# Qt HMI using QML

## Tutorial: Creating a QNX UI with Qt Declarative

*Dennis Kelly, East/SE US FAE, QNX Software Systems*
*dkelly@qnx.com*
18-May-2011

## Introduction

Qt is a platform independent software toolkit for building UI's. Since it is implemented in native C++, Qt interfaces are quite efficient and run well on sub-desktop processors. Qt supports "fluid" animation of visual objects, making it a good choice for today's more demanding look-and-feel requirements.

Traditional Qt requires an "imperative" language approach where objects are created sequentially in a hierarchy directly from code. This is very similar to the approach taken for UI's in Windows or Java.

With version 4.7, Qt added a "declarative" language model. Writing a declarative UI is much like laying out a webpage with HTML – except the "markup language" for Qt Declarative is called "QML". As with webpages, user interaction is completely event-driven.

To understand how QML interacts with C++ code, one must understand the Qt concepts of "signals" and "slots".

"**Signals**" are emitted by a C++ object to indicate the occurrence of a particular event to QML. "**Slots**" are methods in a C++ object which QML can call directly. The QML language element "Connections { }" allows signals from C++ objects to be received and handled by script a QML object.

So in summary, QML can call any C++ object method (slot) directly in response to user input (button press, etc.). And, it can receive any asynchronous event (signal) from a C++ object simply by providing a handler in QML. This paradigm provides all we need to allow native QNX C++ code to control devices and provide updates to a UI.

Note that in this model, C++ objects generate signals whether or not there are any QML handler(s) attached. This is similar to a "publish/subscribe" paradigm - it helps keep the device logic cleanly separated from HMI logic, improving modularity of the code.

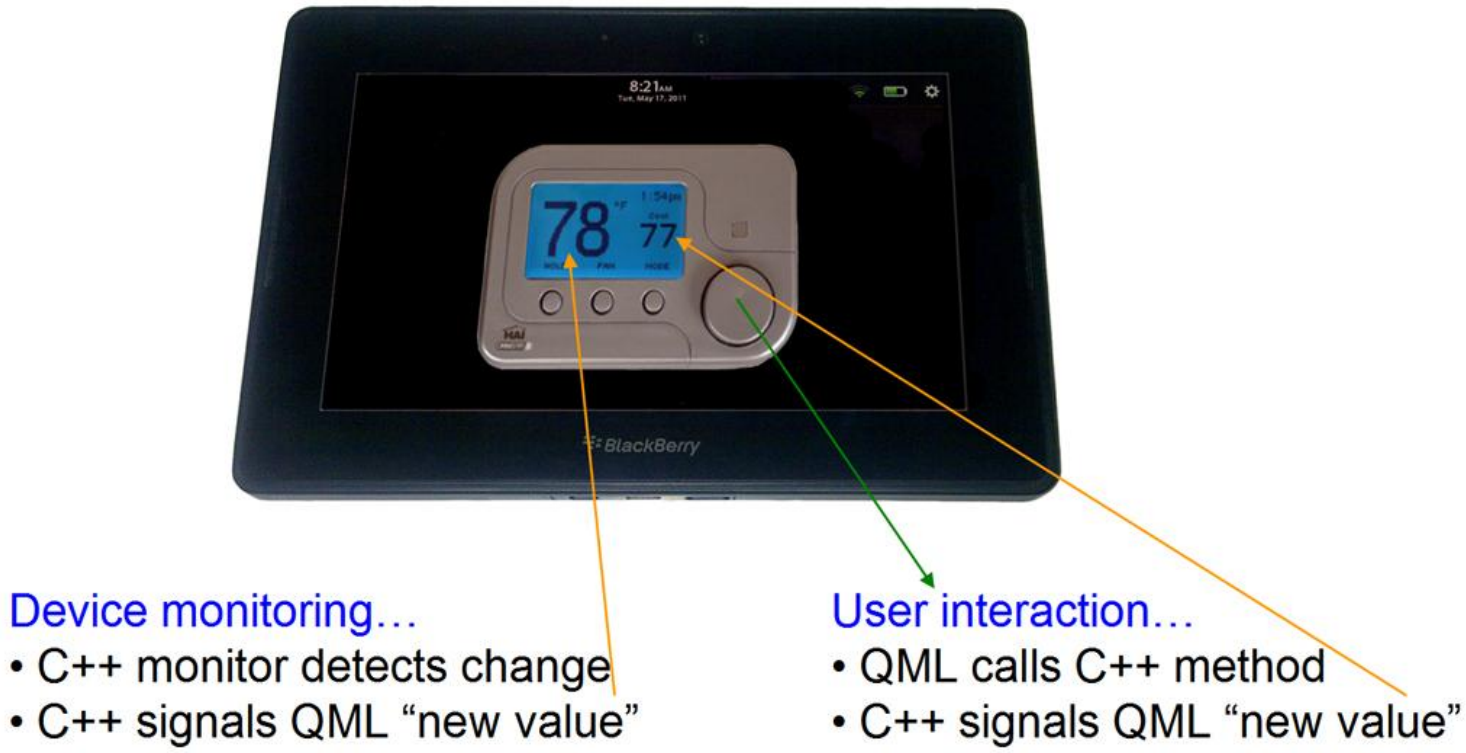


**Figure 1 – Illustration showing control flow between Qt QML UI logic and C++ device monitor/control logic. QML calls C++ methods directly on user events. C++ logic can signal QML asynchronously when values should be updated.**

## Description

The example presented consists of a single button and two text panels.

- **In the lower right, a count of button presses is displayed.** This count is driven indirectly by user interaction – after the C++ method is notified that a button was pushed, C++ signals back to QML with the new count value.

- **In the upper left, the current second from the system clock is displayed.** This is driven by a timer in the C++ object and is completely independent of user interaction.


**Figure 2 – Handling user interactions and quantitative updates from background code are the two fundamental operations of all device-oriented HMI's.**

The button is animated when "pressed" by changing the opacity of the "button-down" .png image which overlays the "button-up .png image. Initially, the "button-down" image is fully transparent, but when pressed, it becomes fully opaque. Note that in a declarative language, it is typical that screen regions declared **later** appear **"in front"** of those declared earlier.

Qt can provide a visual transition time for the opacity change, as it can for a series of other transitions – like position.

## Getting Started

One advantage of Qt is that the code is transportable between operating systems and processors. Recompiling for a particular target should generate an equivalent HMI.

Another Qt advantage is the availability of tools. An IDE environment called **Qt Designer** is available for Windows and Linux. (This project was tested in the Qt Designer version 2.0.1 for Windows before moving it to QNX.)

With that said, the best way to get started with this project is to

- download and start QT Designer,
- unzip the archive cEasy_win.zip to a directory under the Qt installation, like "examples"
    - this should create folders cEasy and cEasy-build-desktop
- open the project file "cEasy\cEasy.pro"

The IDE will allow you to open/view all project files. These include the C++ source and header files, PLUS the .qml file and image files it references.

Note that the Windows installation of Qt Designer installs mingw and a gnu C-compiler to allow you to build and run on your desktop.

When you build the project in Windows (or in Linux), you can also run the project directly from the IDE. You should see the HMI function as described on your desktop. This is possible because no QNX-specific API's are referenced in the example.

## Next Steps

Now that you are familiar with the operation of the UI and the files which make up the project, it is time to build the UI for your QNX target.

This description will assume a "self-hosted" QNX 6.5 development environment (as installed from a CD image) which *will also serve as the runtime target.*

**Install Qt for QNX on your QNX 6.5 machine.** Download the latest Qt archive from:

http://community.qnx.com/sf/frs/do/viewRelease/projects.qt/frs.binpkg.targets

For the 4.7.1 binary package (i.e. qt_qnx_2011-02-24b.zip) you can follow these steps:

1) unzip the archive to /tmp and locate these two files:
    a. qt_qnx_host_qnx6_x86.tar.gz
    b. qt_qnx_targest.tar.gz

2) install tools and target binaries for development
    a. # tar xvf qt_qnx_host_qnx6_x86.tar.gz -C /usr/qnx650
    b. # tar xvf qt_qnx_targets.tar.gz -C /usr/qnx650

3) install target binaries so your system is also a target
    a. # tar xvf qt_qnt_targets.tar.gz -C /tmp
    b. # cd /tmp/target/qnx6
    c. # cp -r usr/* /usr
    d. # cd /tmp/target/qnx6/x86
    e. # cp -r usr/* /usr

4) Edit /root/.profile, and add these lines
    a. export QWS_DISPLAY=qnx
    b. export QWS_KEYBOARD=qnx
    c. export QWS_MOUSE_PROTO=qnx
    d. export XDG_CONFIG_HOME=/root
    e. export QMAKESPEC=unsupported/qws/qnx-i386-g++

    *Note: QMAKESPEC is specific to x86 !*

5) You must logout and login for the new .profile to take effect.

**Install the project archive**, once installation of Qt has been completed, you will find that the .cpp files, .h files and content folder are identical to that of the Windows archive.

*Exit photon to command line for this part*!

1) # cd /usr/qnx650/target/qnx6/x86/usr/lib/qt4/examples

2) # tar -xf /fs/usb0/cEasy_qnx6.tar
   - this should create folder eEasy
   - assumes the tar file is on a usbstick

3) # cd cEasy

4) # make clean
   # make
   - should complete with no errors
   - should have ./cEasy

5) # ./cEasy -qws
   - should display the UI with timer running
   - you will NOT be able to put the button
   - ctrl-C to return to text mode.

6) From a workstation, telnet into your development system. (You may alternatively attach a serial cable to a terminal emulator.) Login as root to a shell prompt.

7) # cd /usr/qnx650/target/qnx6/x86/usr/lib/qt4/examples/cEasy from the external terminal.

8) # /usr/photon/bin/devi-hid -Pr kbd mouse
   - At this point the PC console no longer works – which is why you enter the command from an external terminal.

9) # ../cEasy -qws &
   - now the button can be activated with the mouse and the counter should be operational.

Remember, there is no QNX-specific code in this project – which is why the identical files from Windows build and run on QNX. Of course to implement a meaningful QNX project, you will likely have to add some QNX-specific code. Once you do, you will no longer be able to build/run the UI on Windows without commenting out the QNX-specific code.

## Designing for Success – *Some advice*

HMI code should be loosely coupled to the process code. Experience shows there are reasons for this:

- UI's are subject to frequent "subjective" changes
- UI's often need to be "re-skinned"
- UI's need to run remotely utilizing differing technologies

As long as your "process" code is independent – detached from the UI code – then UI "churn" should require only incremental changes to your process code. And if the system suddenly gains a "remote" requirement, you only need to figure how a transport the process-to-UI communications.

Consider using QNX pps to communicate between your process code and the UI methods. Pps provides asynchronous notification to your UI objects while acting as a "middleman", further isolating your components.

*Thanks to Kunal at -*

http://kunalmaemo.blogspot.com/2011/04/signal-slot-connection-with-qml-and-qt.html

*for providing an example of the proper syntax for the QML "Connections { }" element. The syntax changed in 4.7 and the docs are woefully lacking at this point.*