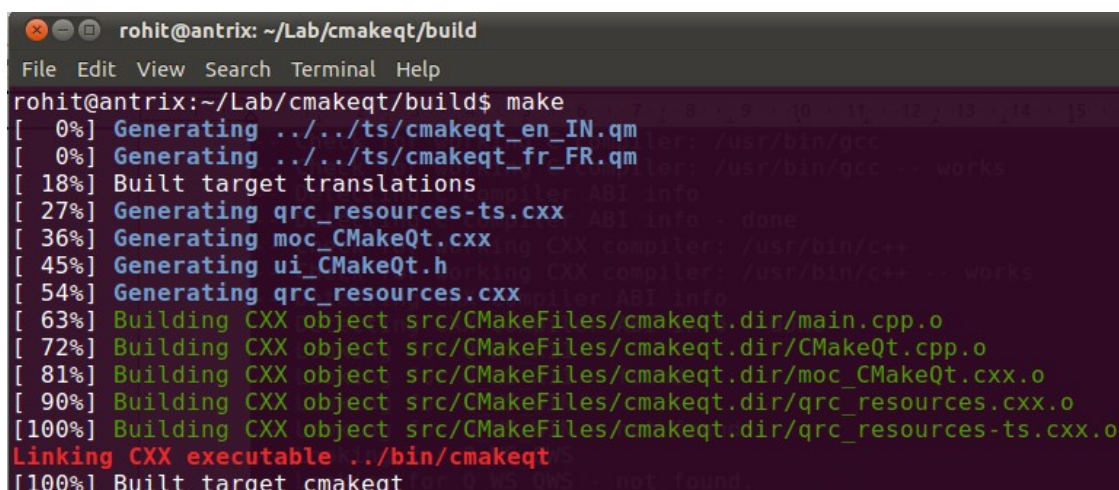
Each folder, except the temporary ones, contains a CMakeLists.txt file which sets up commands and rules and add sub directories. When *cmake* is evoked it recursively processes all these files and generates makefiles which then can be simply evoked by a *make* command. For example to simply build the project, one needs to create a temporary folder

(*build* in our example), switch to that directory and run *cmake*; following snapshot of the terminal shows this process and output:



```
rohit@antrix: ~/Lab/cmakeqt/build
File Edit View Search Terminal Help
rohit@antrix:~/Lab/cmakeqt/build$ cmake ..
Setting build type to 'Debug'
-- Version: 0.3.0-gitc8ba86c
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Looking for Q_WS_X11
-- Looking for Q_WS_X11 - found
-- Looking for Q_WS_WIN
-- Looking for Q_WS_WIN - not found.
-- Looking for Q_WS_QWS
-- Looking for Q_WS_QWS - not found.
-- Looking for Q_WS_MAC
-- Looking for Q_WS_MAC - not found.
-- Found Qt-Version 4.7.2 (using /usr/bin/qmake)
-- rpmbuild NOT found. RPM generation will not be available
-- Found dpkg-deb : /usr/bin/dpkg-deb
-- Translations: en;fr
-- Configuring done
-- Generating done
-- Build files have been written to: /home/rohit/Lab/cmakeqt/build
rohit@antrix:~/Lab/cmakeqt/build$
```

After that one simply needs to evoke *make* to build the target, as shown in the following snapshot:



```
rohit@antrix: ~/Lab/cmakeqt/build
File Edit View Search Terminal Help
rohit@antrix:~/Lab/cmakeqt/build$ make
[ 0%] Generating ../../ts/cmakeqt_en_IN.qm
[ 0%] Generating ../../ts/cmakeqt_fr_FR.qm
[ 18%] Built target translations
[ 27%] Generating qrc_resources-ts.cxx
[ 36%] Generating moc_CMakeQt.cxx
[ 45%] Generating ui_CMakeQt.h
[ 54%] Generating qrc_resources.cxx
[ 63%] Building CXX object src/CMakeFiles/cmakeqt.dir/main.cpp.o
[ 72%] Building CXX object src/CMakeFiles/cmakeqt.dir/CMakeQt.cpp.o
[ 81%] Building CXX object src/CMakeFiles/cmakeqt.dir/moc_CMakeQt.cxx.o
[ 90%] Building CXX object src/CMakeFiles/cmakeqt.dir/qrc_resources.cxx.o
[100%] Building CXX object src/CMakeFiles/cmakeqt.dir/qrc_resources-ts.cxx.o
Linking CXX executable ../bin/cmakeqt
[100%] Built target cmakeqt
```

The compiled binaries can be found in the *build/bin* folder. To

package the binaries, one can evoke *make package*. To cross-compile for Windows on Linux, one needs to simply specify the cross-compilation toolchain rule:

```
cmake -DCMAKE_TOOLCHAIN_FILE=cmake/toolchain-win32.cmake ..
```

When *make* is run MINGW32 compiler toolchain will carry out the build process, to create the NSIS installer for Windows one needs to evoke *make installer*.

For Mac OSX, simple *cmake* and *make* commands will do the trick and create the App/DMG file in the build/bin folder. To add/remove source files and manipulate settings one simply needs to change variables and paths in the CMakeLists.txt files in the root folder and in the src folder.

### **4.3 Testing and Conclusions**

CMakeQt was successfully tested on; Ubuntu Linux 11.04 - x86\_64 and i386; Mac OSX Snow Leopard i386; Windows 7 i386. The results were satisfactory and we conclude that it can be used as a software engineering tool for cross-platform build automation across popular operating systems.

## References

- [1] Build Automation: [http://en.wikipedia.org/wiki/Build\\_automation](http://en.wikipedia.org/wiki/Build_automation)
- [2] Matthew Doar (2005). Practical Development Environments. O'Reilly Media. pp. 94. ISBN 978-0596007966.
- [3] GNU Make. Free Software Foundation:  
<http://www.gnu.org/software/make/manual/make.html>
- [4] CMake Build System Tool, *cmake.org*.
- [5] Pau Garcia i Quiles. Learning CMake , 2008.
- [6] CmakeQt: <https://github.com/rohityadav/cmakeqt/>
- [7] NSIS, a scriptable win32 installer/uninstaller: *nsis.sourceforge.net*