QNX SOFTWARE SYSTEMS

4-Feb-2012 dkelly@qnx.com

Moving Qt Creator Projects to QNX

Introduction

Qt Creator is a great tool for quickly prototyping and testing a UI framework. Most importantly, Qt Creator is tracking current releases (it's already available for 4.8.0).

While Qt Creator is not available for running on embedded platforms like QNX, you can still leverage it in your development cycle. Use Qt Creator to build/test/debug your UI components on a desktop before you deploy them to your embedded system.

This app note describes

- A design paradigm which leverages Qt Creator for embedded projects
- Details useful when moving source from Qt Creator to QNX
- Use of qmake_qnx.bat file for automating qmake on QNX
- Example build details

Keep the 'business logic' Out of Your UI

If you are going to maximize the benefits from Qt Creator on the desktop, then code that applies only to the embedded system must be kept out of your UI. (Remember, QNX code can't be tested on the desktop!)

To understand this concept, think of your UI as if it were a browser application with a server running on the embedded system. Since your UI is remote, it is totally impossible to reach "into" the system and grab some piece of status info. Nor can it kickoff a native process or thread. Instead, the UI must obtain status and issue commands via some communication channel to a "server" resident on the embedded system.

This UI-to-server communication channel may be over a socket, but that is not the only way. You could use QNX "pps" publish/subscribe service – or you could use QNX native message passing. For this discussion, the method is not as important as the concept of strictly isolating the "business logic" from the UI via an API.

What if I build a monolithic program?

- Qt Creator will be of limited use. Either you will abandon it after the initial prototype, or you will have to build an elaborate embedded system simulator on the desktop. *That simulator will require lots of development for a piece of "disposable" code!*

- Your UI developers and your back-end developers will be constantly "tripping over each other", waiting for code to be checked in just so they can build and test their own code!

- When a problem arises, is it the fault of the UI or the back-end code? It can be hard to isolate in a monolithic program, but easier if there is a clear demarcation between the two components.

- When marketing says "time for a new face", you must move all the back-end code to the new UI and spend time locating all the existing "side-effects"… this usually results in a total rewrite!

- When marketing says "time for a smartphone control app", you are again looking at a re-write – not just a "plumbing" effort as would be the case with an independent back-end api.

In QNX, defining your Qt program as a "client" to a back-end "server" allows most UI debugging to be done on a desktop while the back-end is developed and debugged independently using the Momentics IDE.

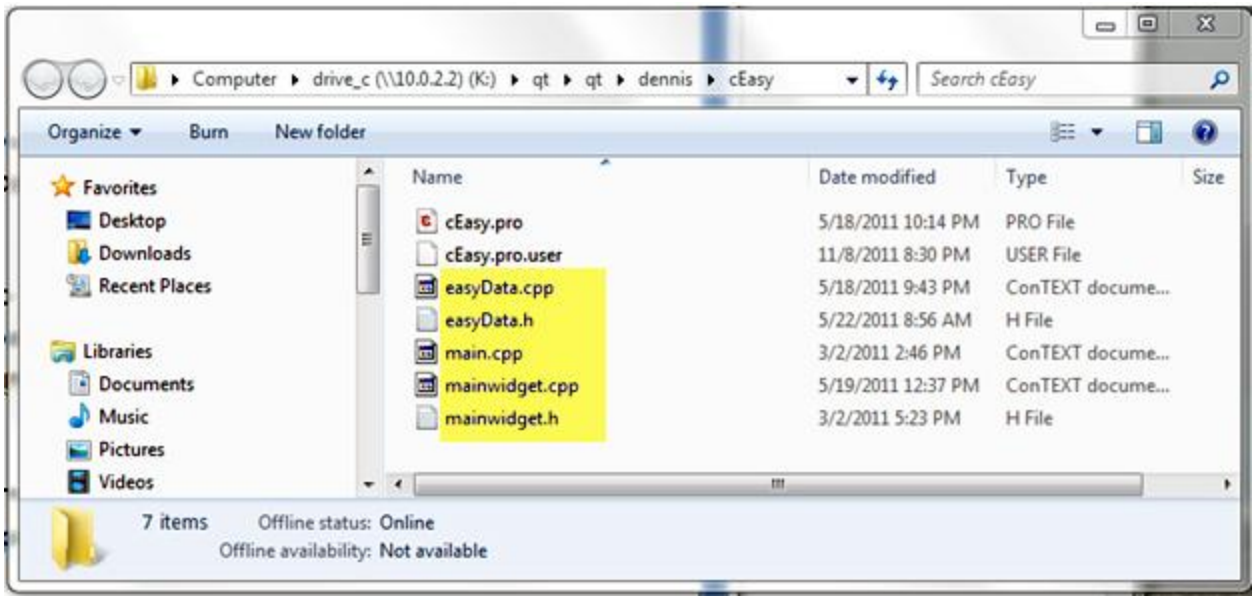
Moving UI Source to QNX

So if you your UI is developed on a desktop using Qt Creator, how to you move it to QNX for compilation and deployment to the embedded target? Clearly the makefile and project file are totally different for the desktop than for QNX.

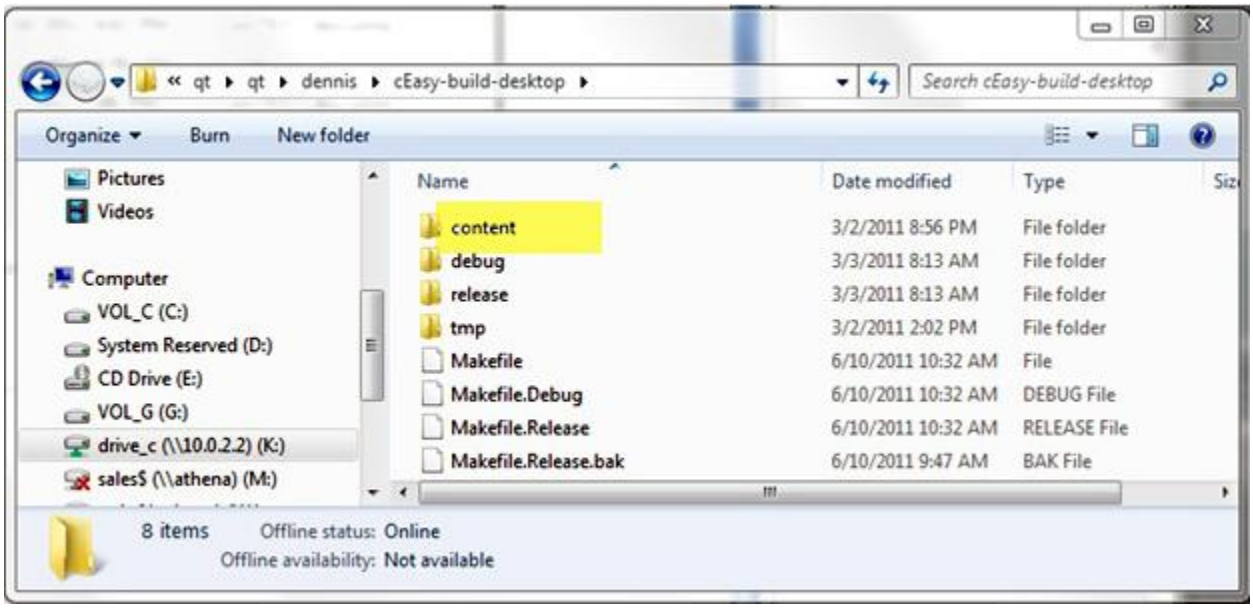
Fortunately, the Qt "qmake" utility is quite smart. All we really need to do is move the .cpp files, the .h files plus any QML assets from the desktop to the QNX development host. Once these files are in place, running qmake against them generates a project file and a makefile. Simple in concept, but "the devil is in the details".

There are a couple of issues. First, a specification of the target platform is required – this is not "contained" in the desktop source files. Second, QNX qmake does not pickup all Qt module dependencies from the source files so these need to be specified as well.

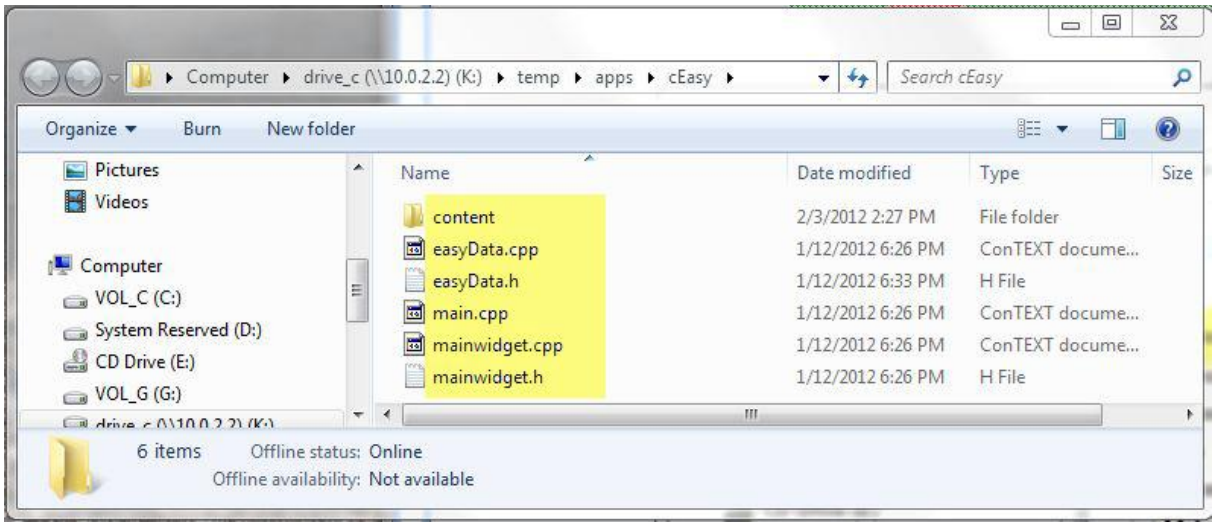
Qt Creator on the desktop "splits" a project into two folders. (I will be using the Qt Creator project for "cEasy" as the example.) One folder (cEasy) contains the source files (.cpp and .h) and the project files (.pro).



The other folder (cEasy-build-desktop) contains the Makefiles for specific builds and build output.



For QtDeclarative, the model used by cEasy, the “content” folder containing all the assets is located in this second directory. The objects highlighted in yellow must be moved to a QNX project directory as shown here.



From the above folder (K:\temp\apps\cEasy) it is only necessary to run two commands to build the project for QNX.

```
> qmake_qnx armv7
> make
```

The "magic" is contained in qmake_qnx.bat. The single required parameter must be "arm", "armv7" or "i386", specifying the toolchain to use. cEasy.pro, Makefile and directory "armle-v7" will appear, the latter containing the output of the build.

A couple of details…

- Put qmake_qnx.bat in c:\qnx650\host\win32\x86\usr\bin
- Run qmake_qnx from a Windows cmd prompt – not from minGW or cygwin
- Run qmake_qnx with the current directory containing your project source
- Run qmake_qnx with no parameters to see the usage text

Depending on your application, it may be necessary to add additional modules to the list in the line below from qmake_qnx.bat:

```
QT += core gui script declarative %QT_MODULES% >> %PROJ_NAME%.pro
```

You will need to add a module if your application fails to build. From the error output you should be able to deduce what is missing. Either modify qmake_qnx.bat and add to the list above, OR set environment variable QT_MODULES to the list of additional required modules (space delimited).

Enabling Debug

The qmake_qnx.bat file will default your Makefile to "release". If you want to enable debugging you can set environment variable QT_CXXFLAGS to "-g" (or for even more debug info use "-g3"). The command is:

```
> set QT_CXXFLAGS=-g
```

The qmake_qnx.bat File

Here is the complete file. Copy and paste to notepad and save as

C:\qnx650\host\win32\x86\usr\bin\qmake_qnx.bat

```
@if "%1"=="" goto help

@call pwd >temp.bat
@set /p   PROJ_DIR= <temp.bat
@del temp.bat
::@echo %PROJ_DIR%
@FOR %%i IN (%PROJ_DIR%) DO @SET PROJ_NAME=%%~nxi
::@echo project name: %PROJ_NAME%

::echo %1
@if "%1"=="arm" set ARCH_DIR="armle"
@if "%1"=="armv7" set ARCH_DIR="armle-v7"
@if "%1"=="i386" set ARCH_DIR="x86"
::@echo %ARCH_DIR%

qmake -project
@echo QT += core gui script declarative %QT_MODULES% >>%PROJ_NAME%.pro
@echo OBJECTS_DIR += %ARCH_DIR% >>%PROJ_NAME%.pro
@echo TARGET = %ARCH_DIR%/%PROJ_NAME% >>%PROJ_NAME%.pro
@echo QMAKE_CXXFLAGS += %QT_CXXFLAGS% >>%PROJ_NAME%.pro
qmake -spec unsupported/qws/qnx-%1-g++ %PROJ_NAME%.pro
@goto exit

:help
@echo .
@echo %0 runs both "qmake -project" and "qmake" resulting in a Makefile
@echo .
@echo First parameter must be desired architecture - i386 or arm or armv7
@echo .
@echo .

:exit
```