Qt and QNX pps

Tutorial: UI Integration with a pps Datasource

Dennis Kelly, East/SE US FAE, QNX Software Systems
dkelly@qnx.com
11-June-2011

Qt Introduction

Qt is a platform independent software toolkit for building UI's. Since it is implemented in native C++, Qt offers interfaces that provide efficient native system interaction 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.

This example utilizes the Qt "declarative" language model added with version 4.7. Writing a declarative UI is much like laying out a webpage using a markup language called "QML".

QML interaction with C++ was covered in detail in a previous tutorial. The main concepts are:

- "**Signals**" are emitted by C++ objects to indicate particular events directly to a QML script

- "**Slots**" are C++ object methods which a QML script can call directly.

To reiterate, QML can call any C++ object method (slot) directly in response to user input (button press, etc.). And, it can react to any asynchronous event (signal) from a C++ object simply by providing a handler in QML

Together, slots and signals provide all the interaction we need to allow native QNX C++ code to control devices and provide updates to a UI.

pps Introduction

"Persistent Publish/Subscribe" (pps) is a QNX Neutrino io-manager daemon which allows data-sources to "publish" data without implementing ***connections*** to specific data-consumers. Published data can optionally "persist" between reboots.

The publish/subscribe paradigm contrasts with say a webserver, which can only "publish" data if one or more clients actually ***connect*** *in the manner prescribed by the server.* Publish/subscribe systems foster data delivery to many different types of subscribers.

With QNX pps, subscriber applications only need to access an object in the filesystem. Via the object, subscribers receive event notification when additional *(or changed)* data is available – without the inefficiencies of polling.

Application Description

The UI for this example represents a typical wall thermostat shown here. Full-size is 640x480 pixels.

Figure 1 – Representation of a home thermostat with typical user interaction buttons.

- In lower right is the **target temperature.** This value is driven by user interaction with the up/down buttons below it.
- In lower left are the **MODE** button (HEAT,COOL,OFF) and the **FAN** button (AUTO,ON).
- in upper portion, the **INSIDE** temperature, **OUTSIDE** temperature and **time** are displayed. These are all driven by a thermodynamics simulation which takes the user settings into account.

When the heat (or cooling) unit is "running", a series of 0 to 8 "dots" above the MODE and FAN buttons are animated. This animation is local to the Qt code – the server only indicates whether the system is running or not.

The environmental simulation runs in “time lapse” mode so the viewer can more quickly see how user interactions and outside conditions impact the interior temperature.

The visible buttons are animated in the Qt code using graphic images. Interestingly enough, Qt has no built-in “button” objects! However, the amount of code required to implement a custom button is not large, as you can see in the source.

When you run qTstat on QNX, you will notice that the application appears “frameless” – unlike other example Qt applications. Assuming you are going to deliver a single-purpose embedded UI, you will not want your application exposing a windowframe. Two lines of code in main,cpp suppress the windowframe.

```
Qt::WindowFlags flags = Qt::FramelessWindowHint;
mainWidget.setWindowFlags(flags);
```

The second line should precede the line containing `mainWidget.show();`

Understanding the Data Source

The thermodynamics simulator is a C++ library object which was wriiten for a different project. It utilizes an open source package called tinyXML to read/write an XML document which serves as the database for the simulation. Since I wanted to utilize an existing simulation object, logic to convert XML-to-pps was added to complete the server (tdpps). The processing schematic is shown below.

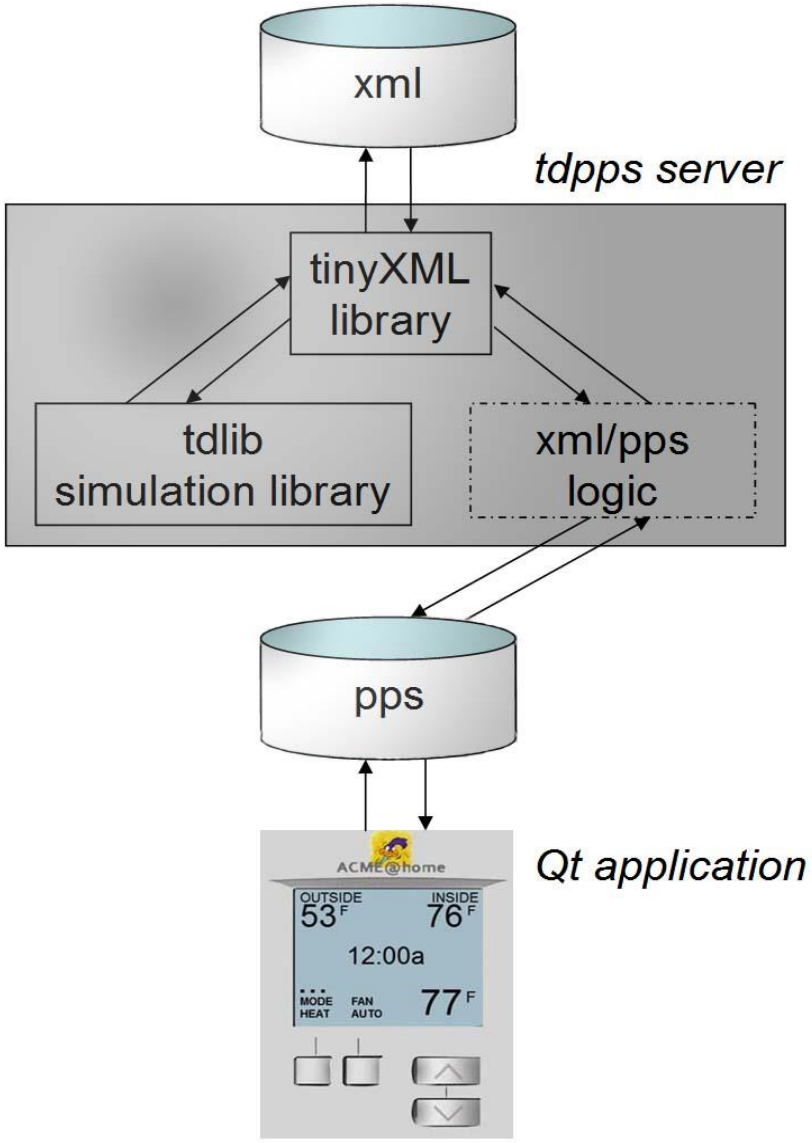

Figure 2 – Schematic showing relationship of objects. "xml" and "pps" reside in the filesystem.

The tdpps server publishes a pps file named “outputs” while subscribing to a pps file named “inputs”. The Qt application does the opposite - it subscribes to “outputs” and publishes “inputs”. These two files appear in the filesystem as

```
/pps/tdpps/inputs  … sample contents
   @inputs
      btn1::
      btn2::event
      btn3::
      btn4::

/pps/tdpps/outputs … sample contents
   @outputs
      ts_mode::HEAT
      ts_fan::AUTO
      ts_temp::68
      td_temp_in_::66.0
      td_temp_in::66
      td_temp_out_::60.0
      td_temp_out::60
      td_sys::OFF
      td_time_::1700
      td_time::05:00p
```

In the Qt application, all interaction with pps occurs in a C++ object named ‘tstatData’.

- Qt is “signaled” by tstatData object when pps “outputs” object changes, and
- Qt calls “slots” (methods) inside the tstatData object to send user events to pps “inputs” object.

C++ source for tstatData is provided as part of the Qt application.

The “tdpps” server object is considered to be a “blackbox” for this discussion. *It is provided in X86 binary form only.*

Due to the nature of pps, when tdpps is running (“tdpps –i /tmp”) you can “see” the progress of the simulation with the following command:

```
# cat /pps/tdpps/outputs?wait
```

All the changes to the pps object “outputs” will be streamed to the console.

While you can run ttpps on a physical disk, you will want to consider creating a ramdisk as it heavily accesses the database xml file.

```
# devb-ram ram capacity=8192
# dinit -h /dev/hd1t77   <<<< careful that /dev/hd1 is ramdisk!!!
# mount /dev/hd1t77 /tmp2
# ./bin/tdpps -I /tmp2 &
```

Alternate Designs

Since Qt programs are just C++ code, it possible that we *could* have linked the thermodynamics object directly to the UI program – bypassing the interprocess communication link provided by pps. Engineers will probably always debate "monolithic" versus "interprocess" designs for projects. Here are a few thoughts.

Monolithic advantages

- Consistent data type checking across entire application
- Efficiencies introduced by fewer context switches

Interprocess advantages

- 'Divide and conquer' strategy for implementation and debugging
- Flexibility to substitute s new UI with minimal impact on 'engine'
- Flexibility to change UI technology, i.e use Java or Adobe Flash
- Generally easier to add a remote interface when the 'engine' is independent of the UI

Getting Started

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

The UI program without any data interaction was implemented initially in **Qt Designer v2.0.1** for Windows. *A zip file for Qt Designer is included in the tar file in directory /qTstat/win32… however, there is no datasource!*

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++
    f. *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/qTstat.tar
   - this should create folder qTstat
   - assumes the tar file is on a usbstick

3) # cd qTstat

4) # make clean
   # make
   - should complete with no errors – but a few warnings
   - should have ./qTstat

5) # ./qTstat –qws
   - should display the UI as a first test
   - you will NOT be able to push the buttons
   - 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/qTstat from the external terminal.

8) Start "pps" daemon if not running. Check for /dev/pps.
   # /usr/sbin/pps &

9) Start tdpps data source
   # ./bin/tdpps –i /tmp &

10) # /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.

11) # ./qTstat –qws &
    - now the buttons can be activated with the mouse and data fields should be updating.

pps Implementation Details

Since the C++ code in the qTstat application is both a **pps subscriber** (for displayed data) and a **pps publisher** (for user inputs), the source provided shows both sides of a pps implementation. Assume pps directory is /pps.

To publish a datasource

- create a subdirectory with a unique name, in this case /pps/tdpps
- open the file object, creating if it does not exist

```
open("/pps/tdpps/inputs",
      O_WRONLY | O_CREATE | O_TRUNC,
      S_IRUSR | S_IWUSR);
```

- write out a buffer with the entire file contents in one write() operation
- leave the file open for frequently changing data (or reopen each time)
- each time data changes, write out the entire file contents in one write() operation
- it is **not** necessary to lseek(0,SEEK_SET) when updating contents – the new data replaces the old in the object
- at conclusion, simply close() the file descriptor if it was left open

The "inputs" file in the example above is used to signify UI button pushes. Text string "event" is appended to the appropriate "btn" line in the file and the buffer is written. The "event" text is removed and the buffer is written a second time. This causes tdppc (the subscriber) to wake twice – but only once will if find and process an event.

To subscribe to a datasource

- subscriber must 'know' the name of the publisher's file object
- open the file object with a special syntax … "?wait"

```
open("/pps/tdpps/outputs?wait",
      O_RDONLY | O_TEXT );
```

- call read()
- each time the publisher calls write(), the read() will wake
- process the fresh data
- loop back to the read() call and wait for more new data

The "outputs" file in the example above is output by the tdpps simulation and used by Qt to populate the display.

See "QNX PPS Developer's Guide" in Momentics Help for more details.

Conclusion

A Qt UI project with an external process datasource has been presented. The project shows how a Qt application can receive asynchronous data updates from the datasource, and how Qt can feedback asynchronous user events to the datasource.

The event-driven interprocess exchanges utilize the QNX publish/subscribe mechanism (pps). In this example, the Qt process is both subscriber (simulation data) and publisher (UI events). Likewise the datasource process is also publisher (simulation data) and subscriber (UI events).

The cpu utilization by the UI on an 1.6Ghz Atom processor was 4-7% with rapidly changing numbers. This assumes gma9xx video driver and tdpps running on a ramdisk. The cpu utilization by the simulation process, including publishing to pps, was negligible.

The example presented is typical of the interface required for many dedicated-use devices. It illustrates the suitability of Qt and pps for lightweight UI implementation.