The Qt Company

Whitepaper

Building the Internet of Things and How Qt Can Help

```
import QtQuick.Controls 1.4
import QtLocation 5.6
import QtPositioning 5.5
import "map"
import "menus"
import "helper.js" as Helper

ApplicationWindow {
    id: appWindow
    property variant map
    property variant minimap
    property variant parameters

    //defaults
    //! [routecoordinate]
    property variant fromCoordinate: QtPositioning.coordinate(59.9483, 10.7695)
    property variant toCoordinate: QtPositioning.coordinate(59.9645, 10.671)
    //! [routecoordinate]

    function createMap(provider)
    {
        var plugin

        if (parameters && parameters.length>0)
            plugin = Qt.createQmlObject ('import QtLocation 5.3; Plugin{ name:"' + provider + '"; parameters: appWindow.para
        else
            plugin = Qt.createQmlObject ('import QtLocation 5.3; Plugin{ name:"' + provider + '"}', appWindow)
```

```
QML module does not contain information about components contained in plugins.

Module path: /Users/samulinivala/Qt5.6.0/5.6/clang_64/qml/Qt/labs/controls/universal
See "Using QML Modules with Plugins" in the documentation.

Automatic type dump of QML module failed.
Errors:
"/Users/samulinivala/Qt5.6.0/5.6/clang_64/bin/qmlplugindump" returned exit code 3.
Arguments: -nonrelocatable Qt.labs.controls.universal 1.0 /Users/samulinivala/Qt5.6.0/5.6/clang_64/qml
QQmlComponent: Component is not ready
file:///Users/samulinivala/Qt5.6.0/5.6/clang_64/qml/loaddependencies.qml:6:1: module "Qt.labs.controls.universal.impl" is not installed


Project ERROR: Xcode not set up properly. You may need to confirm the license agreement by running /usr/bin/xcodebuild.
Error while parsing file /Users/samulinivala/Qt5.6.0/Examples/Qt-5.6/qtlocation/location/mapviewer/mapviewer.pro. Giving up.
```

The Qt
Company

Contents

Building the Internet of Things and How Qt Can Help

The term, Internet of Things (or IoT), is everywhere. For people wanting to impress others, it's the latest buzzword to include in a conversation in order to be seen as relevant. For science fiction romantics, it's an image of dust-sized computers implanted everywhere in everyday objects. For most developers though, the Internet of Things is just a newer, more fashionable term for the old industry workhorse: the connected embedded system. After all, haven't we been building IoT-like devices for decades? Yes and no.

Central to most definitions of IoT devices is the fact that they are embedded systems that are often (but not always) mobile and use M2M — in other words, wandering gadgets communicating machine-to-machine. Of course, these attributes already apply to a large number of embedded devices. However, the IoT promise is that always-on communication will give these devices the information they need to act smarter. This step of imbuing every-day objects with rudimentary intelligence and communication skills gives us a wide array of technological aides: sensor-studded biometric clothing, self-scheduling shipping drones, auto-monitoring homes, freshness-reporting groceries, automatic parking meters, self-diagnosing agricultural crops — the list goes on.

Realizing this vision of IoT requires computers to continue becoming smaller, smarter, and more connected. While everyone seems to understand this requires a hardware transformation, few people are talking about the significant change that's required in software. Adding intelligence to everyday objects while ensuring both human-to-machine (H2M) and M2M conversations are more intuitive and natural requires complex software, and lots of it. This in turn, places a number of requirements on how to develop IoT software.

IoT's Software Requirements

Because there are so many technologies that could be applied to creating intelligent, connected systems, it can be tough to know where to begin. Let's start by figuring out what constitutes an ideal software framework — or at least what are the basic requirements.

1) Powerful

We can't fake smarts without some computational horsepower. We need a software toolkit that is up to the task.

2) Optimal

Shrinking processors and boards won't give us the luxury of boundless resources. We need something that is powerful but efficient.

3) Connected

We'll be handling all kinds of communication so we need flexible connectivity options to let us either push smarts from the network's edge into the cloud, get data from a sensor net, or anything between. Our chosen tools need to easily support a host of protocols, stacks, and wireless technologies.

4) Rapid Development

To fail fast, evolve our products, and support IoT's quick lifecycle churn, we need to develop software quickly and reliably. Development should be as simple as possible with modern tools and IDEs.

5) Cross-platform

We'll want to share our codebase across as many devices as possible from desktop to cloud server, and from headless sensor to embedded UX. Picking a development tool that minimizes porting effort is essential.

6) User Interface

Of course, many IoT devices won't need a UX. However plenty of connected gadgets will. Splitting the development toolchain along a headless versus headed capability isn't ideal; if we can find something that works well for all situations that would be best.

7) Sharing

We don't want to reinvent the wheel. We want an open, active, and global development community with others to learn from.

8) Secure

Hackers are everywhere: our devices need to be tamper-proof and resilient to hacking.

9) Reliable

We can't have a host of new "smart" devices around us that function erratically or continually need reboots: they have to be reliable. While any language and tool should be able to create reliable software, we'll look at the ease of external validation or certification as our measure. That may not apply everywhere but there will be IoT industries where this matters.

10) Stable

Finally, we know that software continues to change — the model where device functionality is completely fixed at the time it leaves the factory is becoming obsolete. Therefore, we'll not only want a way to update software in the field but we'll want to make sure that the software tools we used to build our devices stays stable, even as those tools are refreshed.

Now that we've narrowed our parameters, let's take a look at today's popular embedded languages with an assortment of frameworks to see how they stack up.

	C	C++/STL	C++/Boost	C++/Qt	C#/.NET	Java/ Android	HTML5/ Cordova
Powerful	–	✓	✓	✓	✓	✓	–
Optimal	✓	✓	✓	✓	–	–	–
Connected	–	–	✓	✓	✓	✓	✓
Rapid development	–	–	–	✓	✓	✓	✓
Cross-platform	✓	✓	✓	✓	–	–	✓
User interface	–	–	–	✓	✓	✓	✓
Sharing	–	✓	✓	✓	–	✓	✓
Secure	–	✓	✓	✓	✓	–	–
Reliable	✓	✓	✓	✓	–	–	–
Stable	✓	✓	✓	✓	✓	✓	–

C

C may have lost its crown for development on desktops but it's still quite relevant in the embedded world. However, C's biggest problem is that the frontier has moved elsewhere: developments with IDEs, RAD, frameworks, and communities are pretty quiet in the C world.

Score: 4/10

C#/.NET

C# and .NET does well in a number of ways but isn't quite as efficient, compact, or optimized as C++. Despite Mono, it also has a hard time claiming cross-platform compatibility. The community is active but perhaps not quite as fervent as the open source options.

Score: 6/10

HTML5/Cordova

HTML5 (or JavaScript + HTML + CSS) under Apache Cordova — while ranking great on many of the cross-platform and development options — is far too big and suboptimal for most IoT systems. Not to mention that most HTML5 frameworks are frequently in flux — great for continually picking up new features but not at all great if you're trying to build a stable code base that will be maintained with future updates.

Score: 5/10

Java/Android

Android is quite capable but requires a rather heavyweight environment. However, where it really fails is in cross platform. Android apps run on Android and nothing else, which limits where you'll be able to take your code afterwards.

Score: 6/10

C++/STL

Certainly C++ does well for being both powerful and optimal — it's still the champion of embedded languages for a reason. Augmenting C++ with STL increases its expressiveness but doesn't really help much with connectivity. We could pull in various SOCKS, Bluetooth, or Wifi libraries but that defeats the point of having a one-size-fits-all solution. C++/STL also isn't great for a rapid development or UX creation.

Score: 7/10

C++/Boost

Sticking with C++ but replacing STL with Boost gives us better connectivity options but Boost is still just a library and still lacking when it comes to developer environment. Plus there's no standard UX tooling.

Score: 8/10

C++/Qt

Finally, what about C++ and Qt? Well, Qt can leverage the best of the C++ "bare metal" capability while fully fleshing out rapid application development and UX components. The UX pieces are designed for cross platform and, because there's a complete IDE and community, it also does well in those categories.

Score: 10/10

IoT Challenges

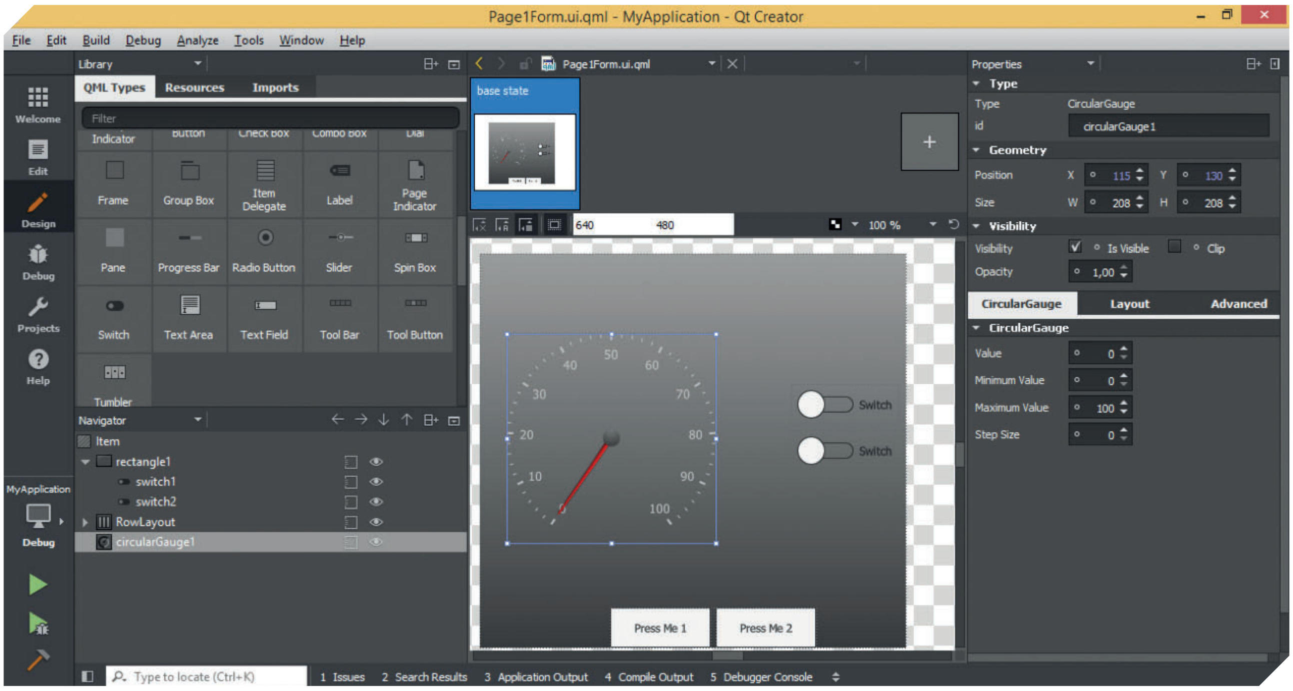
Qt came out on top in our comparison and for good reasons. However, it's not just how small the code or how many developers in the ecosystem that decides what IoT toolkit is best, it's how quickly and effectively you can use it. Let's dive into some IoT challenges in more depth and see how Qt tackles them.

Embedded Development

Much embedded development is still stuck in the stone ages. Going from a modern repository-plugged, auto-completing, syntax-colouring, integrated-debugging, plug-in-hosting, cloud-connected IDE to a green blinking cursor on a black screen feels like trading in the auto-start on your Tesla for the crank on your great-grandfather's Model-T. Sure, terminal sessions, console scripts, and vi/emacs have their place but we've had GUIs for what, 33 years now?

Qt has a huge advantage here. Qt Creator IDE works on all platforms, supports all kinds of extensions, plug-ins, short cuts, and time savers, and it cross-compiles to the target of your choice. It also includes tools for debugging, profiling, analyzing, and designing your code. Because all supported environments are compatible, you can do rapid prototyping on your laptop and push builds to your embedded target after you're happy with the results. If you think Qt is only about building a UX, it's not — there are plenty of Qt libraries for internationalization, strings, threads, XML, and JSON parsing, databases, sockets, Bluetooth, sensors, NFC, event management, and more for headless IoT development.

Most developers today don't have the luxury of working on a single, isolated product that doesn't share code with other devices. This is why having an embedded tool that has a powerful UX capability is really important for several reasons: to support multiple devices with or without screens, to leverage your core communication libraries between a desktop interface and a mobile gadget, and to share code with other IoT developers building different parts of the ecosystem. Qt lets you build a code base for non-UX devices that's lean and trim on which you can then layer a highly sophisticated interface for desktop, mobile, and tablets.


Page1Form.ui.qml - MyApplication - Qt Creator
File Edit Build Debug Analyze Tools Window Help
Library
QML Types Resources Imports
Frame Group Box Item Delegate Label Page Indicator
Pane Progress Bar Radio Button Slider Spin Box
Switch Text Area Text Field Tool Bar Tool Button
Tumbler
Navigator
Item
rectangle1
switch1
switch2
RowLayout
circularGauge1
Welcome Edit Design Debug Projects Help
base state
Switch
Switch
Press Me 1
Press Me 2
Properties
Type CircularGauge
id circularGauge1
Position X 115 Y 130
Size W 208 H 208
Visibility Is Visible Clip
Opacity 1,00
CircularGauge Layout Advanced
Value 0
Minimum Value 0
Maximum Value 100
Step Size 0
Type to locate (Ctrl+K)
1 Issues 2 Search Results 3 Application Output 4 Compile Output 5 Debugger Console

Qt Quick Designer in Qt Creator IDE enables you do create visual prototypes of user interfaces with easy-to-use drag and drop features.

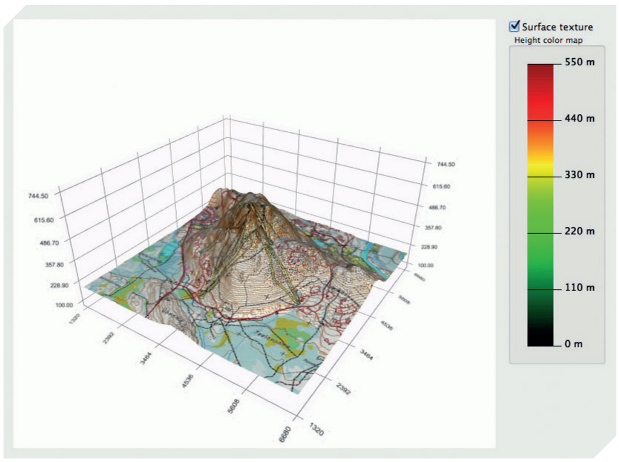
Big Data

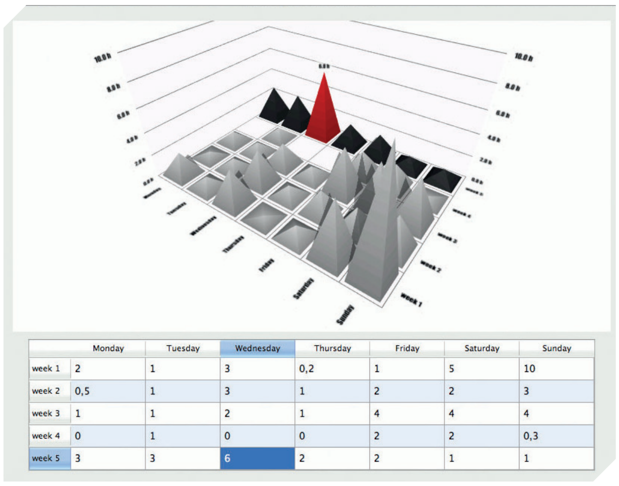
Not only is big data a buzzword on par with IoT but it's also crucial to the IoT vision: the continual data collected by millions of devices helps give our brave new world some brains. You'd be forgiven for thinking that big data is strictly a concern for the cloud. In actuality, endpoints need to manage big data too but on a different scale. Your IoT embedded system may not deal with exabytes but it still may need to collect, store, sort, filter, and process gigabytes of data.

To handle that workload, you need memory-conserving data manipulation, efficient format conversion, and persistent data storage. In C++, the programmer has direct control over data storage formats so there's no need for extraneous metadata or garbage collection. So being memory conscious is something that C++ is very good at and any of the C++ based options will do, including Qt. But converting between formats is needed to send and receive MQTT, JSON, or raw binary data to and from sensors in the most battery-sipping way possible — something not all options can do. Qt, however, has the libraries and the raw speed to handle this simply.

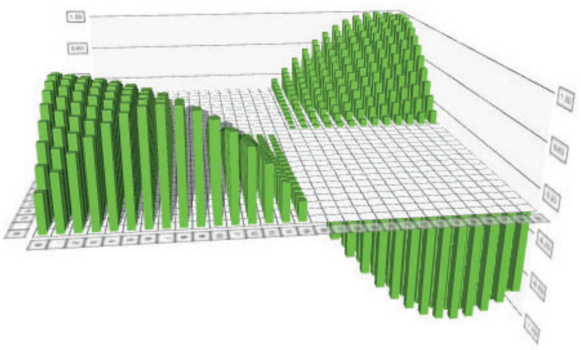
APIs for writing to databases are also necessary for the tidbits that have to be stored or referenced. Qt provides interfaces to SQL — like MySQL, Postgres, or SQLite — to let you easily save and process the data that's being collected.

There is one more aspect of big data to consider: data visualization. Although rarely needed by the end user, data visualization is critical for data scientists to see and interpret the data sets being generated. Qt provides a module specifically for visualization, providing 2D and 3D charts with a huge number of ways to slice, dice, and view your data.





	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
week 1	2	1	3	0,2	1	5	10
week 2	0,5	1	3	1	2	2	3
week 3	1	1	2	1	4	4	4
week 4	0	1	0	0	2	2	0,3
week 5	3	3	6	2	2	1	1



Consumer Experience

Your IoT devices are not standalone. While some IoT devices may have large, fully dot-addressable graphics with touch screens (home control panels, for instance) those are the minority. Users need to have a way to communicate with their devices, to configure device behavior, download data, or view and change status. The fully realized UX requirements of most IoT devices transcend a simple blinking LED and one-button interface. Since human and device interaction is a crucial part of the consumer experience, it's also very important to the success of IoT. And most IoT devices can only interface with us indirectly through a desktop, mobile, or tablet UI.

Qt was designed for creating user interfaces from its very beginning and this innate capability is needed in IoT. With Qt, you can build UIs that are simple or complex, classic or modern, standardized or custom. But most importantly, everything you build with Qt is cross platform. Whether your user is a Microsoft adherent, an Apple fanatic, or a Linux diehard, your IoT interface will run on all desktops. The same goes for the iPhone/Android schism. Supporting only one of those platforms is a recipe for dissatisfied customers; nobody likes to be left in the cold — not to mention those 7 to 8 percent of people with smartphones running something else like Windows, Tizen or BlackBerry. Qt gives you a way to simply support all your customers, regardless of their operating system or mobile phone partisanship, through a single codebase.

Speaking of a single codebase, the other big benefit of Qt is software sharing between your embedded device's firmware and the apps used to control it. In the bad old days, the languages and databases for RAD development forced you to build your desktop in something like VB or Delphi, while the embedded side was typically only in C. That meant you had two instantiations of the same code—in two languages, using two APIs, needing two test suites— which provided two distinctly different ways to hide bugs. With Qt, you'll be designing, building, and testing one data-parsing module, configuration management library, or protocol stack that's shared among all the places your software runs.

Extensibility

No developer team can afford to rewrite software for every new product. Not only is it costly but it greatly reduces the speed of new product creation. So why do we always seem to need to scrap software every couple of years and start over? Redesigns are often necessary due to unforeseen requirement changes and, more devastatingly, because of insufficient foresight while architecting a product. But worst of all, rewrites are necessary due to hitting a dead-end in a hardware or software dependency that your system critically relies on and that doesn't align with where you need to go — a product end-of-life or roadmap stagnation, for example.

There is no silver bullet for future proofing regardless of the language or framework you pick. You'll have to use your experience to architect for the future by building software that can be readily extended, modified, and adapted. There are a couple of ways that Qt can help to ensure your product is adaptable to changing circumstances.

New Embedded Hardware

You can pretty much guarantee that Qt will be among the first frameworks ported to any new board given its large open development community and embedded popularity. As your hardware needs change and grow, you won't need to worry about doing those ports yourself.

New App Environments

As different smartphones, UX paradigms, or Linux distros come into fashion, you'll have a strong Qt community keeping your framework relevant, creating new platforms for your device to interface with its users. Qt is an early porting option because it has such strong cross-platform support.

New Sensors

The Qt Sensor API has dozens of existing sensor types — if your new sensor is in one of these predefined classes, you can take advantage of this abstraction layer by creating a simple plug-in.

New Protocols

Qt has a number of networking and connectivity classes that make it easy to use existing socket communication over new wireless technologies. New protocols will most certainly require new classes to implement them but at least they can take advantage of many existing classes to make that job a bit easier.

New Backend Connections

You're likely going to need access to cloud services in your IoT UX. Qt has RESTful and SOAP interfaces, XML and JSON parsing, and cloud APIs for accessing AWS and Azure, with more development happening all the time.

Support for Plug-ins

If you need an IoT architecture that can dynamically add functionality, you can consider adding plug-ins. With plug-ins, you (or a developer community) can provide your product with new, refreshed features without having to rewrite the base code. Qt has a standard array of plug-ins for the features it provides but, more importantly for IoT developers, it also provides the base plug-in architecture that allows you to easily invent and create plug-in abstractions that fit your own particular needs. If you don't need a modular interface as sophisticated as a plug-in, there are a huge number of libraries from third parties that can be used with Qt: not only any C/C++ library, but many Qt-specific ones as well.

But having all the tools in the world for extensibility won't help you unless you use them.

Incorporating existing modularity into your system (be it Qt or any other framework) starts from the beginning. But the process you use can assist as well. For example, Agile development may be well suited for building extensible systems because it follows a model that essentially extends the system after every scrum. Regardless of the development process you follow, writing things modularly with abstractions that can be easily unit tested, writing for the least coupling between layers, and isolating your dependencies into as few touch points as possible will go a long way to keeping your IoT software alive and flexible.

Summary

More than ever, the world of IoT depends on software to make smarter, more connected devices. The criteria for your software framework may or may not match those we've defined in this paper. However, if your project needs to be powerful, optimal, connected, rapidly developed, and cross-platform with user interfaces and sharing capabilities, then you may want to consider Qt. Moreover, keep in mind that it's not just these basic attributes that will drive your software decisions. Most IoT software can benefit from other Qt attributes like the ease of embedded development, smooth handling of big data, a positive consumer experience, and assurance that your software architecture won't stagnate. But, regardless of what your IoT software needs to do, remember that you're building a framework that will live for a long time — choose carefully.

About Qt

Used by over 1,000,000 developers worldwide, Qt is a full framework that enables the development of powerful, interactive and platform-independent applications. Qt applications run native on desktop, embedded and mobile host systems, enabling them to deliver performance that is far superior to other cross-platform application development frameworks. Qt's support for multiple platforms and operating systems allows developers to save significant time related to porting to other devices.
Qt is created by developers for developers where making developers' lives easier is top priority. It provides an incomparable developer experience with the tools necessary to create amazing user experiences. Qt is platform agnostic and believes in making sure that all developers are able to target multiple platforms with one framework by simply reusing code. Qt gives freedom to the developer. Code less. Create more. Deploy everywhere.

About The Qt Company

The Qt Company is responsible for all Qt activities including product development, commercial and open source licensing together with the Qt Project under the open governance model. Together with our licensing, support and services capabilities, we operate with the mission to work closely with developers to ensure that their Qt projects are deployed on time, within budget and with a competitive advantage.
The Qt Company's goal is to provide desktop, embedded and mobile developers and companies with the most powerful cross- platform UI and application framework. Together with its licensing, support and services capabilities, The Qt Company operates with the mission to work closely with developers to ensure that the projects are deployed on time, within budget and with a competitive advantage.