The Qt Company

Whitepaper

Beyond the Code:

How Designers and Programmers Can Work Together to Deliver Successful GUIs for Embedded Systems

Beyond the Code: How Designers and Programmers Can Work Together to Deliver Successful GUIs for Embedded Systems

Tuukka Turunen, Director of R&D, The Qt Company, looks at the requirements for user interface design in embedded systems, provides suggestions for anyone tasked with developing advanced GUIs, and explains how building bridges between designers and programmers is one of the keys to successful implementation.

From industrial automation to medical devices, and from home appliances to automotive infotainment, well-designed and functioning embedded systems play a critical role to the performance and functionality of a growing number of modern products. Meanwhile, in a world where expectations are drastically getting higher and almost every company is looking for a flashy and 'cool-looking' UI, embedded systems designers are under pressure to meet ever-growing usability requirements consistently while delivering stunning user experience. Although coding is clearly a critical element to building the interface for an embedded design, good coding alone is no longer satisfactory – one needs to understand the user and choose the right development tools first.

Evolution of User Interfaces for Embedded Systems

In the past, many embedded systems were just that: requiring little or no direct link to the end user, therefore, only basic UI requirements were attained (if any at all). Also, many users of such systems were trained professionals of those systems, and it was acceptable for a system to be non-intuitive or even impossible to use by others. However, as embedded devices become more capable and are found at the heart of almost every electronic system, their features need to be more accessible. Paradoxically, as increased features add complexity more work must be done to abstract this complexity away from the user. Finally, add-in requirements such as touch-screen navigation, dynamic content and web access and it is clear that proper user interface design) is now a critical element in a creation of an embedded system.
As we have all grown accustomed to the ease use of mobile phones and tablets, even the most basic non-consumer design is expected to offer a GUI that is intuitive, unambiguous, informative, responsive and reliable. This puts pressure on the project development team, not only because GUI development can become more complex at the coding level – suitable 'building blocks' are not readily available for many embedded operating systems and, while attractive graphics can be displayed by any system with a good display and enough processing power, creating interactive and dynamic GUIs can be very demanding without the right tools. However, even with the right tools developing the best GUIs may require additional design skills that aren't always readily available in a software engineering team.

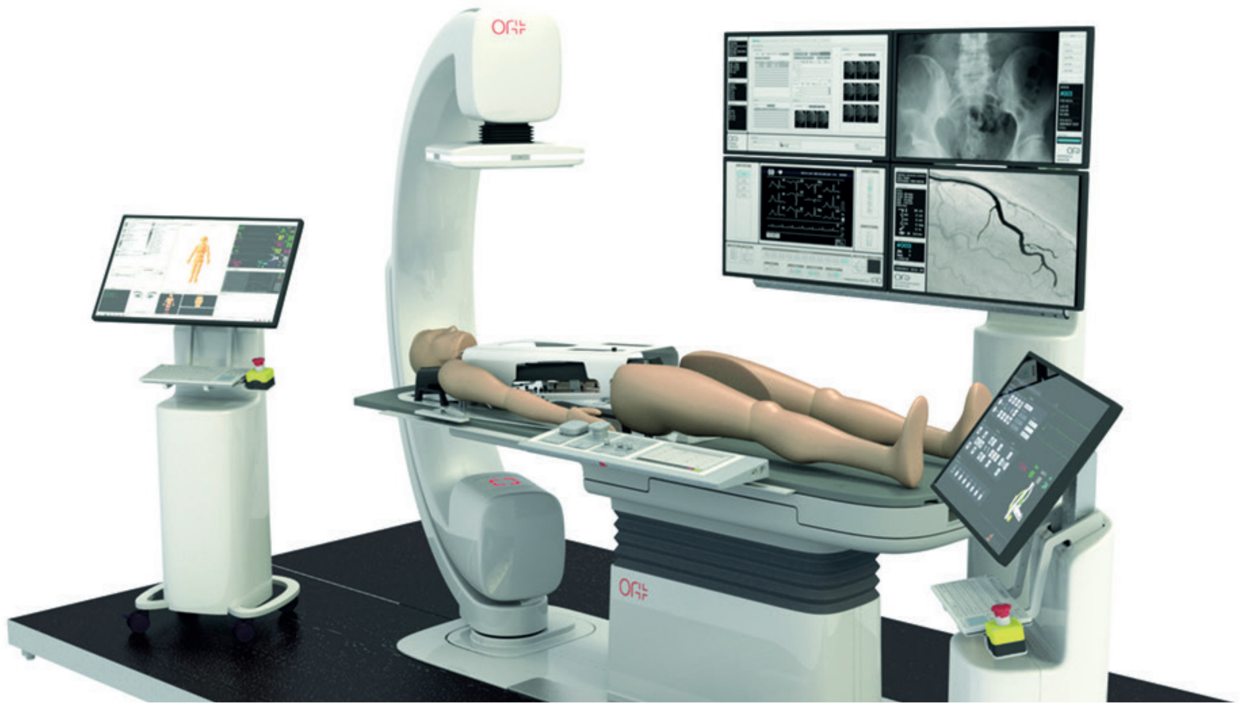
While functionality is, of course, essential, what distinguishes the best GUI designs from simply adequate implementations is good usability. Software engineers will always be able to implement a user interface that meets the functional specification, but the best user experience will only come from a higher level consideration and understanding of how that interface should help the user do their job better, or achieve a desired outcome in the simplest or quickest or most intuitive manner. It's not only about the looks of the application but the intuitive interaction with the end user. This is why there is a distinction between coding an interface and designing an interface – and why, in a world where users will no longer put up with devices and systems that are hard to learn or difficult to use there is much work to do before coding can be started.

Pre-Coding Considerations

The first stage of any good user interface design should be to define the user requirements and analyse how the user will perform the necessary tasks. With this in mind there is no substitute for understanding what the users need to accomplish and how they will approach key tasks in order of priority. Naturally this is true of all designs but becomes especially important in mission- and safety-critical applications such as medical or aerospace systems where how good the interface is can, literally, make the difference between life and death. With this in mind it can be valuable observing or recording the users as they go about similar tasks and it is always worth considering what problems and frustrations they might have with existing systems.

Once they understand what functionality the user needs, the designer can move onto the look-and-feel of the interface and the underlying workflow. Storyboards and prototype visuals should be created and discussed with the users and new iterations created based on their feedback. Modern tools can assist here, not only enabling fast prototype development but also allowing this work to be utilised in the final product rather the developers having to begin again from scratch, trying to imitate the designed prototype. Throughout, the designer should be looking to create an interface that is intuitive, consistent in form and function and 'clutter-free'.

Beyond the user requirements and task analysis, other factors that will impact the design can range from the operating environment – high levels of ambient light or, as is the case for many medical or industrial automation applications, the need to use gloved hands on a touchscreen may, for example, have an impact on icon size, colour and position – to the need for portability across hardware and software platform. Global deployment may necessitate several language versions or the need to integrate web functionality will also need to be considered at this stage. Pure hardware requirements may also set limitations to the UI design.

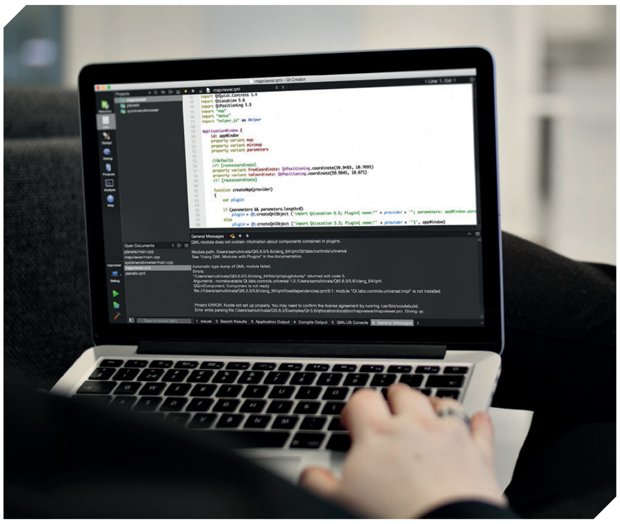


Design Environment and Tools

Once the designer has established the ‘big picture’ requirements for the embedded user interface, it is the task of the software engineer to make that interface a reality. The choice of design environment and associated tools can make a dramatic difference here – not only to the finished product but also in terms of the cost and speed of the project development.

For GUI design, the environment is necessarily divergent; graphical on one hand, source code on the other. Bringing the two together can successfully bridge the design and development aspects of GUI creation that begin with the pre-coding issues outlined above. A GUI design environment that supports a graphical approach to design (aesthetics) and development (coding) is particularly advantageous in this respect. In such an environment user controls represented graphically will ideally also contain, inherently, all of the code needed to invoke them on the target platform. In engineering terms, this implies an object-oriented approach to design; by selecting an element, all of its features are inherently invoked and, through this ‘inherency’, anything built using these elements will implicitly have access to the individual elements’ features and functionality. Superficially such an approach is not restricted to visual elements; any ‘component’ developed using an object-oriented methodology implicitly inherits the features of its component parts and passes on its features to any other components using it in a hierarchical fashion.

This underpins the structure of object-oriented programming languages such as C++, and is conceptually extended to graphical design environments that support object-oriented programming. Significantly, object-oriented software is inherently portable because ‘components’ embody (through inheritance) all of the features needed to define their functionality, making object-oriented code less hardware dependent. In most cases, porting object-oriented software is a simple case of re-compiling for a different architecture – at least as long as no hardware or operating system specific



functionality is used. And in the case of modern frameworks, such as Qt, that offer true cross-platform support this process is simplified even further. Such portability can minimise the development overhead of new designs and make it easier to migrate existing designs while providing a route for embedded developers to target different silicon price points. For example, Qt can be used to create an interface for a portable medical device built around a low-power, cost-effective processor and the same code can be leveraged on a significantly larger machine such as a CAT or MRI scanner that uses a more powerful processor.

One of the leading object-oriented programming languages in use today is C++. Based on the ubiquitous C language, C++ is a natural progression for many engineers looking to adopt an object-oriented methodology. By using an object-oriented approach it is possible to create a development ‘framework’ where modular libraries contain ‘components’ (or rather ‘classes’) for GUI widgets such as buttons, sliders, windows or dials, as well as other functional elements needed in embedded devices, including networking, support for multimedia (codecs) and internationalisation (languages). This advantage over interpreted languages, such as Java, can be the difference between a great and a mediocre user experience.

Framework Functionality

Of particular benefit to both designer and engineer is if the environment chosen offers a wide variety of pre-fabricated and customisable user interface components along with classes and modules that abstract common functionality in operating systems, ideally, beyond just GUI creation. For example, by also providing classes for XML, networking, inter-process communication (IPC), 2D and 3D graphics with hardware acceleration, threading, data bases, internationalization and multimedia the engineer effectively has access to a whole application framework. This advances implementation of key functionality while allowing developers to focus on core competencies that add competitive advantage to the final product.

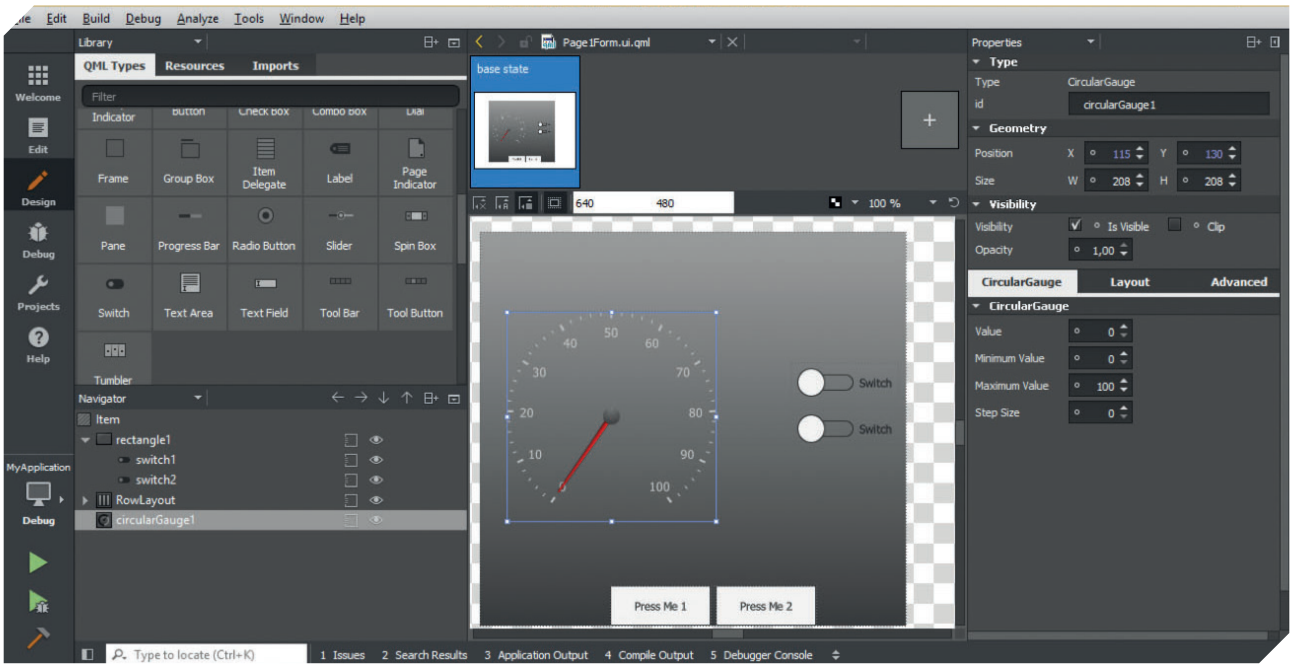
Take, for example, the Qt application and UI framework, which is well-established in the desktop environments and has also been well adapted to embedded and mobile systems. Consisting of both development tools and class libraries, the Qt framework allows designers and programmers to work together to produce a GUI using pre-built components that are easily customised to match the desired look-and-feel and behaviour of the application.

Because of object-oriented design, the components can also be extended effortlessly to create new UI components.

The Qt design environment enables the rapid creation of graphical elements, for example, drag-and-drop UI creation in addition to writing the code. High performance text rendering as well as pre-rendered fonts help to reduce processing overhead while, beyond the graphics, special web tools meet connectivity requirements by allowing designers to build advanced user interfaces that incorporate real-time web content and services. For global deployments a dedicated linguist tool enables engineers to translate and adjust applications to different world languages, including support for Asian characters and right-to-left script.

Portability

One of the significant advantages of Qt is the way in which the framework has been built to provide a top-level, cross-platform application programming interface (API), which enables deployment of GUI designs and applications across multiple software environments with minimal additional work. For



instance, source code from one device target, such as a desktop PC environment, can be deployed to an embedded operating system or a mobile device without any requirement for re-writing. This not only allows multiple deployments from a single development project but also future-proofs the code against next-generation challenges in the platform itself—a significant benefit in a world where operating systems themselves are subject to continual revisions and where product lifetimes can sometimes be measured in months rather than years.

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.

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.