**Compile Qt Statically with MySQL**

The version of Qt that comes as a download from Trolltech does not have MySQL support out of the box. This can be compiled as a module very easily. There are instructions for that in their documentation. My goal was to be able to distribute an application that depended on Qt, MySQL, and Qwt without needing to install anything on the client computer. So I needed to compile the whole thing statically. Additionally, I’m a newb at C++, so I wanted the debug versions of all my libraries lying around so that I could more easily fix my bugs. Figuring out how to get static, debug versions of this whole stack created involved piecing together a lot of info on the internet, and a lot of trial and error. I’m going to put up my “recipe” in the hopes that it will help someone else.

Note that the full build of Qt with both release and debug static libraries will require something on the order of 6 GB of disk space. While this isn’t a big deal by today’s disk sizes, I was doing this in a virtual machine, and ran out of space. In the examples below, I’ve created a secondary disk, mounted as E:, where I manage all of this software. I typically install the various pieces of software in version-numbered subdirectories, so that I can easily have several versions sitting side-by-side.

First, install the Qt SDK for Windows. Also get the mingw-utils archive and unzip it into the MinGW subdirectory created under the Qt SDK installation location. The current versions at the time of this writing are qt-sdk-win-opensource-2009.02.1.exe and mingw-utils-0.3.tar.gz.

Second, get the MySQL download — *without* the installer — and extract it someplace on disk. At this point, you have to modify a couple of the libraries to work with mingw compiler. (I’ll readily admit that I am very foggy on why or what this accomplishes, but this is what needs doing.) Open a *regular* command prompt and do the following:

E:\MySQL\5.0.67\lib\opt> set PATH=%PATH%;E:\Qt2009.02.1\MinGW\bin

E:\MySQL\5.0.67\lib\opt> reimp -d libmysql.lib

E:\MySQL\5.0.67\lib\opt> dlltool --input-def libmysql.def --dllname libmysql.dll --output-lib libmysql.a -k

E:\MySQL\5.0.67\lib\debug> reimp -d libmysql.lib

E:\MySQL\5.0.67\lib\debug> dlltool --input-def libmysql.def --dllname libmysql.dll --output-lib libmysql.a -k

E:\Qt\4.4.3\> configure -platform win32-g++ -static -debug-and-release -no-accessibility -no-sql-sqlite -qt-sql-mysql -I e:\mysql\5.0.67\include -L e:\mysql\5.0.67\lib\opt -l mysql

E:\Qt\4.4.3> mingw32-make sub-src

configure -platform win32-g++ -static -debug-and-release -nomake examples -nomake demos -qt-sql-mysql -I C:\MySQL\Server\include -L C:\MySQL\Server\lib -l mysql

Get the Qwt source files. Edit the qwtconfig.pri file and comment out the QwtDll line in order to make a static library, and change the “release” keywords to “debug.” Then, use a *Qt command prompt* and do the following:

E:\Qwt\5.1.1> qmake

E:\Qwt\5.1.1> make sub-src

Now you can edit your application’s .pro file like this:

CONFIG += static  
INCLUDEPATH +=  E:/Qwt/5.1.1/src  
LIBS += E:/Qwt/5.1.1/lib/libqwt.a

Note that *if* the library is built as *shared*, and Qwt is *actually* installed, it’ll wind up going in something like C:\QWT-5.1.1, the INCLUDEPATH would point at C:\QWT-5.1.1\include, and the LIBS would point at C:\QWT-5.1.1\lib\qwt.dll.

In my case, I have to distribute mingwm10.dll and libmysql.dll with my application. I’ve seen some writeups that talk about how disabling threads will allow the MinGW runtime to be compiled into the application statically, but it would require rebuilding MinGW from source, which seems really difficult. Also, it should be possible to rebuild the MySQL library as static, and have it get built into the application during the process above. Again, that seems really difficult, and would — I’m sure — require loads of time and disk space. At the end of the day, this method ultimately needs but two libraries to be distributed along with the application binary, avoiding the need to install Qt and Qwt on the end-user’s computer, and that’s good enough for me.

The default build command in the Qt command window will build the debug version of the application by default. The `make release’ command is needed to build the non-debug version via the default-created makefile.

**Shared Setup (for reference)**

Alternately, the MySQL plugin can be produced to work with the default install version of Qt. This will then require that Qt be installed on every machine that will run the program, and that it be included in the machine’s PATH. To compile this plugin, MySQL will still have to be extracted to the filesystem, and the library reconfigured, all as above.

Use a Qt Comand Prompt:

C:\Qt\4.4.3\src\plugins\sqldrivers\mysql> qmake "INCLUDEPATH+=C:\MySQL\5.0.67\include" "LIBS+=C:\MySQL\5.0.67\lib\opt\libmysql.a" mysql.pro

Note that the “libmysql.a” above differs from Qt’s documentation on this process. In their official instructions, this is a .lib file, but I have found that this does not work.

To go along with using a shared setup of Qt, Qwt only needs to have its qwtconfig.pri file edited to reflect the desired installation path, and then one can `qmake’, `make’, and `make install’.

In the shared configuration, the Windows PATH environment variable will need to be set to include the following (examples): C:\MinGW\bin;C:\Qt4.4.3\bin;C:\MySQL\5.0.67\lib\opt;C:\Qwt\5.1.1

## Compilação estática em C++ e QT no Windows

17/11/2009 [Hamilton Sena](http://hsena.wordpress.com/author/hsena/) [Deixe um comentário](http://hsena.wordpress.com/2009/11/17/compilacao-estatica-em-c-e-qt/#respond) [Go to comments](http://hsena.wordpress.com/2009/11/17/compilacao-estatica-em-c-e-qt/#comments)

Para adicionar as bibliotecas estáticas aos programas gerados em C++ e QT é necessário recompilar o QT, pois o que é instalado por padrão utiliza as bibliotecas dinâmicas, dificultando assim a distribuição do aplicativo.

1. Set as variáveis de ambiente include e lib.
   * No menu “Iniciar”, selecione a opção “Painel de controle”;
   * Acesse o item “Sistema”;
   * Selecione a guia “Avançado”;
   * Acesse “Variáveis de ambiente”, e no campo Variáveis do Sistema acesse “Nova”
   * Cria duas variáveis, uma INCLUDE e o outra LIB
     + Na variável INCLUDE adicione a seguinte linha no campo “Valor da variável”

C:\Qt\mingw\include

* + - Na variável LIB adicione a seguinte linha no campo “Valor da variável”

C:\Qt\mingw\lib

1. OBS: Caso o caminho da instalaçõa do seu QT não seja C:\QT, troque o pelo caminho correto.
   * Pressione o botão “OK” para confirmar cada uma das configurações.
2. **Entre no console da Qt:** Menu iniciar -> Qt SDK (ou outro) -> Qt Command Prompt
3. **Execute o comando:** configure -static -no-phonon
4. **Execute o comando:** mingw32-make sub-src (Esse comando deve demoras algumas horas, dependendo do seu hardware).

Após efetuar essa recompilação ao gerar algum projeto no QTCreator adicione a seguinte linha no arquivo .pro:

CONFIG += static

Pronto, agora para disponibilizar seus programas apenas envie o executável gerado com a dll mingwm10.dll.

## Building Qt Static (and Dynamic) and Making it Small with GCC, Microsoft Visual Studio, and the Intel Compiler

October 11th, 2009 [Charles N. Burns](http://www.formortals.com/author/Charles/) [Leave a comment](http://www.formortals.com/build-qt-static-small-microsoft-intel-gcc-compiler/#respond) [Go to comments](http://www.formortals.com/build-qt-static-small-microsoft-intel-gcc-compiler/#comments)

[](http://www.formortals.com/build-qt-static-small-microsoft-intel-gcc-compiler/)

This article will show you how to build Qt, the popular C++ framework from Nokia, so that it is both small and, if you prefer, available for [static linking](http://www.formortals.com/how-to-statically-link-qt-4/). Your Qt applications will be smaller, possibly faster, and can be distributable as a single executable.

Also answered: How small can Intel’s C++ compiler make a large library? How does Microsoft fare? Three compilers (settings tuned for small file output) and their resulting code size is compared.

### A nice table of contents so that you can see what you’re getting into:

\* optional

1. **Download the latest Qt source code and put it in its own directory.**
2. **\*Modify the compiler flags for use when building Qt.**
3. **Open a command-line window for your compiler.**
4. **Configure.**
5. **Compile.**
6. **\*If you want to use static linking, modify your Qt project.**

A few things to keep in mind before we get started:

Before getting started, you may wish to install a separate compiler. The full Qt SDK comes with G++3 as of this writing (G++ is the C++ compiler that comes with GCC). It works, but G++4 generates better code. I recommend the [TDM release](http://www.tdragon.net/recentgcc/). Microsoft Visual Studio Intel’s compiler are also very capable. This article covers them all.

G++ and the Intel compiler are compatible. Either compiler can link to libraries built with the other compiler.  
Microsoft’s C++ compiler is incompatible with the other two, so if you build Qt with it, you are stuck with Microsoft’s compiler for the whole project. This isn’t necessarily bad.  Visual C++ is a fine compiler, and finishes compiling Qt noticeably faster than the other two.

GCC is available free of charge, source code included.  
Microsoft’s compiler is available for free in their “[Express edition](http://www.microsoft.com/Express/VC/)“. It is more or less fully functional.  
Intel’s compiler, however, is only available free of charge on the Linux platform, and even then [only for non-commercial](http://software.intel.com/en-us/articles/non-commercial-software-development/) software development. That said, it is widely thought of as producing the fastest code.

With that out of the way, let’s begin:

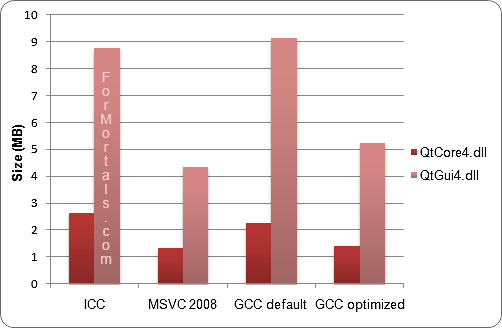
1. **Download the latest Qt source code and put it in its own directory.**You can get the [entire SDK](http://qt.nokia.com/downloads) or  [just the source code](http://download.qt.nokia.com/qtsdk/" \o "Use the source" \t "_blank). Either way, each compilation should be in a separate folder.
2. \***Modify the compiler flags for use when building Qt.**You’ll be changing one line of the file “make.conf” in the “mkspecs” folder. If Qt in the “C:\Qt” folder, for example, then the file is “C:\Qt\mkspecs\COMPILER\make.conf”, where COMPILER is :  
   “**win32-g++**” for GCC  
   “**win32-icc**” for the Intel compiler  
   “**win32-msvc2008**” for the Microsoft compiler (replace 2008 with the year of release).In all three cases, you will edit the line: “**QMAKE\_CFLAGS\_RELEASE =**“, changing the contents after the equals sign to the following:
   1. For GCC: -Os -momit-leaf-frame-pointer
   2. For ICC: -Os -Oy
   3. For VC++: -O1 -Og  -GL -MD

For example, my ICC line shows: “QMAKE\_CFLAGS\_RELEASE    = -Os -Oy”  
You may wish to add other optimizing flags as well. See your compiler’s documentation.

1. **Open a command-line window appropriate for your compiler**.
   1. GCC: Open the command prompt (Start –> Run –> “cmd”), then run “mingwvars.bat”. For example, if GCC is installed in “c:\MinGW”, then enter the command: “c:\MinGW\mingwvars.bat“
   2. Microsoft Visual Studio: Look for the “Visual Studio 2008 Command Prompt” entry in your start menu. It is usually found under “Programs\Microsoft Visual Studio 200X\Visual Studio Tools\”. The Express versions should be similar. Alternatively, search your hard drive for “vcvarsall.bat” and from that folder, run “vcvarsall.bat x86” from the command line.
   3. Intel compiler: Similar to Visual Studio, an icon is provided in your start menu, usually in: “Programs\Intel(R) Software Development Tools\Intel(R) C++ Compiler x.y.z“. Alternatively, find iclvars.bat and from that folder, run “iclvars.bat ia32“
2. **Configure Qt.**  
   Change to the Qt folder. For example, type cd C:\Qt\4.6.3-msvc if you unzipped Qt to C:\Qt\4.6.3-msvc. This folder should have configure.exe within. Run the following long command:  
   configure -release  -nomake examples -nomake demos -no-exceptions -no-stl -no-rtti -no-qt3support -no-scripttools -no-openssl -no-opengl -no-webkit -no-phonon -no-style-motif -no-style-cde -no-style-cleanlooks -no-style-plastique -no-sql-sqlite
   1. Add “**-platform win32-???**” for your compiler. Replace “???” with the name of your compiler, same as in step 2. You can also look for your compiler’s name in the Qt\mkspecs folder. For example, a Microsoft Visual Studio 2008 user would use “-platform win32-msvc2008“.
   2. Add “**-static**” if you are compiling for static linking (libraries included in your .exe rather than as separate files. This mostly applies to qtcore4.dll and qtgui4.dll)
   3. Add any static functionality your application needs., if you are compiling static For example, **-qt-libjpeg -qt-zlib -qt-libpng** for JPEG image, ZIP compression, and PNG (which also needs zlib) respectively.  See [How to Statically Link Qt4](http://www.formortals.com/how-to-statically-link-qt-4/).
   4. Remove “**-no-exceptions -no-stl -no-rtti**” if you need those C++ features. Note that with **-no-stl**, you are still able to use the STL, but Qt will not have built-in convenience functions to, for example, set a QVector equal to a C++ vector. That is, you will have to do so manually.
   5. Replace “**-release**” with “**-debug-and-release**” if this will be your debug install, too. Do not use -static.
   6. The remaining “**-no-foo**” options disable various features. See the output of “configure –help” for documentation. You can write this to a file with “configure –help > options.txt“.
3. **Compile Qt**.  
   After step 4, you will be informed of which command starts the actual compile (usually **mingw32-make** or **nmake**).
4. (static only) [Configure your project](http://www.formortals.com/how-to-statically-link-qt-4/) for use of the static library.

### It’s the size that counts and how you use it.

When working towards making Qt (and therefore your projects) small, it is important to have some idea of what each compiler can do. I will compare the sizes of **QtCore4.dll** (core Qt functionality) and **QtGui4.dll** (Graphical interface libraries and operating system interface). These two files that are fairly representative of size scaling among Qt libraries, and also happen to be the two most common files needed by Qt applications.



### Interpretation:

* ICC’s code size is about the same as GCC’s unoptimized. The sum of both file’s sizes is 11.38MB on both compilers.
* With appropriate settings, GCC’s compiled Qt code size drops more than 40%, from 11.38MB to only 6.63MB combined.
* Microsoft Visual C++ handily beats the other compilers, with a library size less than half of ICC’s and about 15% smaller than GCC’s best effort.(Edit: Arik notes that this compilation is still dependent on Microsoft’s msvcrtXX.dll. See his comment below regarding how to eliminate this dependency. This may change the size of your application).

Additionally, Microsoft’s compiler finished the job in about 1/5 the time of ICC and about 1/3 the time of GCC Optimized. I actually ran it twice because I thought it may have aborted without error, because the time difference was so substantial.

**Update**: I’ve compiled [CNB ImageGuide](http://www.formortals.com/introducing-cnb-imageguide/) to provide a real application size example. Using G++, executable size is 5.8MB. Using Microsoft C++ 2008 and Arik’s suggestion, it is 3.56MB, or about a 60% difference. I’d like to welcome theories as to why the difference is so substantial.

**However**:

* This does not take into account the performance of the resulting libraries, with Intel likely taking the lead.
* ICC is designed for absolute performance, not small code size.
* MSVC is not available on most platforms supported by Qt.
* The author is far more familiar with the compiler flags for G++ than for the other two compilers.  It is possible that a particular combination of options will significantly change the results.

I hope this has been helpful and informative. Communicate any feedback, good or bad, in the comments section.

### Repeatability

Everyone is welcome to try to repeat, improve upon, or dispute my results. The settings used follow.

**Intel Compiler 11**

cflags: -Os -Oy  
Qt configuration:  
-release -qt-libjpeg -qt-zlib -qt-libpng -nomake examples -nomake demos -no-exceptions -no-stl -no-rtti -no-qt3support -no-scripttools -no-openssl -no-opengl -no-webkit -no-phonon -no-style-motif -no-style-cde -no-style-cleanlooks -no-style-plastique -no-sql-sqlite -platform win32-icc

**Microsoft Visual C++ 2008**

cflags: -O1 -GL -MD  
Qt configuration:  
-release -qt-libjpeg -qt-zlib -qt-libpng -nomake examples -nomake demos -no-exceptions -no-stl -no-rtti -no-qt3support -no-scripttools -no-openssl -no-opengl -no-webkit -no-phonon -no-style-motif -no-style-cde -no-style-cleanlooks -no-style-plastique -no-sql-sqlite -platform win32-msvc2008

**GCC 4.4.1 default (TDM release 2)**

cflags: -O2  
Qt configuration:  
-release -nomake examples -nomake demos -platform win32-g++

**GCC 4.4.1 Optimized**

cflags: -Os -mpreferred-stack-boundary=2 -finline-small-functions -momit-leaf-frame-pointer  
Qt configuration:  
-release -qt-libjpeg -qt-zlib -qt-libpng -nomake examples -nomake demos -no-exceptions -no-stl -no-rtti -no-qt3support -no-scripttools -no-openssl -no-opengl -no-webkit -no-phonon -no-style-motif -no-style-cde -no-style-cleanlooks -no-style-plastique -no-sql-sqlite -platform win32-g++