Exploring Cross Platform Development using the Qt Framework

Tiernan Kennedy
B.A.I. Engineering
Final Year Project April 2011
Supervisor: Dr. Jeremy Jones
Part I

Declaration

I declare that the present work has not previously been submitted as an exercise for a degree at any university. It consists entirely of my own work, except where references indicate otherwise. The library of Trinity College, Dublin may lend or copy this thesis on request.

Signed: Tiernan Kennedy
Part II

Abstract

This project was an experiment in the creation of a single build environment, to harness the power of Qt, to bring multiple platform support from a single source. The premise and outcome was experimental, yet there are real world commercial applications for a system like this. Weighing up the benefits against the relative infancy or certain aspects of the project to find the most favourable outcome is the aim.

“code less, create more, deploy everywhere”
Contents

I Declaration 1

II Abstract 2

1 Introduction 6

2 Project Aims 7
  2.1 What’s the problem? 8
  2.2 What is the ideal solution? 11

3 Cross Platform Development - State of the Art Review 12
  3.1 Why is there a need for cross-platform software development? 12
  3.2 Operating Systems 13
  3.3 Commercial Viability 19
  3.4 Market Share 21

4 What is Qt? 26
  4.1 History of Qt 27
  4.2 Application Development in Qt 30
  4.3 Alternatives to Qt 31

5 The Demonstrator Application 35
6 Deployment

6.1 Deploying Qt Applications ............... 46
  6.1.1 Static linking ......................... 46
  6.1.2 Dynamic Linking ...................... 48
6.2 Deployment of Demonstrator .......... 49
  6.2.1 Linux ................................. 49
  6.2.2 Mac OS X ............................. 53
  6.2.3 Symbian ............................... 56

7 Android 60

7.1 The Qt-Android Port ................. 60
7.2 Qt Lighthouse .......................... 61
7.3 Android NDK ........................... 61
7.4 Creating the Android Build Environment .. 62
7.5 Java Connections ....................... 65
7.6 Working Features ....................... 66
7.7 Unsupported features ................... 67
7.8 Duplicating the Android Look .......... 68
7.9 Result .................................. 70

8 Overall Results 72

9 Conclusions 73

  9.1 Is Qt a viable alternative to native platform development? .................. 73

10 References 76
1 Introduction

Cross platform development is the process of creating software that can be run on multiple platforms. Since the advent of UNIX, cross platform development is technically possible, but competition between manufacturers has led to a number of competing Operating Systems, in the mobile and desktop space. For commercial reasons it is more advantageous for applications written for a single platform to be incompatible with competing platforms. However from a developers point of view, rewriting an application multiple times to ensure everyone can access an application is time consuming and also technically difficult. It requires proficiency in the language used for application development for every platform the application needs to be deployed to. Cross platform development can be divided into two types, the first is software that requires no modification to run on each platform, the second is where it is modified specifically for each platform it runs on. The first type, usually requires a translator, and is therefore not technically satisfactory. Over the course of this report, both methods will be examined and discussed. The Qt cross-platform application framework will be utilised as a possible answer to the issue, as software compiled with Qt requires no translator when deploying with Qt, and thus is a more complete solution.
2 Project Aims

The aim of this project was to investigate seek a solution for developing native GUI applications across multiple platforms from a single codebase. The Qt Framework was chosen as the application framework for this task, with a number of platforms already supported. A major objective was to target to the fast growing Android platform and deploy a cross platform application using Qt libraries to a Google-supplied Android development phone, while running that same application to other platforms. Examining the Operating System market, paying particular attention to the rapidly expanding smartphone operating system market, as a target for these applications, is also covered. We will discuss alternatives to the Qt framework and weigh up the benefits of cross platform development as a whole.

This project required the creation of a demonstrator application, which utilises the many features of a modern application, e.g. graphics, touch/key input, network access etc. It also required the creation of build environments for the different platforms the demonstrator application can be deployed to. Getting the application running on as many operating systems possible to display the potential such a system would be, while ensuring that it ran correctly, given the different parameters of each operating system. Differing device inputs such as using a touchscreen instead of a mouse, or using virtual keyboard instead of a physical keyboard demanded that the application would have to be tested thoroughly to ensure a seamless user experience across all platforms, desktop or mobile.
2.1 What’s the problem?

The problem we face today in application development is the same one that has been in existence since the very beginning of software development. Generally code written for a specific system or platform will be incompatible with all other systems. This goes back to early computing, when programs were written in machine code, using low-level languages like Assembly, where the programmer must have detailed knowledge of every aspect of the computer’s architecture, including processor register function, I/O details and memory layout, and will operate at bit and register level and interact directly with the machine. As machine code addresses a specific computer system, no code written for one computer could run on any other non-identical computer. Every manufacturer of hardware designed their own custom operating systems. At this point they competed in terms of software and hardware. With the creation of UNIX and then the C programming language we had the first high level language and multi platform operating system.

UNIX was originally written in Assembly language. When the kernel of UNIX was rewritten in the C programming language, a programming language that compiled high level commands into machine code, it became the first portable operating system, capable of being easily ported to any hardware. This created the first cross platform system, an operating system that could run on any computer, and UNIX applications could be run on any of these systems. Different manufacturers could create computers that could run UNIX, and share the same software, recompiled on each processor. When UNIX became very popular, vendors such as IBM, HP and Sun began to modify UNIX to differentiate themselves, which
splintered UNIX to the point where all versions of UNIX were not able to run all UNIX software. It was this fragmentation by different manufacturers that put an end to the possibility of a common Operating System, that all software could be written for. Microsoft Windows had a market share of well over 90% in the 1990’s, and Apple Computer almost went bankrupt, which could have left us with one major operating system and the open source OS Linux, which many consider not to be consumer friendly due to reliance on the command line for many of its functions. If this had happened, software for PC would be essentially all written for Windows, ending any cross platform necessity in desktop platforms. However the opposite has occurred, and OS X and Linux are both on the rise against Windows, so programmers cannot afford to ignore either platform.

![Diagram of Operating System Layers](image)

**Figure 1: Role of an operating system**

With the explosion in smartphone usage, we have seen a large number of operating systems created in a very short space of time, each vying for dominance in this new market. We currently have six large players, Symbian, iOS, Android, Blackberry, WebOS and Windows Phone 7. With an increasingly fragmented market the need
for solutions to multiple platform development is a hot topic. Take software like Dropbox, popular file sharing software that is used to sync files across multiple platforms. This software is available for multiple platforms in order to increase the amount of customers it can reach and also to cater for an individual who needs to sync files between their Mac and their Android phone, or from their Windows PC to their Blackberry. The Dropbox application is available on Windows, OS X, Linux, Android, iPhone, iPad and Blackberry. Despite this broad range of supported OSes, there are still community made Dropbox apps for Symbian, Maemo and Windows Phone 7, which are officially unsupported by Dropbox. This shows the level of demand for popular applications to be present on every platform. To maintain that number of independent applications while still aiming to continue improving your service can be a daunting task for any company. And thus, there is a need to develop applications for a wide range of target devices using one simple development framework. Targets should include desktop computers, tablets and mobile phones. Operating systems such as Windows, Linux, OS X, Android, iOS, Blackberry and Symbian are all major operating systems and to pick and choose which operating systems to support is a major issue in software development today. To create software form a single codebase would accelerate innovation by focusing on software developers time on improving existing applications rather than expanding the amount of operating systems it is rewritten for.

Figure 2: The multitude of competing operating systems
2.2 What is the ideal solution?

The ideal solution to this issue is the creation of operating system independent applications that can be deployed to all major platforms using the same source code. Java was initially mooted as the solution to this issue, but it is not a true cross platform solution, as it requires a virtual machine, and is now incompatible with most of the major mobile platforms, most notably Android. While this sounds like an overly simplified solution to the fragmentation issue, there are frameworks like Qt, which have already achieved results across a number of platforms. While we do have to make admissions for specific differences in operating systems, Qt has methods of handling platform specific features, using #ifdef statements, which will be discussed in detail later. While there is no existing complete solution for cross platform development in existence we will be examining Qt as the possible nearest candidate to this ideal model. Given the LGPL license and a strong developer community behind it, Qt appears to have momentum in this goal. A unique aspect of Qt is the fact that after its inclusion of Symbian followed by Maemo/MeeGo it became one of the few cross platform frameworks that supported both mobile and desktop Operating Systems.
3 Cross Platform Development - State of the Art Review

3.1 Why is there a need for cross-platform software development?

There has always been a market for cross platform toolkits; when UNIX was a very popular platform, there were solutions such as XVT, Galaxy and Zinc which aimed to fulfill this purpose. However, on the release of Java in 1995, these toolkits became obsolete as developers flooded to Java as the cross-platform solution. It became apparent eventually that Java was not actually very suited to creating client facing applications, as it required a virtual machine, alternative GUI toolkits began to appear. Java was mooted as the answer to this with its “write once, run everywhere” mantra. However this has not been the case, especially in the mobile space. Cross platform mobile development implies using one code base to build an application that runs on multiple platforms. Qt is one of those toolkits, and has support for many features Java does not handle, such as XML, SQL, threading, network programming and advanced data structures. As an application developer, you are looking to optimise applications by utilising as many features as possible within your projects budget and development time. Given the multitude of Operating Systems that are competing today, with many more since the onset of Smartphone operating systems, there is a huge advantage to having application on as many of these platforms as possible. However, given that all the operating systems are structurally different to each other this is not an easy task. However there is no magic bullet solution to this issue just yet, there are many purported answers but none deliver a consistent user experience across all the platforms. Features like native hardware acceleration, native look
and feel via the application API and sensor support are often unreliable. Some solutions are written in web languages such as PhoneGap, which lets developers code their applications in JavaScript, CSS and HTML. These applications will run on multiple platforms, but have more in common with mobile optimised websites than true native applications for the platform.

3.2 Operating Systems

Almost all computers have operating systems (OS), the most common being the Windows family of operating systems, developed by Microsoft. The other leading operating systems are Macintosh, developed by Apple and the UNIX family of operating systems, which have been developed over years by corporations, collaborators and individuals. The operating system performs two main functions:

- It manages the hardware and software resources of the system. These resources include the processor, memory and disk space in a desktop computer. For mobile systems the OS also manages the screen, keypad, battery and network connection.

- It provides a stable system for software to interact with the hardware of the device without having to know all the details of the hardware. The second task of the OS, providing a consistent application interface, is what is of interest to us, as it affects how software runs on different hardware configurations. This is essential given the multitude of hardware currently in existence, which is constantly changing. An Application Program Interface (API) allows a software developer to write applications for an OS and be assured that the software will run correctly on any machine that has that OS installed, regardless
of the configuration of the underlying machine. For this to work correctly, the OS developers must keep their systems flexible, so they are compatible with the thousands of hardware devices available, including peripherals. APIs let application programmers use functions of the computer and OS without having to know all the details of the CPUs operations. The operating system, connected to drivers for the various pieces of hardware, deals with the changing states of the hardware. The programmer simply writes the program to be compatible with the API and let the OS handle the rest.
Desktop Market Share

April 6, 2011

There are three dominant Desktop Operating Systems, Microsoft Windows, Mac OS X and Linux. Linux has quite a small market share but is popular among developers. It is currently said to have 1.65% market share, with Mac OS X at 5.19% with Windows taking up nearly all of the remainder. While this is very favourable for Windows, certain countries have very high Mac OS X penetration, such as Switzerland, where Mac has 17.61% of the desktop market. It is in Asian countries where Windows is very dominant, possible due to the high cost of Apple hardware. As desktop operating systems have been on the market for decades we now have a situation where there are three big players, and in the 1990s there was really just one, Windows had 91% market share in 2009. However Apple have seen a resurgence in sales, possibly due to the unprecedented success of the iPod, iPhone and iPad. Also the arrival of netbooks running Linux to keep costs down have further eroded Windows’ market share.

Mobile Market Growth

The combination of a number factors in recent years, such as a wealth of devices, price reductions and performance and battery improvements, mobile devices are now a viable platform for which to deliver high quality applications. The onset of the smartphone has led to a boom in an already massive industry. The 1990’s saw the explosion of the feature phone, and we can now look at 2010 onwards as the boom of the smartphone. Mobile phone users in
OECD countries grew from 10.5 million users in 1990 to 292.8 million users in June 1999, a 2688% increase.

Despite the impressive increase that was seen in the 1990’s, Smartphones look to grow even quicker, this statistic showing the smartphone growth predicts that smartphone sales will grow from just under 100 million in 2009 to 970 million in 2013. That’s an 870% increase in just four years.
Mobile Platform divergence

The popularity of mobile applications is linked to the popularity of Smartphones, and as Smartphone penetration increases, so does the number of applications available, and the amount of money spent on mobile applications. There are six main platforms available in the market today: Apple iOS, Google Android, Microsoft Windows Phone 7, RiM Blackberry, HP webOS and Nokia Symbian. However applications for these operating systems are incompatible with each other, applications for iOS are written in Objective C, Android uses its own version of Java, webOS uses web languages such as Javascript, CSS and HTML, Blackberry uses Java and Symbian uses C++ with the Qt libraries. With the market as open and fragmented as this it shows the huge potential of an effective cross platform solution to target as many of these Operating Systems as possible. At this stage mobile applications have become a huge part of peoples everyday lives. Mobile applications become more popular as Smartphones become more widespread. Advertising to people in a device that is permanently on their person is holds huge advertising allure for companies. The most popular mobile applications are relevant applications which complement the users daily life and increase the functions their phone is useful for.

Drawbacks of cross platform mobile development

Each platform and each phone has support for different features, i.e. Android is the only platform currently supporting NFC (near field communication) chips in its SDK, and only one phone is on the market which has one of those chips. There are also features such as accelerometers, GPS and dual cameras, which may not be on the lower end phones, rendering some applications useless. Creating an app for all platforms means you will not be able to play to the strengths of a certain phone/OS. Creating more basic apps to ensure compatibility is the outcome of this, and can end up with
far simpler apps than if natively coded. Creating an application for two platforms does not necessarily take twice the time and cost. Development may take double the time, but deployment and testing will still take an equivalent amount of time, so only money is saved in development. User inputs and outputs make a true cross platform mobile device. Although touch screens are very popular with Smartphones, a large number of phones use key input and do not have a touch screen. Trying to create a truly seamless experience in the application will be very difficult with such a different input and output. One must compensate for this, possibly to the overall detriment of the application. Keeping up with this fast moving market can be an issue if you are trying to keep on top of changes to each OS. Android updates alone occur every six months.

Figure 5: Mobile OS market share
3.3 Commercial Viability

Desktop Applications

The size of the market for desktop applications is estimated at US$303.8 billion in 2008, and an increase of 50.5% is expected by 2013, valuing the global software industry at US$457 billion. Due to Windows’ dominance in this area, the majority of that money is spent on Windows application, and is the primary target of any major software company. Interesting figures were released however, showing a slight decline in ownership of desktop PC’s as mobile devices such as laptops, tablets and cellphones all gained market share.

Mobile Applications

Mobile applications have seen unprecedented growth in the past 24 months. Sales reached €3.6 billion in 2010, with 8.2 billion applications downloaded and sales are forecast to surpass €10 billion in 2011, with 17.7 billion application downloads. The growth between 2010 and 2014 is predicted to be over 1,000%. Currently Symbian is the market leader for Smartphones, but its ten year lead is about to
be broken by Android, which has amassed a huge market penetration in just over two years. In the US market, Android has already become the market leader, cornering 29% of the market, globally Symbian has 40% penetration, with Android lagging behind with less than 20%. Figures from Gartner suggest that the two platforms will be tied at 30% each in 2014. This means that if you were developing for Android and Symbian, you would be targeting 60% of the global market, and would be into the foreseeable future. Using the latest figures available from NetMarketShare, of all devices accessing the Internet in March 2011, Windows is still by far the most popular, with 89.58%, so despite the growth in mobile devices with internet access the desktop remains the main way of accessing the internet. Mac OS X follows with 5.25%, and iOS is the highest ranking mobile OS, with 1.87% of the market.
3.4 Market Share

Microsoft Windows has long been the dominant desktop platform, currently enjoying 90% share of the desktop market. Mac OS X has been shipping larger numbers every year, but still remains an expensive niche. In enterprise solutions, Microsoft have maintained what is close to a monopoly, the Microsoft Office suite that is created by is the recognised standard in word processing and other office related desktop applications. Microsoft Windows 7 is based around the Windows NT kernel. Desktop applications for Windows have traditionally been distributed by their vendors through physically distributed discs, or over the internet, which is increasingly the sole distribution of software. The advantage of this method of distributing software this way is that the developers make receive 100% of the money paid for the software. The drawback is that payments have to be handled, and there is no central place where consumers can find products. Software developers and companies must market their applications independently.

Mac OS X is a UNIX based operating system that was released in 1999. It has 5.25% market share worldwide, but this can be as high as 15% in certain countries. Apple recently released an App Store for desktop applications for OS X, in an effort to replicate the success they have seen with their mobile App Store. On the first day of sales Apple had one million application downloads. It is too early to say if this will be a success, but given users familiarity with the App Store model for iOS, this could become an interesting new revenue stream for desktop applications. There have been similar efforts to this by Microsoft in the form of Windows Marketplace. However it was discontinued and now there is only the “Microsoft Store” for desktop applications, which only sells applications devel-
oped by Microsoft. As we have seen in recent history, especially with tablet computers, Apple has been able to use clever marketing and its fanatical user base to revive concepts that have failed, like the case of tablet computers with the iPad. So with this new direct marketing of desktop applications Apple may open up another revenue stream for developers. Some Linux distributions such as Ubuntu have a preinstalled software manager where free applications can be directly obtained.

Symbian OS is the most popular smartphone OS on the market with 37.6% of the market sales. 111.6 million handsets running Symbian were sold in 2010. However Symbian has been ceding big losses in market share in recent years to the other platforms, particularly Android of late. Androids huge percentage gains are almost equivalent to Symbians percentage losses in the same period. Symbians market share is less than 1% in the highly competitive US market, but it has market share of higher than 40% in many emerging Asian markets, so it cannot be discounted as a relevant platform. Nokias recent announcement has left the future of Symbian in huge doubt, as Nokia ships such a high volume of Symbian devices. They have claimed that there will be a further 150 million devices sold before they fully transition to Windows Phone 7. Nokia currently sells applications through their Ovi Store, which has been gaining traction, with 4 million downloads a day by January 2011.

Android is an open-source operating system developed by Google and the Open Handset Alliance. Android has been characterised by its fast growth and very quick development schedule. Applications for Android are primarily written using the Android SDK. These
applications consist primarily of Java code, with the addition of Android specific libraries used to control the device. However it is not compatible with Java SE or Java ME, as it only uses the Java syntax, and does not include the required libraries for these platforms. In the last quarter of 2010, Canalys reported that the Android operating system was the world’s best selling smartphone platform, edging out Symbian from the lead it had held for ten years. Google has shaken up the market in terms of software updates with Android. Android has had 11 versions to date, from 1.0 which was released in September 2008, to 3.0, which was released in February 2011. This rapid development has led to criticism about fragmentation on the platform, where some users are still using version 1.5 (Cupcake) while 3.0 is released. However from the graph below we can see that only 3% of users are using this version, and that the vast majority (92.8%) of users are using 2.1 or higher. This rapid development has aided Android in becoming the largest smartphone platform in the world in such a short period of time. The Android Market has also been growing at a fast rate, and now boasts over 130,000 applications. What is miraculous about that feat is that the Android platform less than three years old. The first mobile phone commercially available was the HTC Dream or G1, and that was released on October 22, 2008. The Android Market has been very successful too, boasting over 150,000 applications two years after its launch.

iOS was an early innovator, and could be credited for the current smartphone boom, especially the boom in application sales, and the monetization of Application sales. The iOS application store has sold 10 billion applications to date, and it has the highest percentage of paid apps to free apps of any application store. Apple have the
strictest policies of any app store and are notoriously closed in what is allowed to be sold on their app store. This had led to a developer backlash, and could be cited as one of the reasons for the growth in the Android Market. iOS is currently unsupported by Qt, there is a community port under development, but has not reached the level of the Android port. Legally, it is technically sound, as Apple eased restrictions on how iOS applications are developed after a controversial rule they put in place, banning apps not written in Objective C, the native iOS programming language.

Maemo is a Linux based platform, based on the Debian Linux distribution. It was developed by Nokia for Smartphones and tablets. The current version is Harmattan, version 5 of the software. It has been announced that Maemo will be merging with Moblin to form MeeGo, but this process is in the early stages. There are only four devices that currently run Maemo, all developed by Nokia. However MeeGo under development in a collaboration between Nokia and Intel, Intel were previously developing Moblin, a similar platform and
it was announced that the two projects would merge. MeeGo is designed for a large number of targets, such as netbooks, tablets, automobile entertainment systems and IPTV boxes. From Maemo 6, the operating system will be referred to as a MeeGo platform.
4 What is Qt?

Qt is a cross-platform application framework that facilitates the deployment of software to multiple platforms whilst maintaining a common source code. It is written in C++, but there are bindings for many other languages such as C, Python and C#. Qt uses standard C++, but uses a code generator called the Meta Object Compiler (moc) to enrich the language with the addition of Qt classes, of which there are over 500. This allows the creation of loosely coupled components that can be reused. A graphical user interface, GUI, consists of widgets (or controls as they are called in win32). These widgets either consist of other widgets and are called composite widgets, or are a unique widget themselves such as a button, line edit or any other element you find in an application. These classes provide easy implementation of a huge array of GUI features, as well as other supported features, such as SQL support, Threading, XML, CSS and Networking. Qt supports OpenGL graphics which means it supports Multimedia and 3D graphics. It is fully object orientated and can be considered to be event driven. Qt provides a conventional event model for handling mouse clicks, key presses, and other user input. Qt applications can support all the user interface functionality required by modern applications, such as menus, context menus, drag and drop, and dockable toolbars. Officially Supported Platforms are Mac OS X, Windows, Linux (Desktop and embedded),

26
Symbian and Maemo. Qt applications run natively, meaning that they are indistinguishable from native applications, and can compile from a single source code on Linux, Max OS X or Windows, with few or no changes. Qt includes a rich set of widgets that provide standard GUI functionality across these platforms. Another interesting feature of Qt is the level of internationalisation support it contains. When beginning to write code in Qt, programmers are encouraged to use the "tr" tag before any strings that can later be translated using a tool called Qt Linguist. I found this quite interesting as it makes Qt portable to other languages as well as other platforms with the minimal possible effort. The Qt C++ framework is widely used by major companies such as Boeing, Adobe, IBM, Google, NASA, Skype and Motorola.

4.1 History of Qt

[1] The Qt framework was initially developed by Haavard Nord and Eirik Chambe-Eng, two Norwegians who met at the Norwegian Institute of Technology in Trondheim, where they both received Masters Degrees in Computer Science. The beginnings of Qt were started when they were working together on a C++ database application for ultrasound images. It was a GUI application which needed to run on Windows, Macintosh and Unix. Inspired by the challenge, Haavard began writing the first classes that would become Qt in 1991. A year later, Eirik devised the implementation of “Signals and Slots”, a powerful feature within Qt for easily connecting objects. It is a simple system, but very powerful, and has since been implemented in other C++ frameworks. By 1993, the graphics kernel was complete and were now able to implement widgets and at the end of that year they decided to go into business to market their framework. The company was incorporated on March 4, 1994,
originally as Quasar Technologies, and eventually evolved into Trolltech. Their first customer was the Norwegian company Metis, who awarded them a contract to develop software in Qt. Trolltech's first employee outside the two founders was then hired, they hired Arnt Gulbrandsen who was responsible for the extensive documentation in Qt, as well as writing code for the framework.

[1] Qt's first public release was in May 1995, and was available for both Windows and Unix development, with the same API on both platforms. On Qt's release, they established a business model of commercial and open source software development, which remains to this day. Open source development was free with the platform, but if it was to be used commercially, a commercial license was required. Qt gained its second customer in March 1996 after a long wait where their only source of income was the Metis contract. The client was the European Space Agency, and they bought ten commercial licenses. Qt 1.0 was released in September of that year, and another developer was added to the Qt staff. At the end of 1996, Qt had sold 18 commercial licenses to 8 different customers. Matthias Ettrich joined Trolltech and oversaw the release of Qt 1.40, Qt 2.0 was released in June 1999, and in August Qt won the LinuxWorld award for best library/tool. Qt 3.0 was released in 2001, now featuring Windows, Mac OS X, Unix and Linux (desktop and embedded) support. Qt 4.0 was released in the summer of 2005, and was a major upgrade, and now exceeded 500 classes, 9000 functions and over half a million lines of code. The Qt libraries were now separated so developers would only have to include whatever libraries they needed. This was the first version of Qt to support commercial and open-source development on every platform it was available on.
Nokia acquired Trolltech in June 2008 with a view to using Qt as the main development environment for all its devices, and Qt was ported to the Symbian S60 platform. Version 1.0 of the Nokia Qt SDK was released on June 2010, with the inclusion of a Symbian and Maemo emulator. Qt was chosen to be the development environment for MeeGo, an operating system that Nokia collaborated with Intel and other partners in developing. However on the 11th of February 2011 Nokia announced that they would be adopting Windows Phone 7 as their primary Smartphone strategy, all but abandoning Symbian and MeeGo in an unprecedented move. Nokia will still produce Symbian phones and a single MeeGo phone in 2011, but after this point they will be solely developing for Windows Phone 7. Nokia CEO Stephen Elop also confirmed that there would not be a Qt port for Windows Phone 7, and that Silverlight and the XNA framework will be the only development options for Nokia Windows Phone 7 developers. This announcement was met with shock and anger from the Qt and Symbian development communities. Despite Nokia’s claim that 150 million Symbian phones remain to be produced, it essentially makes Symbian a dead platform, and Qt’s future lies in the balance. However the independent developer port of Qt to Google’s Android platform, unsupported by either Google or Nokia, offers an interesting future for Qt. Strategically it could be very harmful for Nokia, as all of its Qt developers jump ship to the rival platform Android instead of developing applications for the WP7 operating system that Nokia have gambled their future upon. A month later Nokia stated that they had the commercial licensing division of Qt to Digia PLC.
4.2 Application Development in Qt

Investigate Qt’s “code less, create more, deploy everywhere” strategy.[6]

The Qt cross-platform application and UI framework was created with the purpose of reducing the complexity of creating applications for multiple platforms. The Qt class libraries handle most of the platform specific changes, such as the user interface design. Creating an application that looks appealing and maintaining that look for all your deployments can be very time consuming. Using rich features like tabbed and stacked widgets, line edits and other GUI components can increase the complexity of the project while the underlying logic of the program remains the same. Qt uses C++ to write the application logic, but uses it libraries to create all the GUI components, not relying on the native APIs. This means that rich GUIs can be supported across platforms without difficulty. The Qt application framework has both development tools, like the Qt Creator, a fully featured IDE which has Qt Designer specifically for designing User Interfaces, as well as the class libraries, which provide easy implementation for a broad number of features. A good example is QtWebKit, introduced with Qt 4.4, which uses WebKit technology, and supports the rendering of HTML, CSS and Javascript within Qt applications.

Pros

• When compiled machine code is generated for supported operating systems, allowing performance similar to native

• Multiple targets can be created within the Qt Creator IDE, very quick deployment

• Supports desktop and mobile operating systems, has support for multiple different inputs
• Native look supported in Mac, Linux and Windows
• Custom Qt look for all platforms also supported

Cons

• Qt is not currently compatible with the newer mobile platforms.
• It will not support Windows Phone 7, and and iOS port is only in the very early stages.
• Relying on a community built port to Android is not ideal
• Mac OS X system needed to test and target a Mac application, Windows system needed to target Windows

4.3 Alternatives to Qt

Virtual Machines (desktop)

Virtual Machines hide the underlying platform from the application by running a virtual machine between the application and hardware. Java is the an example of this and is also the most popular programming language in the world. The advantage of a VM system is that the instruction set need only be deployed and the application will run accordingly. Java is an interpreted language, so when a program is compiled it creates bytecode. This bytecode is then translated by a translator from bytecode into machine code and then the code is inserted into the CPU to execute it. Every machine that wants to execute the Java bytecode must have the translator, which is a Java Virtual Machine(JVM). So Qt is faster than Java code, as compiling Qt code contains machine code run directly by the CPU, not through translation like Java. However the introduction of JIT (Just In Time) Compilers has made Java a lot faster and
is now comparable to native code running times. As the platform must have a JVM to enable the running of a Java Applications, we can not really claim Java is fully platform independent, and is incompatible with many platforms. A notable case of this is Android, where a new Dalvik Virtual Machine was created for the OS, making Sun Java incompatible with the platform. The same is true of iOS, which does not support running of Java code.[7][8][9][10][11][12]

Pros

- Java has automatic garbage collection, which is not a standard feature in C++, however every QObject handles deletion of it’s children. QWidgets and all their descendants are derived from the QObject class, so they are QObjects, so all QWidget garbage collection is handled. There is also a qDeleteAll function for anything not declared as a child of a QObject.
- Java performance is approaching native speeds with JIT compilers
- Java is open source (this may be subject to scrutiny given Oracles recent buyout of Sun and resulting IP lawsuits)
- Compatible with desktop systems

Cons

- Target platform must have compatible Java Virtual Machine, Android and iOS are incompatible
- Swing (the Java GUI creator) has ceased proper development in recent years, not at the standard of Qt’s functionality. Qt GUI creation is a complete feature set
PhoneGap

PhoneGap is an open-source mobile development framework. The main advantage of PhoneGap is that it allows mobile applications to be written in web languages such as HTML, CSS and JavaScript. The advantage is similar to Qt, where only one programming language is used, and PhoneGap supports six mobile operating systems, iOS, webOS (HP), Android, Windows Phone 7, Symbian and Blackberry. It must be noted that PhoneGap is not a true competitor to Qt in this respect as it does not support deployment to desktop operating systems. Also as the applications are written in web languages, they are not truly native, like Qt applications are. All the graphics rendering is done via a webview, similar to QWebView in Qt, but the applications do have access to native features that a web application does not, like the accelerometer, camera and compass. This is achieved through the packaging up of these native functions into a project that allows access to the functions via JavaScript calls.

Pros

• Written in common web languages - HTML, CSS and JavaScript.
• Widely supported - iOS, Android, WP7, webOS, Symbian, Blackberry

Cons

• Not natively run. Access to all APIs incomplete.
• Multiple IDEs needed to compile for the different platforms i.e. Xcode required for iOS deployment
• Desktop Operating Systems not supported
• Native look not supported, can be implemented with CSS.
5 The Demonstrator Application

The demonstrator was created similar to the current applications of today, leaning towards the style of the mobile applications, as it would have to function correctly on three desktop platforms, and three mobile platforms. For this reason, the tabbed application seemed like the best choice. The iPhone tabbed applications have their tabs situated at the bottom of the application whereas the Android tabbed apps usually feature their tabs at the top. As Android was being targeted, as well as the desktop operating systems, the tabs on top were used. Qt has great support for tabbed applications like this in the form of the QTabWidget class and subclasses of this, like QTabBar, which allows more control over the tab bar itself. Included in the application were a number of different functions, which all could have been standalone QWidget applications themselves, but in the interest of testing the usability of the application across multiple Operating Systems and forms all functions were added. There are five tabs, one tab with tabs within that tab, to show the expandability of the application. Firstly we will discuss the way Qt creates GUIs and the feature set for doing so.

QWidget

The QWidget class is the base class for all user interface objects and is integral to the functionality of Qt. The demonstrator application features six independent QWidgets. The widgets receive mouse and touch inputs, and paint themselves on screen. Each widget is clipped by its parent and by the widgets in front of it. A widget with no parent is called a window, or a top level widget, and will be drawn independently with whatever children it contains. Each widget can also be styled independently of each other.
The demonstrator application was all based around the Qt QTabWidget class, a class that is designed for creating tabbed applications. A tabbed interface allows multiple widgets to be contained within a single window, and uses tabs as a way to navigate between these widgets. It creates a stack of tabbed widgets, a tab widget consists of a tab bar and the area of the widget, which is where the content of each widget is displayed. This page area behaves the same as a standard QWidget. By default in Qt the tabs are set to the top of the screen, but this can be changed to any side. The demonstrator application consists of five tabs, which is a very standard application design across Android, iOS, OS X and other platforms.
Layouts

Qt has a layout management system to provide an easy way to arrange widgets in the GUI. When creating a parent widget with multiple children layouts can be used to arrange the widgets. There are many options such as grid layouts, horizontal layouts and box layouts. These layouts will automatically position and resize widgets based on the space available in the window, ensuring that they are always arranged correctly and every widget is usable. Layouts can be created within layouts to create complex interfaces that will automatically be arranged based on the size of the screen, ensuring a nicely formatted GUI across different systems and platforms. In
the code below taken from the GUI Widget it is demonstrated, a QVBoxLayout (Vertical box layout) lines up widgets vertically in a box formation. Note that the first child widget to be added is the highest up in the layout, and the QPushButton is last, as it is the last widget added to the layout. This layout system is quick, and easily interchangeable.

    // GUI Tab
    QWidget *guiWidget = new QWidget; // create the GUI Widget
    QVBoxLayout *guiLayout = new QVBoxLayout; // Create a new QVBoxLayout
    guiLayout->addWidget(tSpinBox); // Add the QSpinBox to the layout
    guiLayout->addWidget(tSlider); // Add the QSlider
    guiLayout->addWidget(tDial); // Add the QDial
    guiLayout->addWidget(tButton1); // Add the QPushButton
    guiWidget->setLayout(guiLayout); // Set the layout of the GUI Widget to the new QVBoxLayout

    Figure 11: Creating a layout for a QWidget

Signals and Slots

Signals and slots are a convenient way for objects to communicate. A signal is emitted when a particular event occurs and a slot is a function that is called in response to a certain signal. Here is the code that I used to link these three Objects: (movement diagram)
// Sliders + Spinbox connections
QObject::connect
(tSpinBox,SIGNAL(valueChanged(int)),tSlider,
SLOT(setValue(int)));
QObject::connect
(tSlider,SIGNAL(valueChanged(int)),tSpinBox,
SLOT(setValue(int)));
QObject::connect
(tSlider,SIGNAL(valueChanged(int)),tDial, SLOT(setValue(int)));
QObject::connect
(tDial,SIGNAL(valueChanged(int)),tSlider, SLOT(setValue(int)));
tDial->setValue(65);

Figure 12: Slots and Signals example

When the value of the QSpinBox (tSpinBox) is changed, its integer value is passed to the QSlider (tSlider), and similarly when the QSlider is changed the QSpinBox value is updated. The QSlider and QDial (tDial) are double linked in the same way ensuring all three objects will always hold the same integer value. We set the value of the QDial (tDial) after the connections have been made between the objects, and each object will then hold this value. We could just as easily use the setValue function on the QSpinBox or the QSlider as all three objects are now mutually dependent on each other.
There was also a quit button in this tab that, when clicked/pressed, would exit the entire application. This also made use of the Signals and Slots as seen below:

```cpp
QObject::connect(tButton1, SIGNAL(clicked()), &a, SLOT(quit()));
```

This button could have been added to all tabs if necessary as an easy quit function, regardless of which tab the user is using. It is a good example of the convenience of signals and slots in Qt.
Launching the application

When the application is first opened we see our first example of a Qt GUI class in action, the QSplashscreen:

Splashscreen

The splashscreen was created using the Qt Class QSplashscreen. Many applications display a splash screen as the application is launching, for ads or to mask a long loading time. In this application a hybrid Qt Android logo is used. Most splashscreens consist of a graphic that is shown for a few moments before the main window appears. It is useful if there is a long loading time in the application, and can also be used to serve ads.

```cpp
#include <QSplashScreen>

class SleeperThread : public QThread
{
public:
    static void msleep(unsigned long msecs)
    {
        QThread::msleep(msecs);
    }
};

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    // Splashscreen
    QSplashScreen *splash = new QSplashScreen;
    splash->show();
    SleeperThread::msleep(2000);
    topTabWidget->show();
    splash->finish(topTabWidget);
    delete splash; return a.exec();
}
```

Figure 14: Splashscreen code

The above code shows how the the Splashscreen is created, displayed for 2 seconds, then the topTabWidget (the window of the Application) is called. After this the Splashscreen is stopped to reveal the Application.
Tab 1. GUI Tab

The first tab was designed to utilise the GUI features that Qt offers, the aim of this to test to see how they are rendered for each platform. The Qt GUI classes QSpinBox, QSlider, QDial and QPushButton were used for this purpose. The QSpinBox allows for two ways to change its value, a number can be typed (via physical or virtual keyboard) into the box, or the up and down arrows can be clicked to increment or decrement the value. This was useful to test physical and virtual keyboards when deploying across platforms. I linked the value of the QSpinBox, the QSlider and the QDial using “Signals and Slots”. [13]

Tab 2. Clocks Tab

The clocks tab was added to display the Qt application using system information from all the platforms to perform a function. In this case the application is accessing the system clock on each platform and using it to display the current time, with an analog or digital clock. The clocks tab consists of a second QTabWidget (clocksTabWidget) which is a child of the top QTabWidget (topTabWidget), and is a parent to the two clock widgets. This displays
the flexibility of working within these widgets. There was originally only a Digital Clock in this tab and it was a standard QWidget. Then to display the QTabWidget functionality a child QTabWidget was created, adding two tabs at the bottom, one for the Digital Clock, one for an Analog Clock. The tabs were set at the bottom of the application instead of the top, which it does by default. This was for a better user interface. This created a nice interface for the tab where a digital or analog clock could be selected. The code for the clocks was adapted from the Qt sample code and implemented as part of a larger tabbed application.

To set the tab bar position the following was used:

clocksWidget->setTabPosition(QTabWidget::South);

The size of the tabs was also manually set like the tabs in topTabWidget, to keep the design coherent across platforms.

On the Android platform images were deployed to serve as backgrounds to the clock faces, in Linux, Mac and Symbian the clocks kept the native look.

![Figure 16: The Clocks tabs](image)

**Tab 3. Sketch Tab**

The aim of the sketch tab was to compare the performance of
the different inputs across the platforms. For desktop Operating Systems such as Mac OS X and Windows the mouseClicked event would let the user draw on the sketchpad and would stop drawing when the mouseClick was released. On the mobile touchscreen interfaces touching the screen was the equivalent event. The results of this were very impressive and the sketchpad performed very well with both types of input. The smaller screen on the Android device made using the sketchpad harder than desktop but it did function correctly. An Android tablet would be ideal to utilise the touch interface correctly. The sketch tab also uses a QPushButton to clear the drawing the user has made. The sketch tab was useful in assessing the lag in the application.

![Figure 17: Sketch Tab and Calculator Tab](image)

**Tab 4. Calculator Tab**

The code for the calculator was also taken from the sample Qt code and its purpose was to demonstrate Qt performing calculations on the native hardware. The calculator ran perfectly on all platforms and was not perceivably different in speed to any of the native calculators. The calculator also performed correctly across the touch/mouse interfaces, and was compatible with the physical
and virtual keyboards on all devices. It is a good example of what could be used as a standalone functional application to be used on a multitude of platforms. A QGridLayout is utilised here for arranging the buttons.

**Tab 5. Maps Tab**

The maps tab was designed to demonstrate the QtWebKit module in action. [14]QtWebKit provides a Web Browser engine to embed content from the Internet into applications. It can be used to render HTML, XHTML and SVG documents. It can also be scripted with JavaScript and styled using CSS. The QWebView class is the main widget component of QtWebKit and it provides a widget that is used to view Web documents. For an interesting use of QWebView the Google Maps JavaScript API was utilised to create a map specifically for the application. The map provided street view for the application, and also used geolocation to determine the position
of the user.

6 Deployment

6.1 Deploying Qt Applications

There are two different methods of application deployment of applications written in Qt, static linking and dynamic linking. Deploying a Qt Application is a very streamlined process, where you set the Application to release mode and choose the target platform. There can be multiple target platforms, and it is a very convenient and quick way to compile an Application for multiple operating systems. When deploying an application, the Qt libraries are needed for the application to function. This is how Qt ensures the applications function correctly on each platform, as the libraries, which are OS independent, are responsible for the creation of the GUI. There are two ways of deploying the libraries with your application, statically linking the libraries, i.e. including the required libraries in the application executable, or dynamically linking, which will utilise shared Qt libraries. When deploying dynamically linked Qt applications, the application will check if the required libraries are present on the system, i.e. QtCore, QtGUI, and if not will need to install them. The Qt libraries can be included in the application zipped package if required, or the application can retrieve the libraries from a repository. This is shown in the Android deployment, where the libraries are downloaded from a third party source, Ministro.

6.1.1 Static linking

When you link an application statically, all the libraries needed by the application are packaged in the executable. This means that the code of the libraries gets copied into the package. This has many advantages and disadvantages, the main advantage being that will
be compiled into a single application file, which has the following pros and cons:

[21]

- A single file has to be deployed to the system
- The application will be independent of any other Qt applications or frameworks that may already exist on the system.
- The application will have a slightly faster start up time.

- The application will be far bigger in size, as the required Qt libraries are included in the package
- The Qt libraries cannot be updated, as they are part of your application, and do not exist centrally, independent of your application.
- If multiple instances of this application, or other statically linked Qt applications, more RAM will be consumed, as the Qt code is loaded separately for every instance.
- Nokia restricts the licenses of statically linked Qt applications so a commercial license is required.

Figure 19: Structure of a statically linked Qt application
6.1.2 Dynamic Linking

Dynamic linking is the most widely used method and offers flexibility for users and developers. It does not include the full source code of the libraries in the executable. Instead, it includes only a reference to the used members of a library. This is the most widely used method for deploying Qt applications. When you start up a dynamically linked application, the linker will search the system for the libraries that are referenced by the application, and then use them, from the shared library.

Pros

- The application will be smaller in size, important for mobile applications with limited data allowances.
- The common Qt libraries can be updated easily, and the update will apply to every dynamically linked application accessing these libraries, very convenient for security fixes and updates.
- If multiple instances of Qt applications are running at once the shared Qt libraries are only loaded once, leading to less RAM use.

Cons

- Deployment is not as easy as with statically linked applications, and will require testing on different systems to ensure the application will work on the target.
- If the required Qt libraries are not available on the system, a method of obtaining them is required, otherwise the application will not run (as we will see with Ministro)
6.2 Deployment of Demonstrator

6.2.1 Linux

Linux was one of the four platforms supported in my main build environment. I was using the Ubuntu distribution my development platform for the Symbian, Maemo, Android and Linux deployments of the Demonstrator application. The Demonstrator application performed excellently on Linux. I used the Sketch Tab of my application to test mouse events extensively on all platforms. The QMouseEvent class contains the parameters that describe Mouse Events. Mouse Events occur when a mouse button is pressed or released within a widget, or when the mouse cursor is moved. A QMouseEvent was the equivalent to a touchEvent on a touchscreen. MouseHeld was the equivalent to a held touchEvent on a touchscreen. Performance on Linux was indecipherable from a natively
written application as was the look.

The application was written and compiled using Qt Creator. Three development environments were installed for Qt Creator, Linux, Android and Symbian/Maemo. Building the application for Linux the target is selected, and placed in release mode. The Demonstrator source code was compiled, and deployed to Linux. The deployment was straightforward as the application was developed and tested on the Linux platform.

Figure 21: The three installed targets in Qt Creator

The default native Ubuntu Linux theme is called Ambiance. It features dark menus, panels and rounded widgets with salmon coloured highlights.
The size that the Application was set specifically for the Linux platform using an `#ifdef` statement, which means “if defined”. This command is used for platform specific changes. This was important to define for deploying to mobile device resolutions as well as desktop resolutions without needing to modify the code.

### ifdef statements

```c
// To set Application size for Linux
#ifdef Q_OS_LINUX
QTabWidget->setFixedSize(600,800);
#endif

// To set Application size for multiple platforms
#ifdef Q_OS_LINUX
QTabWidget->setFixedSize(600,800);
#elif Q_WS_SYMBIAN
QTabWidget->setFixedSize(400,400);
#elif Q_OS_ANDROID
QTabWidget->setFixedSize(800,400);
#endif
```

The application was deployed in this manner, and below are the results of the deployment.

### Results

- The physical keyboard was compatible as was the MouseEvents.
- The physical keyboard was tested in the QSpinbox, where an integer value between 0 and 130 could be entered to change the position of the QSlider and the QDial.
• Using the Linux APIs it supported the native Linux look.

![Figure 22: Linux: Menu bar, QTabWidget, QSpinBox and QSlider](image)

• System clock is accessed for the two clocks.

![Figure 23: Linux: Clock Widgets](image)

• The QMouseEvent class handles all the mouse input correctly. This was tested in the Sketch Widget and was very responsive.

• The QtWebKit library works correctly and QWebView uses the systems network connection for the Maps widget.
• The calculator works correctly, and resizes to fill the widgets available area.

6.2.2 Mac OS X

Mac OS X is a UNIX platform and is similar in behaviour to other UNIX-like systems. The main difference is that the main windowing system is not X11. Mac OS X uses its own native windowing system that is accessed using the Carbon and Cocoa APIs. Application development for OS X is usually done using Apples Xcode Tools, which is available as an option on every version of the operating
system. Apple have modified the GCC compiler to include with Xcode.

The compiling of the application was performed on Mac OS X with Qt Creator, using the source code of the Demonstrator application. It is not currently possible to deploy OS X Qt applications directly from Linux, the development environment being used for Android and Linux deployment. OS X and Linux were installed in a dual-boot configuration on the development machine. The Demonstrator source code was compiled, and the OS X target was targeted in release mode.

The native OS X look that is desired is called “Aqua”, it uses the Lucida Grande font as the standard system font. White, blue and grey are the primary colours of the Aqua look.

In OS X, an application must be built and run from a bundle. A bundle is basically a directory structure that is represented by a single file when viewed in the Finder. The bundle typically contains the executable and all the required resources for the executable to run.

A bundle is very convenient, as it provides the user with a single file, that provides easy drag and drop to the applications folder, and users do not need to navigate the inner file structure to find an executable. Within this bundle we package the required Qt libraries. In this application the Qt Libraries QtCore, QtGUI and QtWebKit required. These libraries are placed in the “Contents/Frameworks” folder, where the Qt application will look for them.
Figure 26: Package deployment structure in OS X

The application was deployed in this manner, and below are the results of the deployment.

Results

- The Aqua look is implemented across all widgets, the main window is identical to a native OS X application, and all buttons and objects are rendered identical to the OS X equivalents. This is performed by the Qt class QMacStyle, this class is a member of QtGUI, and it is implemented as a wrapper to the HITheme APIs, which are present on all Mac OS systems. These APIs allow applications to be styled by the current OS X theme in use by the System. This style cannot be

- The QMouseEvent class handles all the mouse input correctly. This was tested in the Sketch Widget and was very responsive.

- The QtWebKit library works correctly and QWebView uses the systems network connection for the Maps widget.
• The calculator works correctly, and resizes to fill the widgets available area.

• System clock is accessed for the two clocks.

6.2.3 Symbian

Symbian was deployed to the emulator supplied in the Nokia Qt SDK. The native look was supported, however the tabs did not work very well in this deployment. The native symbian tabs overlapped by default, and changing of the font sizes or QTabBar parameters did not produce much better results. To resolve the issue, a theme was designed to create better tabs. The application no longer looked native, instead it mimiced the Android design which is discussed
Figure 29: OS X: Maps Widget and Calculator Widget

Figure 30: OS X: Clock Widgets
later. This led to a very functional application, which had a better tabbed Widget implementation than what was natively supported.

Touch response in Symbian appeared to work well in the emulator, but as there was not a test device for Symbian there was no way of checking physically. The included documentation for the Symbian SDK says that a test device is desirable but the application should perform as it does in the emulator on devices. The Symbian size and tabs had to be manually set using #ifdef statements, as the
setMaximised function didn’t work correctly.
7 Android

Deployment of Android was a difficult task, given the fact that it until recently it was impossible to run any form of C or C++ code on Android. Investigation of the feasibility of deploying Android applications using Qt was one of the main goals of the project. With the community efforts taking place in this area it proved possible to run the demonstrator application on the Android device, but it was a far less polished deployment than we saw with the previous applications. The main parameters that were needed did function correctly however, and with some modifications a workable application was eventually deployed.

7.1 The Qt-Android Port

Porting Qt to Android, which is essentially a Java based operating System, was made possible by the release of the Android NDK (Native Development Kit). This allows implementation of applications running machine native code such as C or C++ under the Dalvik Virtual Machine. With this release the Qt code can run using Android Java wrappers to call the C++ functions and despite being in very early development has proven quite effective. The other factor in the ease of porting Qt to Android is the continued development of Qt-Lighthouse, a project that is currently under development by Nokia. The porting of Qt to Android makes it a very interesting prospect for developers to port Qt to iOS too, as iOS and Android are arguably the most advanced mobile Operating Systems available today, with certainly the most developers actively creating apps for their ecosystems. Currently the iOS App Store has approximately 350,000 applications available, while the Android Market has at least 100,000 and is gaining ground on iOS. From an application
developers point of view, being able to target Qt written C++ code to both these huge application ecosystems with no modification if an exciting prospect, and would ensure the survival and relevance of Qt in light of Nokia’s recent apparent abandonment of the framework.

7.2 Qt Lighthouse

Qt Lighthouse is a version of Qt which uses its own framebuffers to draw widgets and windows. This means that it is independent from any specific platform or windowing system. To make it compatible with a platform a simple plug-in needs to be written which would push Qt’s framebuffers to screen, and forward user input (keyboard, mouse, touchscreen) to the Qt application. So, "Qt for Android" is Qt Lighthouse with an adapter to receive function calls from Android OS, as well as calling the Qt framebuffers to the screen. Because Android simply does not provide a lot of system functionality (including application start-up and access to screen and user input events) to native binaries (i.e. C++/Qt/NDK), The adapter consists of two parts, one in C++ and another one is a set of Android SDK (Java) classes. The Java classes initialize the environment, load the Qt app as a dynamic library (and other libraries via Ministro, Qt Webkit in my case), runs the Qt application and then proxies screen, input and all necessary SDK functionality for it via JNI. This provides a seamless integration, where Qt writes all pixels to screen and receives all user input from the Android interface to respond to.

7.3 Android NDK

The Android NDK is a toolset that was released to allow certain parts of Android applications run in native C or C++ code. Up to this point Android applications could only be written with Java
syntax and using the Android libraries to access the device and control the applications. Android applications run in the Dalvik virtual machine. The Dalvik virtual machine was designed to be suitable for machines that are constrained in terms of memory and processing speed, which is what Android is targeted at, mobile devices such as Smartphones and tablets. Before execution all applications are converted into the .dex format which stands for Dalvik Executable. The NDK provided a set of tools that could be used in conjunction with the Android SDK to package .apk files for the Android operating system. The Qt port uses the JNI (Java Native Interface) to access the Android APIs provided in the NDK. This marries the convenience of writing native code, or reusing already written code in C/C++ while utilising the Android framework. The Java Native Interface is a programming framework that enables Java code running in a Java Virtual Machine, in our case the Dalvik virtual machine, to be called by native applications and libraries written in other languages, so the port utilises C++ code to call the relevant Java code to interface with the Android OS. In the JNI framework, native functions are implemented in separate .cpp files. So in this port Java<>JNI functions are implemented to connect the two infrastructures. The Android NDK is only compatible with Android 1.5 (Cupcake) and later, but this is not an issue, as only very dated developer versions would run anything less than that version. When I was creating applications for Android I deployed from API levels 7 - 10, which have the highest current penetration of the market. The Android device I was deploying to had the API level 7 installed, and I created emulators with the Android AVD tool to test API levels 8 and 10 (No emulator available for level 9).

7.4 Creating the Android Build Environment

The Android Build environment was created by utilising a mod-
Figure 34: Flow diagram of the Qt-Android port
ified version of Qt Creator, it was a result of a community effort to port Qt to Android.

It required installation of the following packages to Ubuntu 10.10:

- The Android SDK,
- Necessitas for Qt Creator
- The Android NDK, version r5b
- The OpenJDK
- The Apache Ant tool
- Ministro installed on the Android device

Ministro is an Android Application deployed on the Android Market which is a system wide installer of the Qt libraries. When a Qt application written for Android needs a Qt library to run, this acts as the installer of the library. When a library is downloaded once i.e. QApplication, it does not need to be installed again regardless of how many Qt applications are installed on the device. This helps keep the size of the Qt Android applications small by not bundling the libraries with the .apk file. And the application will only download the required libraries. This is important with mobile limits on data downloads, as the full Qt libraries would be prohibitively large.

The port had to be performed on a Linux machine, so our four targets after creating the Android port were Symbian, Maemo, Android and Linux Desktop. A Mac deployment is possible from the same machine as it is dual booting OSX and Linux. The port is a combination of Qt Lighthouse and the Android NDK to create connections so that the Java code used to initialise the application can call the Qt code with these connections, achieved using the JNI (Java Native Interface).
7.5 Java Connections

QtActivity.java connects with Ministro to load the Qt libraries that are downloaded through Ministro. Ministro is a system wide Qt shared libraries installer/provider service. It acts as a bridge between your apps and Qt libraries. It has to be installed via the Android Market and the user is prompted to do this automatically when an application written with Qt is installed. Only the Qt libraries that are needed are installed, and the libraries that are needed must be defined when creating an app. All applications need the QApplication library and the QtGUI library. Other libraries such as QWebKit may also be needed.

QtActivity.java ensures the required libraries are present on the device, if not it will prompt to download them. Then it loads the libraries and runs the application. However there is a loop in the code that if the libraries are unavailable i.e. no network connection to download them, that it will exit the application. QtActivity.java
also checks the API version that the device is running:

The device I was testing on was the Nexus One which currently runs Android 2.1 (Froyo) which is API level 7. It then returns the Qt libraries for the relevant API level: libandroid-"+apiLevel+".so".

```c
int apiLevel=android.os.Build.VERSION.SDK_INT;
if (apiLevel<4) // this should never happen 
    apiLevel=4; // Necessitas, we have a problem !
else
    if (apiLevel>5 && apiLevel<8) apiLevel=5; // android-6, and 7
    else
        if (apiLevel>8)
            apiLevel=8; //
        android >8 are compatible with android-8
    }
    libs[i]="/data/local/qt/plugins/platforms/android/libandroid-
        "+apiLevel+".so";
    m_ministroCallback.libs
        (libs,"QT_IMPORT_PATH=/data/local/qt/imports\tQT_PLUGIN_PATH=/data/local/q", "-platform\tandroid", 0,null); return;
```

Figure 36

### 7.6 Working Features

There was the deployment of the application written and deployed using the Qt Creator IDE. There is full touch screen functionality, as well as Native Keyboard for any text input. There was not an Android device with a physical keyboard available to test on but using the emulator physical keyboard performed correctly. Orientation was also a feature that performed as desired, so our
application rotates when device is rotated, this is convenient for applications that demand a wider screen, such as a media player application.

7.7 Unsupported features

Unsupported features include the Android native look, which
I were created as shown above using Qt Stylesheets. The GPS sensor or accelerometer could not be accessed as Qt Mobility, the class which makes these features available for Symbian, was not functioning correctly as the JNI connections were incomplete. IP Address tracing for the Google Maps Implementation was used instead, which tracked the location to the IP address of the network connection.

7.8 Duplicating the Android Look

Fonts

Android employs three fonts in its operating system, Droid Sans, Droid Sans Mono, and Droid Serif. They were created specifically for the OS. To use these fonts I accessed them directly through Qt, by giving the absolute path, as they could not be packaged as a resource. The Android fonts are stored in the devices “/system” folder so the path to Droid Sans is “/system/fonts/DroidSans.ttf” when running the application on the phone. Using #ifdef statements to access the font caters for deployment to other platforms. The fonts are also publicly available so I obtained them to use in the
Linux deployment which used the Android look. To add a font to the application we use the QFontDataBase class. The QFontDataBase class provides information to Qt about the underlying fonts available by the platform.

**Droid Sans**

Figure 40: Droid Sans example

```cpp
// Change font
#ifdef Q_OS_LINUX // for a Linux System
int id = QFontDataBase::addApplicationFont
( "/home/tiernan/Documents/addedfonts/DroidSans/DroidSans.ttf"
);
// the local path of the font on the Linux System
#else int id = QFontDataBase::addApplicationFont( "/system/fonts/DroidSans.ttf"
);
// for all other systems (Android)
#endif QFont newFont("DroidSans", 14, QFont::Bold, true);
// define the parameters of the font ("name", size, additional parameters (bold, italic, underlined)
QApplication::setFont(newFont);
// set font of the Qt application

This sets our applications font to Droid Sans for all displayed text.

**QTabWidget Customisation**

For the tabs in QTabWidget, it is fitting with the theme that they fill the screen, but by default empty space was kept on the right side.
To achieve this the Tab Bar had to be modified to define the exact size of the Tabs, and fill the screen.

```cpp
#ifdef __Q_OS_ANDROiD
  topTabWidget->setStyleSheet("QTabBar::tab{height:95px;width:96px;}");
#endif
```

By only stating the ifdef statement for Android, all other operating systems that are deployed to will behave as normal if they are not Android.

**Colour Scheme**

The next step was to get the Android tabbed application standard colour scheme correct. This was achieved in the same way as the tab sizes. By setting the style of the parent Widget of the application, `topTabWidget`. The colours were as follows:

```cpp
  topTabWidget->setStyleSheet("QTabBar::tab{height:95px;width:96px;background:black;color:white;font-family:DroidSans;font:27px;margin:0px;padding:0px;}
  QTabBar::tab:hover {background:yellow;}
  QTabBar::tab:selected {background:gray;color:white;}");
```

This set the selected tab to appear gray in colour, with the tab title font coloured in white. The deselected tabs appeared in black, with the font in gray, just as they behave in Android.

### 7.9 Result

On the whole the Android port was a success, and it was possible to get all the required features of the demonstrator Application functioning correctly. The Android look was a workaround but neverthe-
less a passable method for having an Android looking application, and keeping the overall benefits of Qt Application development.

The deployment of the native look had the following advantages:

- Flexible - can be changed on a per Widget basis
- Can maintain Qt look if needed
- Android look can be deployed to other platforms, as was done with Symbian in this project

Disadvantages

- Manual implementation
- Certain elements behave differently

Elements that had issues initially were the QWebView, as it needed an extra Qt library to be deployed, QtWebKit. The native Android virtual keyboard was capable of editing the QSpinBox and the touchscreen could be used to draw on the Sketch Widget with no lag.
8 Overall Results

The results of this project were successful. The Demonstrator application performed well on all platforms it was deployed to. Speeds on all platforms appeared no slower than native applications, and inputs were responsive in all iterations. For the desktop deployments, the native looks were utilised, Linux and Mac OS X looked and behaved like applications created natively, achieved by Qt using the native OS APIs. For mobile deployment there was more work involved, enabling and disabling different features, Symbian needed tweaking to make it more user friendly as the native Symbian feel wasn’t very intuitive. Android performed surprisingly well, given the experimental nature of the port. More platform specific tweaking went into this deployment but the native touch input provided a very usable interface. After applying styles to replace the lack of native support, there were solid results and it resembled a native application in look and feel. Physical and virtual keyboards performed perfectly, as did the different mouse interfaces, with trackpad and touchscreen. Native Android virtual keyboard working correctly. Native support from the platforms API, achieved the regular look of the platform. Android was the exception, the most technically demanding deployment: a port was utilised to deploy to the Nexus One device. The benefits of a solid cross platform framework with support for Android, the fastest growing mobile OS, are huge, and from a single source, a huge amount of systems can be targeted, and from a single machine.
9 Conclusions

9.1 Is Qt a viable alternative to native platform development?

Yes, from the research into using Qt as a multi-platform application development environment it is clear that Qt is capable of most functions demanded by a GUI application today, and is ready to be deployed across a multitude of platforms, that only stands to grow, with ports like the Android one utilised here in development. The build environment that has been created have is capable of deploying applications to 60% of the worlds Smartphones, as well as the three dominant desktop platforms. This makes for a very strong case for performing all development with Qt, as with its large collection of classes make it as modern a framework as any, and the added benefits of these classes performing the same across a multitude of platforms. As seen from this project, only minor cosmetic changes were really needed for each platform, all Qt functionalities performed flawlessly. Relying on the extensive Qt classes and leaving the compatibility to the framework can deliver excellent results, there is a strong community of developers behind Qt and if they keep expanding the scope and targets of the system, as I firmly believe it is the best in class for this type of development.
List of Figures

1  Role of an operating system .......................... 9
2  The multitude of competing operating systems .... 10
3  Growth in mobile subscribers in OECD countries 1990-
4  Installed base of Smartphone Application Users, 2009-
   2013, Source: [5] ........................................ 16
5  Mobile OS market share ................................. 18
6  Gadget ownership 2006-2010 (US) .................... 19
7  Overall Operating System market share [15] ....... 20
8  Fragmentation of the Android Platform. Source: [4]. 24
9  QWidget Structure ...................................... 36
10 Application Flow Diagram .............................. 37
11 Creating a layout for a QWidget ....................... 38
12 Slots and Signals example ............................ 39
13 Slots and Signals visual representation ............... 40
14 Splashscreen code .................................... 41
15 Splashscreen and GUI .................................. 42
16 The Clocks tabs ....................................... 43
17 Sketch Tab and Calculator Tab ......................... 44
18 The Maps View ........................................ 45
19 Structure of a statically linked Qt application ... 47
20 Structure of a dynamically linked Qt application ... 49
21 The three installed targets in Qt Creator ........... 50
22 Linux: Menu bar, QTabWidget, QSpinBox and QS-
    lider .................................................. 52
23 Linux: Clock Widgets .................................. 52
24 Linux: Sketch Widget and Maps WebView ............ 53
25 Linux: Calculator Widget .............................. 53
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Package deployment structure in OS X</td>
</tr>
<tr>
<td>27</td>
<td>OS X: Menu bar, QTabWidget, QSpinBox and QSlider</td>
</tr>
<tr>
<td>28</td>
<td>OS X: Sketch Widget</td>
</tr>
<tr>
<td>29</td>
<td>OS X: Maps Widget and Calculator Widget</td>
</tr>
<tr>
<td>30</td>
<td>OS X: Clock Widgets</td>
</tr>
<tr>
<td>31</td>
<td>Comparison of native tabs and manually styled tabs in Symbian</td>
</tr>
<tr>
<td>33</td>
<td>Symbian: Sketch Widget</td>
</tr>
<tr>
<td>32</td>
<td>Symbian: Clock Widget and Calculator Widget</td>
</tr>
<tr>
<td>34</td>
<td>Flow diagram of the Qt-Android port</td>
</tr>
<tr>
<td>35</td>
<td>Ministro handling installation of Qt libraries when launching a Qt application for the first time</td>
</tr>
<tr>
<td>36</td>
<td>Android: Sketch Tab</td>
</tr>
<tr>
<td>37</td>
<td>Android: GUI Widget</td>
</tr>
<tr>
<td>39</td>
<td>Android: Calculator Widget</td>
</tr>
<tr>
<td>40</td>
<td>Droid Sans example</td>
</tr>
<tr>
<td>41</td>
<td>Comparison of Qt and native Android tabs</td>
</tr>
</tbody>
</table>
10 References


[18] Molkentin 2006 - Qt 4, The Art of Building Qt Applications
I have included the main class, as well as the source for the Widgets. Please bear in mind that the included classes on the CD, excepting the main are provided sample code with the Qt framework and are NOT original work, but are supplied to understand the workings of the application. The main is also printed below.

Classes:
- Main (included)
- analogclock
- digitalclock
- button
- calculator
- sketchdialog
- sketchwidget
main.cpp

#include <QApplication>
#include <QTabWidget>
#include <QTabBar>
#include <QStackedWidget>
#include <QVBoxLayout>
#include <QPushButton>
#include <QSlider>
#include <QPushButton>
#include <QSpinBox>
#include <QDial>
#include "digitalclock.h"
#include "analogclock.h"
#include "sketchdialog.h"
#include <QThread>
#include <QWebView>
#include <QSplashScreen>
#include <QFontDatabase>
#include "calculator.h"
using namespace std;

// Splashscreen
class SleeperThread : public QThread
{
public:
static void msleep(unsigned long msecs)
{
QThread::msleep(msecs);
}
};

int main(int argc, char *argv[])
{
QApplication a(argc, argv);

// Widgets
QTabWidget *topTabWidget = new QTabWidget(); // Top Tabbed Widget
QWidget *clocksTab = new QWidget(topTabWidget); // Tab number 1 (Clocks)
QWidget *sketchTab = new QWidget(topTabWidget); // Tab number 2 (sketch)
QTabWidget *clocksWidget = new QTabWidget(clocksTab); // Clocks Tab Widget
QWidget *inTab1 = new QWidget(clocksWidget); // Digital Clock Widget
QWidget *inTab2 = new QWidget(clocksWidget); // Analog
QWidget *inTab3 = new QWidget(topTabWidget); // Analog

// Splashscreen
QSplashScreen *splash = new QSplashScreen;
#ifdef Q_OS_ANDROID
splash->setPixmap(QPixmap("/sdcard/download/qtandroid.png"));
#else
splash->setPixmap(QPixmap("/home/tiernan/Desktop/qtandroid.png"));
#endif
splash->show();
SleeperThread::msleep(2000);

// Change font
#ifdef Q_OS_LINUX
int id = QFontDatabase::addApplicationFont("/home/tiernan/Documents/addedfonts/DroidSans/DroidSans.ttf");
#else
int id = QFontDatabase::addApplicationFont("/system/fonts/DroidSans.ttf");
#endif
QFont newFont("DroidSans", 14, QFont::Bold, true);
QApplication::setFont(newFont);

// Digital Clock
DigitalClock *dClock = new DigitalClock(inTab1);
QVBoxLayout *dClockLayout = new QVBoxLayout;
dClockLayout->addWidget(dClock);
inTab1->setLayout(dClockLayout);

// Analog Clock
QVBoxLayout *aClockLayout = new QVBoxLayout;
AnalogClock *aClock = new AnalogClock(inTab2);
aClockLayout->addWidget(aClock);
inTab2->setLayout(aClockLayout);

// Calculator
QVBoxLayout *aCalcLayout = new QVBoxLayout;
Calculator *calc = new Calculator(topTabWidget);
aCalcLayout->addWidget(calc);
inTab3->setLayout(aCalcLayout);
// Clocks Tab Widget
QVBoxLayout *clocksTabLayout = new QVBoxLayout;
clocksTabLayout->addWidget(clocksWidget);

 clocksTab->setLayout(clocksTabLayout);
clocksWidget->addTab(inTab1, ("Digital"));
clocksWidget->addTab(inTab2, ("Analog"));
clocksWidget->setTabPosition(QTabWidget::South);

// Sketch Tab Widget
SketchDialog *sketch = new SketchDialog(topTabWidget);
QHBoxLayout *topLayout = new QHBoxLayout;
topLayout->addWidget(sketch);
//QVBoxLayout *bottomLayout = new QVBoxLayout;
//bottomLayout->addWidget(sket
chLabel);
QVBoxLayout *sketchLayout = new QVBoxLayout;
sketchLayout->addLayout(topLayout);
sketchTab->setLayout(sketchLayout);

// Quit Button
QPushButton *tButton1 = new QPushButton("QUIT");
QObject::connect(tButton1, SIGNAL(clicked()), &a, SLOT(quit()));
//tButton1->setStyleSheet("background: white;" demonic

// GUI Sliders + Spinbox
QSpinBox *tSpinBox = new QSpinBox;
//tSpinBox->setStyleSheet("background: white;";
QSlider *tSlider = new QSlider(Qt::Horizontal);
QDial *tDial = new QDial;
tSpinBox->setRange(0, 130);
tSlider->setRange(0, 130);
tDial->setRange(0, 130);

// Sliders + Spinbox connections
QObject::connect(tSpinBox, SIGNAL(valueChanged(int)), tSlider,
SLOT(setValue(int)));
QObject::connect(tSlider, SIGNAL(valueChanged(int)), tSpinBox,
SLOT(setValue(int)));
QObject::connect(tSlider, SIGNAL(valueChanged(int)), tDial,
SLOT(setValue(int)));
QObject::connect(tDial, SIGNAL(valueChanged(int)), tSlider,
SLOT(setValue(int)));
tDial->setValue(65);

// GUI Tab -> adds all relevant widgets
QWidget *guiWidget = new QWidget;
QVBoxLayout *guiLayout = new QVBoxLayout;
guiLayout->addWidget(tSpinBox);
guiLayout->addWidget(tSlider);
guiLayout->addWidget(tDial);
guiLayout->addWidget(tButton1);
guiWidget->setLayout(guiLayout);

// Maps Tab
QWidget *mapsWidget = new QWidget;
QVBoxLayout *mapsLayout = new QVBoxLayout;
QWebView *mapsWebView = new QWebView;
mapsWebView->load(QUrl("http://allflags.ie/maps/ipsv.html"));
mapsLayout->addWidget(mapsWebView);
mapsWidget->setLayout(mapsLayout);
//mapsWebView->load(QUrl("http://allflags.ie/maps/locationwatcher.html")); // alternate Google Maps

#ifdef Q_OS_LINUX // for tab icons in linux
QPixmap guiPixMap("/home/tiernan/Downloads/Icons/icons/116-controller.png");
QPixmap clockPixMap("/home/tiernan/Downloads/Icons/icons/11-clock.png");
QPixmap sketchPixMap("/home/tiernan/Downloads/Icons/icons/16-line-chart.png");
QPixmap chatPixMap("/home/tiernan/Downloads/Icons/icons/08-chat.png");
QPixmap mapsPixMap("/home/tiernan/Downloads/Icons/icons/07-map-marker.png");
#else // ... Android
QPixmap guiPixMap("/sdcard/download/116-controller.png");
QPixmap clockPixMap("/sdcard/download/11-clock.png");
QPixmap sketchPixMap("/sdcard/download/16-line-chart.png");
QPixmap mapsPixMap("/sdcard/download/07-map-marker.png");
#endif

QIcon icon1(guiPixMap); // tab icons
    int index1 = topTabWidget->addTab ( guiWidget, "GUI");
topTabWidget->setTabIcon( index1, icon1);
QIcon icon2(clockPixMap);
    int index2 = topTabWidget->addTab ( clocksTab, "Clock");
topTabWidget->setTabIcon( index2, icon2);
QIcon icon3(sketchPixMap);
    int index3 = topTabWidget->addTab ( sketchTab, "Sketch");
topTabWidget->setTabIcon( index3, icon3);
QIcon icon5(mapsPixMap);
int index5 = topTabWidget->addTab( mapsWidget, "Maps");
topTabWidget->setTabIcon( index5, icon5);

// Top Tabs Widget
#ifdef Q_OS_ANDROID // Android style
topTabWidget->setStyleSheet("QWidget{background-image: url(/sdcard/download/background.png)}QTabBar::tab
{height:95px;width:96px;background:black;color:lightgray;font-family:DroidSans;font:27px;margin:0px;padding:0px;}
QTabBar::tab:hover {background:yellow;}
QTabBar::tab:selected {background:gray;color:white;}"");
clocksTab->setStyleSheet("QTabBar::tab
{height:95px;width:225px;background:black;color:lightgray;font-family:DroidSans;font:18px;margin:0px;padding:0px;}
QTabBar::tab:hover {background:yellow;}
QTabBar::tab:selected {background:gray;color:white;}");
sketch->setStyleSheet("QWidget { height: 30px; width: 30px;border: 7px solid gray;background:white;alignment:center;}");
inTab1->setStyleSheet("QWidget {background-image: url(/sdcard/download/clock.png);}
QWidget {border: 3px solid gray;border-radius: 80px;}");
inTab2->setStyleSheet("QWidget {background-image: url(/sdcard/download/clock2.png);}
QWidget {border: 3px solid gray;border-radius: 80px;}");
#endif

topTabWidget->addTab(inTab3, "Calc");
topTabWidget->show();
splash->finish(topTabWidget);
delete splash;
return a.exec();