Embedded's gone cute

The need for next-gen UIs has finally reached the embedded world. Owing to increasing demand, vendors have been looking for alternative technologies to enable modern user interfaces on their products, and have been finding Qt (pronounced 'cute') is the perfect fit.

The presence of somewhat elaborate GUIs on embedded platforms is nothing new. As a matter of fact, you can find embedded GUIs in a variety of devices and appliances: vending machines, tractors, ATMs, machine panels – this list could grow indefinitely. Given the tight coupling between hardware and software in the embedded space, these GUIs are in general implemented using whatever tools and toolkits that have been provided by the hardware manufacturer or its partners. You can imagine that, while the people behind the UI of an airplane infotainment system might make an effort to make their GUI attractive to the end-user, eye candy was probably never a concern for those designing UIs for subway control systems, leading to the birth of state-of-the-art soviet style interfaces (Motif, anyone?).

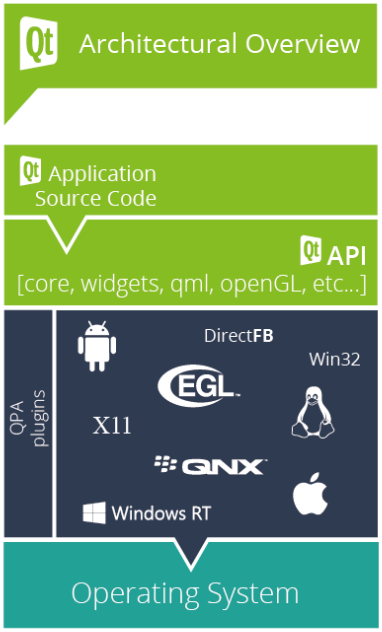
The iPhone revolution

After innumerous attempts of introducing touchscreen-based handheld devices, Apple finally nailed it with the introduction of the original iPhone back in 2007. That was the very first embedded device to run that type of modern gesture-based GUI that would pave its way to ubiquity. Several years later, the iPhone multi-touch approach became the default choice for mobile and embedded devices, quickly spreading beyond the borders of mainstream consumer products, knocking down the iron curtain of the old-school graphical interfaces. A variety of embedded systems manufacturers started to work on implementing modern GUIs into their products, and consequently, the need for means to achieve that arose. There were few alternatives, some proprietary, some originating from the free software world.

Enter Qt

Among those alternatives is Qt. It is a cross-platform C++ application framework that was born in 1991, initially as a GUI toolkit providing common widgets. Qt has been growing at a steady pace, and is now at its 5th major release. During all those years, Qt's has expanded its domain from a GUI widget toolkit to become a general-purpose C++ framework, implementing functionality beyond GUI classes. At the time of this writing, Qt finds itself at version 5.4.1, natively supporting dozens of platforms, including QNX versions 6.5 and 6.6, Linux (X11 and EGL), Windows CE, Windows, Android, iOS, Wayland, among others. As a consequence of its domain growth, Qt has been split in several modules, each implementing a different group of functionalities, a feature that also opens the possibility for vendors to write their own modules to Qt if they wish to. One of them is called *Qt Quick*. It provides the infrastructure for implementing modern gestured-based and fluid graphical interfaces, using both C++ and Qt's own declarative language called *QML*. Qt also ships with its own IDE, called *QtCreator*. But before we dive into that, let's take a look on how Qt is architectured.

Architecture



As depicted above, Qt is made of several orthogonal modules. Many of those modules can be disabled, and several Qt features can be turned off during compile time, minimizing the application footprint. Some are worth an additional explanation:

QPA

QPA is an acronym for *Qt Platform Abstraction*. This is the layer that promotes the integration between Qt and the underlying platform. Each platform abstraction is then implemented in the form of a QPA plugin, that is loaded at run-time. These platform plugins implement a common QPA interface, which allows Qt to query for platform capabilities, request raster and OpenGL surfaces (if supported), deal with and translate native platform events, etc. Thus, porting Qt to a new platform or operating system is mostly a matter of writing a QPA plugin for that platform. There is no need to become acquainted with all the internals of Qt. Since Qt has been already ported to several embedded platforms, including popular ones like Windows CE, embedded Linux (with multiple backends, from directfb to OpenGL and also Wayland) and even QNX, there are high chances you won't even need to write your own QPA plugin when deploying Qt to your platform.

QtCore

As the name says, QtCore is the Qt module that implements core classes and functionality, including event handling and dispatching, as well as classes for, among others, threading support, strings, internationalization, XML handling, text and data streams and template-based containers (QMap, QList, QHash, QlinkedList, QVector, QStack and a dozen more). Like many C++ classes in Qt, these container classes use implicit data sharing and *copy-on-write* to maximize resource usage and performance.

As stated above, one of QtCore's jobs is to implement event handling and dispatching. This is worth mentioning, because this is part of a mechanism that is at the heart of Qt: ***signal and slots***.

Signals and Slots

Signals and slots are one of the most important features of Qt. It provides an alternative over the old-school callbacks. Each Qt object (i.e. all classes descending from the **QObject** class) has the ability of "emitting" signals that can optionally be connected to one or more slots. Slots are simply class methods, the only difference being that they can be "connected" to signals. For instance, the *QPushButton* class emits a signal conveniently named *clicked* whenever the button is, well, clicked. We can then proceed to write the following piece of code:

```
QPushButton *button = new QPushButton("Click me!");
connect(myButton, &QPushButton::clicked, this,
            &MyClasss::handleButtonClick);
```

The event loop

Even though it is possible to write Qt applications without an event loop, in most cases there is at least one event loop running. Here is the typical main file of a Qt application running an event loop:

```
#include <QApplication>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
     QApplication app(argc, argv);
     MainWindow w;
     w.show();
     return app.exec(); // start event loop

}
```

On the above piece of code, we use the QApplication class to perform the Qt initialization, including querying and loading available platform plugins, setting up the display, and more. After we've created our MainWindow widget, all we have to do is calling QGuiApplication::exec() to start the event loop. The MainWindow will then start receiving events (including platform events) and reacting to them. Even though we are not showing it here, it is perfectly possible to install an event handler and even inject events on the event loop.

QtWidgets

We saw above a class named QPushButton. This class belongs to a module called QtWidgets. This is the original Qt widgets implementation that evolved together with Qt. It is still being maintained, and has been marked as feature complete. Because of the paradigm shift regarding GUIs that happened in recent years, a new approach focused on modern GUIs has been developed, in addition to QtWidgets. It is called QtQuick.

QtQuick

QtQuick is the solution provided by Qt to modern GUIs. It uses Qt's own declarative language, called QML, to describe UIs. QML is itself based on JavaScript, which means that JavaScript code is also allowed. QtQuick is shipped with the QtDeclarative module.

Other modules

Qt offers modules for a variety of functionality, including, but not limited to, D-BUS, bluetooth, localization, NFC, serial port handling, etc. As mentioned, these modules can be cherry-picked into the final Qt deployment at the developer's convenience, which can be extremely helpful especially for low-footprint embedded targets.

Declarative is the way to go

Together with the advent of modern UIs, the usage of declarative languages for implementing them started to gain ground. Big players, such as Microsoft with their Windows Phone platform, Google's Android, Apple's iOS, as well as BlackBerry offer declarative frameworks for UI design on their platforms, and as you already know, QtQuick/QML is Qt's answer to that. Because it was designed with modern UIs in mind, QtQuick makes it a piece of (preferably chocolate) cake to implement transition animations, gestures and all the shiny wobbly things distinguishing these new UIs from their utilitarian forefathers. The screenshot pictured at the left shows a gesture-enabled application that was written almost entirely using QML and JavaScript (if you are interested on seeing it in action, it is part of the Qt demo package).

QML hierachy and property binding

```
1  import QtQuick 2.4
2  import QtQuick.Window 2.2
3
4  Window {
5      visible: true
6
7      Rectangle {
8          width: parent.width
9          height: parent.height
10         color: "#ffb37c"
11
12         MouseArea {
13             id: mouseArea
14             anchors.fill: parent
15         }
16
17         Text {
18             anchors.centerIn: parent
19             text: "Hello RTC!"
20         }
21     }
22 }
```

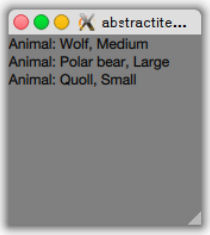
The screenshot above shows a very simple example of what QML code looks like. Notice that QML items are part of a hierarchy. In this particular example, there is a Text and a MouseArea item (a MouseArea item responds to mouse events or gestures) inside a Rectangle, that is itself contained in the root Window item. In order to make sure that the rectangle is always adjusted to the window whenever someone resizes it, we need to *bind* its *width* and *height* properties to those of its parent (i.e. the window). Unlike a normal value assignment, bounded properties will always update their values according to the properties they are bound to. For our Rectangle, it just means that the its width and follows the parent with and height. Moreover, we use *anchors* to keep the text centered. *Anchors* provide an alternative way for specifying an item position. In addition to that, QtQuick also offers layouting functionality as yet another alternative for item positioning.

A cute frontend with a C++ backend

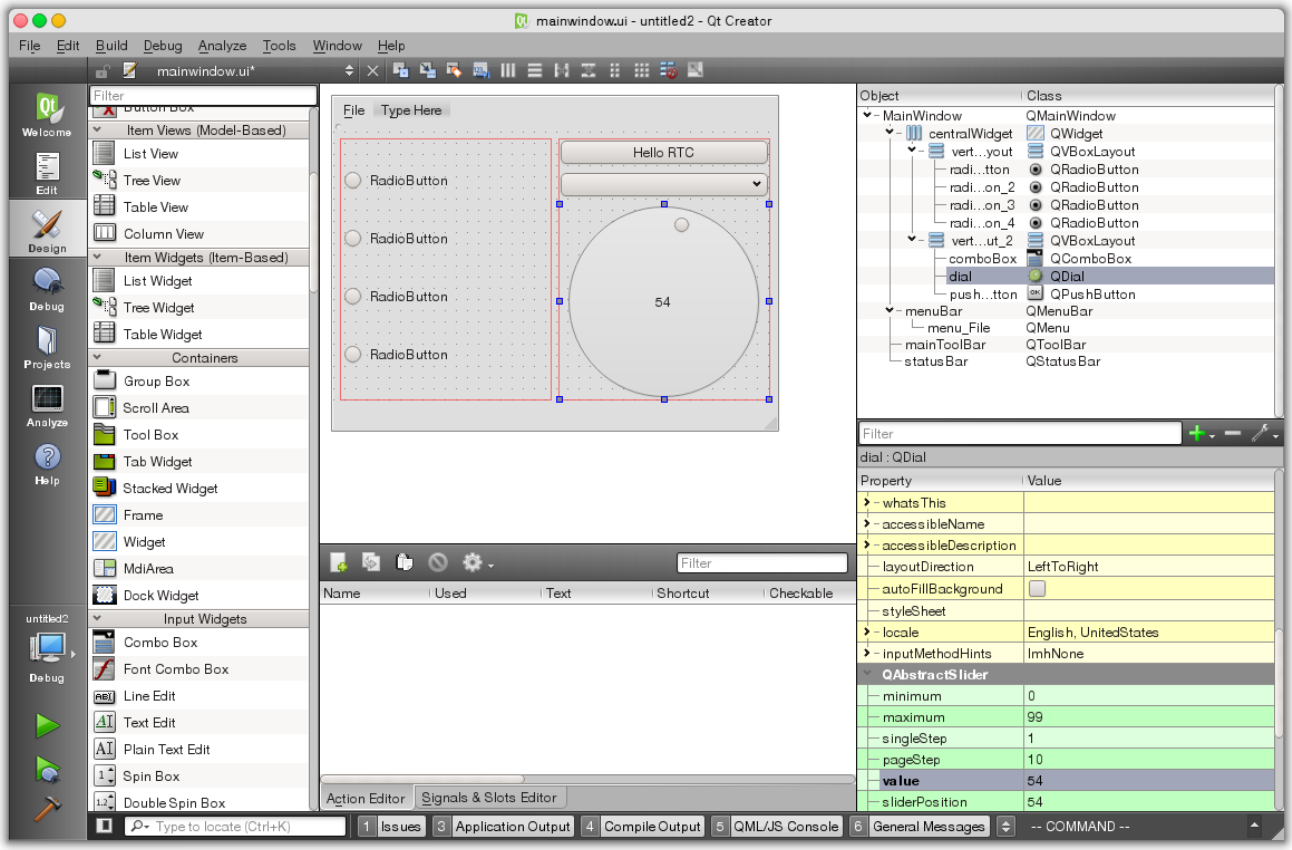
Despite all the nice features and JavaScript support, it is often the case where you will need to interface with C++ code, maybe to connect to a database or even to talk to the CAN bus. Either way, Qt offers mechanisms to allow for such an integration and, as a side-effect, to help keep the UI layer decoupled from the rest of the system. This is achieved by allowing the C++ to export properties into the QML context (or even exporting an entire alternate context). These properties can naturally be modified at the QML layer (and vice-versa) and even be bound to other properties. This functions as a very thin layer between the QML and C++ codes. The example below exports the “AnimalModel” as a property in the QML context. A ListView item is then used to render the model.

```cpp
#include "model.h"

#include <QGuiApplication>
#include <qqmlengine.h>
#include <qqmlcontext.h>
#include <qqml.h>
#include <QtQuick/qquickitem.h>
#include <QtQuick/qquickview.h>

int main(int argc, char ** argv)
{
    QGuiApplication app(argc, argv);

    AnimalModel model;
    model.addAnimal(Animal("Wolf", "Medium"));
    model.addAnimal(Animal("Polar bear", "Large"));
    model.addAnimal(Animal("Quoll", "Small"));

    QQuickView view;
    view.setResizeMode(QQuickView::SizeRootObjectToView);
    QQmlContext *ctxt = view.rootContext();
    ctxt->setContextProperty("myModel", &model);

    view.setSource(QUrl("qrc:view.qml"));
    view.show();

    return app.exec();
}
```

```qml
import QtQuick 2.3

Rectangle {
    width: 200;
    height: 200;
    color: "grey";

    ListView {
        anchors.fill: parent
        model: myModel
        delegate: Text { text: "Animal: " + type + ", " + size }
    }
}
```



Tooling



Qt has its own IDE, called *QtCreator*. While you are free to choose whether you want to use it, QtCreator does provide a bunch of useful functionality that makes our lives easier. Namely, it includes an advanced code editor with auto-completion, syntax highlighting, and a lot of other features, that even includes a *vi* mode. Apart from the text editor, there is a form editor to be used on QtWidget projects (depicted on the screenshot above), integrated debugger (for C++ and QML), introspection support and a QML designer. It also provides integrated functionality for seamless deployment for targets running embedded Linux, QNX, Blackberry OS, Android and iOS. For version control, it supports git, baazar and mercurial. Last, but not least, because QtCreator is scriptable and supports plugins, it is easy (and actually common practice among vendors that already adopted Qt) to extend and integrate QtCreator with third-party tools and SDKs.

To the infinity and beyond

Qt offers a wide range of APIs and functionality, but most importantly: it is proven technology that has been around for more that a decade and continues to evolve, fostered both by the free-software community and by companies like Intel, QNX, Garmin, Ford, KDAB, The Qt Company and many more people and entities that share a mutual interest on keeping Qt bleeding-fast and rock-solid.