Qt AND ANDROID

Let's compare by building an embedded system
with a music player app on a NXP i.MX 8 Quad board.

Witekio

Executive summary

This white paper all started with Witekio software engineers wanting to tackle a recurring question in the engineering community: **should we use Linux and Qt or Android Native for our next embedded product?**

Qt and Android are often complementary (automotive, smartphone) and Qt is platform agnostic*. Witekio teams complete an equivalent number of Android and Linux/Qt projects. Everyone has their favorite, but no one had ever achieved an exhaustive and unbiased comparison.

That is what Julien, Erwan and Stephen decided to do.

It was all about comparing **Android and Qt, evaluating each solution's advantages and drawbacks** from the point of view of embedded software developers.

To achieve this, Witekio developers team decided to build on a real-life example: **develop a Music player app on an NXP i.MX 8 Quad board.**

Their work first started with Julien comparing **low-level OS choice between Android and Linux.** Then our 2 application developers, Erwan and Stephen, faced each other in a breath-taking match to develop the same application with Android and Qt. **They compared both solutions performance, step by step, and had heated discussions.**

Their conclusion:

First, from a **low-level perspective**, it appears that, on the chosen board, an **NXP i.MX 8 Quad**, Android's strengths do not make it when opposed to Linux performance (RAM, flash memory) and flexibility which allows drastic boot time optimization.

At the **applicative level**, however, it is far less obvious to decide between Android and Qt. Erwan and Stephen developed the same Music player app, **compared more than 15 criteria**, and had passionate debates. Their conclusion: **Qt wins by a short head mostly on the data sorting and filtering criteria.**

To know more about the debate details, you can **read the full article.**

*Qt supports over 12 OSes *https://doc.qt.io/qt-5/supported-platforms.html*

The same question keeps coming up for developer teams at Witekio: “Is it better to develop a user interface with Qt, or with Android?”
In this white paper, we’ll try to answer that question.
Rather than churning out ready-made facts and theories, we’ve decided to give you answers based on an actual example: developing a music player application on an NXP board that’s geared up to deal with connected objects, the i.MX8.
We’ll start by talking about the first step in any embedded application project: choosing your OS. Android or Linux?
Once this choice has been made, by Julien, you’re invited to watch the face-off between Erwan and Stephen:

 Julien, our embedded software expert

 Erwan, our Android expert

 Stephen, our Qt expert

This simultaneous development will give us a chance to really focus on the strengths and weaknesses of each framework.

PART 1

Choosing an embedded operating system for our Music Player App on i.MX.8

Most embedded apps used today function with a microprocessor and an embedded Linux or Android operating system (OS). Having an operating system gives them several advantages. Primarily, it shortens the development process. Developers can use lots of different libraries, tools, and frameworks that they're already familiar with to build and debug, like Qt, Electron, GDB, SSH, and others. Updates are usually easier and less risky long-term because the device has a file system that helps avoid potential write errors, particularly in case of a power outage.

However, if your application is running on a microcontroller, it is worth noting that as of January 2020 Qt can work in that type of environment. With the recent release of Qt for MCUs (https://www.qt.io/qt-for-mcus), Qt is available on an increasing number of platforms, including real-time operating systems like FreeRTOS. That tends to blur the frontier even more between microcontrollers and embedded microprocessors. The main difference is that microcontrollers are better at real-time tasks that necessarily require little computing power,but need to have extremely short response times (close to a few microseconds). On the other hand, microprocessors running Linux or Android are more practical for feature-rich applications, with complex user experience, business logic, and connectivity.

Here, we will focus on two classic OS: Linux and Android.

So our goal today is to choose between Linux and Android for a system with a microprocessor.

Is it better to use Linux or Android in terms of criteria like security, flexibility, and usability?

Julien
embedded software expert

Choosing your OS: The Android vs. Linux Face-Off

What are the arguments for using Android as the OS for our system?

As a general rule, makers of evaluation boards have a version of Android all ready for their evaluation kit (EVK) that you can download and install.

However, there are several problems. If you want to develop your own hardware, porting Android can end up being time-consuming and expensive.

Android needs a large amount of RAM: 512 MB minimum. It also needs an important size for the flash memory, with a minimum recommended of around 1 GB.

When it comes to updates, Android supports the Google Play Store on verified devices. However, getting a device approved for the Play Store is a long and costly process and you probably won't want to give the user access to it, but updating a package is as easy as downloading an APK (Android Package File) and installing it.

For OS updates, all the tools are already included in the latest versions of Android by default, including support for dual bank updates when properly configured.

Android is really an operating system developed for mobile phones. If the specific hardware you develop uses components that aren't usually a part of mobile phones (for example, a CAN bus controller), the Android framework won't let you just implement it.

This is especially visible when it comes to permissions and access controls: Android itself mostly manages permissions to use peripherals, not single files (although it is possible to do so if you have root access, which is quite dangerous). It works well out of the box, but don't expect a lot of customizability.
Also, Android has several embedded components that are necessary for mobile phones but completely useless for most connected objects. Removing these components without breaking Android's overall operation is extremely complicated. The overabundance of unused components also increases the attack surface of your connected object, for no good reason. One final issue is that the boot time for Android is generally much longer than for a classic Linux system, and difficult to optimize. So we can say that Android is practical for developing and deploying a demo, but not necessarily ideal for our finished product.

On the other hand, Android clearly has all the right tools for smartphone development. If your team is used to developing in Java or Kotlin, using Android in your product will let your high-level development team start working on the project immediately, with no ramp-up time! Also, if your product is similar to a smartphone, requiring a desktop, several applications, etc., Android might be a good choice.

What are the advantages of choosing Linux?

Linux is extremely flexible and customizable, and the skills to do it are more common than for customizing the Android image (which actually uses the Linux kernel). Almost all the components of the kernel and the chosen distribution can be removed to make something small that boots quickly and has a minimal attack surface.

Moreover, Linux is a good RAM saver. In most standard use cases with Qt, 128 MB RAM is already quite comfortable, while Android requires 512 MB minimum. This allows considerable hardware cost savings.

Linux is also a flash memory saver. A basic Linux system can easily fit into 10 MB and a system including Qt can easily fit into 500 MB even with all the modules.

The main way to update Linux is to use online repositories, much like the Google Play Store but for both OS (kernel) and software packages. You can make your own easily. But how to do it will depend on what package manager you use: apt, yum, dnf, pacman… It can get confusing. You may want to setup a custom solution, too, for example to support dual bank updates or to handle custom encrypted packages.

But it's easy to add additional libraries, you can customize the libraries you want to install, and you can even customize the kernel! The permissions system is also more flexible and powerful than Android's, and the access controls are much more advanced, which gives you access to top-level security, using users, user groups, and individual file permissions.

In short, Linux is very easy to customize. Maybe even a little too easy. It can be a little overwhelming, and you might not know where to start! For example, which Linux distribution do you want to use for your system: Alpine, Ubuntu, Debian, Gentoo, or something customized?

As a general rule, you should use Yocto if you want to create a fully customized distribution. Yocto ensures that your Linux image build process generates something valid that can be reproduced, analyzed and easily adapted for whatever device you want to produce. Yocto ensures that library and software dependencies are respected, and also lets you deploy an SDK so that developers can deploy and test their apps quickly.

Let's now wrap up the comparison between Android and Linux!

Android may be easier to implement at first, if:

• Your hardware platform supports Android out of the box.

• All your peripherals are natively supported by Android.

In this situation, you won't have to recompile the entire system or install additional libraries, and everything will work right away. But if you want to customize your system (and I think that's the best option) or use features that aren't available on Android by default, it 's more efficient to use Linux. In fact, I'd go so far as to say it's better to use Linux if you want a faster, more secure and cheaper product.

Linux has several advantages over Android:

• Linux can support systems with limited resources. Using Android is recommended for platforms that have at least 512MB of available RAM, but you can easily run a Linux and Qt with only 128MB of RAM. Linux with Qt also has lower persistent storage requirements. This allows for a lower bill of materials, especially when choosing Linux+Qt versus Android.

• It can greatly reduce the attack surface. If you don't need Bluetooth, you can remove all the Bluetooth drivers and compatibility from the kernel, which is much more difficult on Android.

• It lets you significantly reduce the boot time, much more than Android could. Times as low as 2 seconds have been achieved with basic customizations.

• You can (and I think this is important) build a customized system to fit your product, which reduces not only deployment size but also energy consumption.

Comparison of Boot Statistics for Linux and Android

Boot time optimization with Linux

It's fairly easy to find Linux system boot statistics. For an NXP i.MX 6 SABRE board, this article tells you how to go about it.

https://www.witekio.com/blog/challenge-called-boot-time/

i.MX 6-based boards are fairly common and used in a wide variety of products. This makes them a good reference when it comes to boot time.

In the article, they began with a startup boot time of over 22 seconds (which is honestly still faster than my smartphone). That's much too long for a product that frequently turns itself off. At that point, the image was just the "standard" Yocto image, without optimization. The goal was to reduce this to under 2 seconds.

The first step was to select a preferred graphical interface framework. The 3 considered choices were Qt, Cairo and Enlightenment. In the end, Cairo was chosen as the device didn't have any GPU which would make Qt suboptimal and Enlightenment took too much time to start.
The second step was to tweak the initialization process so that their application is started before the rest of the system. This effectively reduced the boot time to around 8 seconds.

The third step was to optimize the kernel. By choosing the right kernel configuration options, and particularly by removing whatever wasn't useful, they reduced boot time to around 6 seconds.

The fourth step was to optimize the configuration of the bootloader to make loading the kernel faster, increasing the SD card access clock frequency and replacing the reset chips That got it down to a 1.99 second boot time.

A 2-second boot, just by doing simple operations that also drastically reduced image size and the attack surface.
Their process is explained in greater detail in the article, particularly the tools they used and their references.

I'm convinced that they could have gotten the same kind of results with an i.MX 8.

Boot Time with Android

There are several techniques for optimizing Android. The most common are the same as for Linux, particularly the optimization of the kernel and the boot loader.

Unfortunately, during boot, Android has to preload a huge amount of necessary information. One technique to avoid this is hibernation, described here

http://jultika.oulu.fi/files/nbnfioulu-201810062898.pdf

The environment used is a little different (and granted, they were using an older version of Android), but they still managed to go from a 60-second boot to around seven seconds, which is remarkable. But even though that's a huge improvement, they did reach a limit: the technique required suspension of the system on the flash memory, so performance depended on the read speed.

Also, for the tests, they went into hibernation mode immediately after boot, which doesn't represent reality and leads to an underestimation of the boot time, since RAM use is much lower right after boot-up than it is after using an app.

What's more, this technique requires many more changes that are more technical than for Linux, and none of them allow a reduction of the attack surface.

More detailed information about boot time optimization in Android 8 is available here

https://source.android.com/devices/tech/perf/boot-times

Android? Linux? My OS recommendation for the Music Player App project

Android is a very good choice for starting the development of a product on an Evaluation Kit (EVK), while waiting for a customized board. It gives you a usable environment very quickly, and lets you deploy and test Android apps immediately at the beginning of the product's life cycle, for example, to create a proof-of-concept.

However, it tends to be pretty weak compared to Linux in terms of long-term security, maintainability, performance, and boot time, particularly on customized boards .

Qt provides all the necessary tools to ease development with Linux. It may also be used on nearly every other major embedded and desktop operating system (QNX Neutrino, Green Hills INTEGRITY, VxWorks, and more). Android has no real clear advantage there.

Both operating systems allow you to quickly get an environment up and running (if you use existing distributions) and deploy and test Qt applications immediately at the start of your product life cycle. That is perfect for producing a POC.

Android drivers are updated less frequently than Linux drivers. Android is more difficult to customize and extend, and security updates can take some time before being available (if at all) on your device.

And Linux needs far less RAM and flash memory which allows reducing the production costs.

So I recommend that, for our test, we use Linux as the OS.

Now that the choice is made, I'll let Erwan and Stephen compare Android and Qt at the application development level! I'm going to go buy some popcorn, pull up a comfy chair and watch. 😄

Part 1 Recap

	Android	Linux
Necessary RAM	512 MB	128 MB
Necessary flash memory	1 GB	512 MB
Customization	YES	NO
Boot time optimisation	7 sec.	<1.9 sec.

PART 2

Comparing Development of our Music Player on Android and Qt

Our test app, Music Player App, is a simplified music player that will let us compare development methods and tools for embedded software in C++/Qt and Java/Android.

Erwan and Stephen will develop the app simultaneously, and compare Qt and Android in terms of functions, ease of development and development time.

Our app has to perform the following functions, which will allow us to compare strengths and weaknesses in the two frameworks:

- Display a mobile interface, for interacting with the app.
- Select a file containing an artist's discography, so users can download songs to play.
- Display the song list and provide the usual filtering options (by artist, album or genre).
- Read each song's metadata to create a model within the app (both for displaying and filtering).
- Play tracks, both via a dedicated screen with all the usual controls (pause/play, progress bar, volume control, and next/previous track) and a sidebar on other menus that includes minimal controls (pause/play) and the option to return directly to full-screen.

In this paper, we're also going to talk about the "Integrated Development Environment" (IDE) for each of these frameworks, provided by their editors: QtCreator by The Qt Company and Android Studio by Google.

Stephen
will develop in Qt,

Erwan
will develop in Android

Language versions
Qt Application:
C++11/Qt 5.13 -
Android Application:
Java 8, Android
API 28

ANDROID AND Qt, LET'S COMPARE.

1/ Architectural Breakdown

For development, our media player application separates its functions into two modules, according to the SOLID best practices for encapsulation:

• One back end module to handle the song files, compiled as a module exposing an API, accessible by QML.

• One interface module, acting as the main application, which presents the various controls to the user.

This separation of functions is even more practical with Qt, because the Qt framework uses two languages, C++ and QML, and each has its own separate purpose. The back end module is written in C++ to facilitate interaction with the OS and the many modules within the Qt framework that supports the handling of specialized files (media audio files in our case). The GUI module is written in QML, to support the organization of components in a declarative, natural way (a component contained within another component is automatically a "child" and will be automatically positioned relative to its "parent")

With Android, we see the utilization of a dedicated markup language (XML) to define the screens. Naturally, the app will be composed of a UI module that encapsulates these definitions and a back-end module that encapsulates the data manipulation. Remember that, unlike Qt, the XML references are compiled on the go and made available on the Java end for better auto-fill.

As I understand it, since you don't have access to that kind of verification, you won't catch any little syntactic errors until you actually run the app. Isn't that awkward?

You're only partly right, since a QML compiler actually exists. It will catch errors related to the QML syntax and semantics, but indeed errors with bindings, between QML components or even with the C++ back end, and JS errors aren't verified upstream.

As the compiler is opt-in, this method has its advantages, like a shorter compilation time during development (since files are simply deployed with the rest of the app, then uploaded on demand).

Architectural Breakdown: Qt 1 // Android 1

2/ Mobile Style Interface

2.1/ From Design to Implementation

Using Qt Design Studio lets the designer create their interface using their favorite graphic design software (Photoshop or Sketch, for example), then import the design file directly into Design Studio, which then automatically creates the corresponding QML components. It really speeds up this phase of development. Integration with the rest of the app is then mainly done through QML bindings with the back end module, in the case of Music Player so that it can handle audio files.

Argh, I don't have anything like that, so it probably took me a lot longer. I couldn't use your resources and original graphics in Photoshop. In fact, instead of Photoshop, I exported the PSD file in Zeplin. For me, integration was conceptually the opposite. You referenced your C++ API with QML, while I pulled the references for the graphic components from the Java end. Okay, I admit it, it's a lot wordier. That said, there is a binding available now with Android Jetpack, but I didn't use it.

From Design to Implementation > Qt 1 // Android 0

2.2/ Internal Navigation

Navigation of the different pages is managed by a stack, which makes it much easier to manage the history. The manipulation of the stack is very flexible. You can add one or more components, specify a set of parameters for the instantiated component or components, or even specify the type of transitions to make (using the graphic transitions defined in the initial stack view object).

A few years ago, that was pretty difficult to do on Android. But with Android Jetpack and its Navigation class *(https://www.youtube.com/watch?v=JFGq0asqSuA)*, now I'd say that it's just as easy and even clearer to navigate.

Internal Navigation > Qt 1 // Android 1

2.3/ Component Display

Every page is organized with traditional layouts (columns/rows, grids, or comparatively), which lets you position elements relative to each other to preserve the layout when the application runs on screens of various sizes. Also, integrating JavaScript into the QML lets you define bindings for the dimensions of various components using ternary operators based on various application properties (orientation, available resolution, etc.).

With Android, obviously, the same layouts are available to use. However, you can't include scripted language in the XML files. Nevertheless, dimensions (margin, padding, height, width, …) can be defined in dedicated files organized by target resolution. It is efficient but if you have too many specific cases, you end up with too much boilerplate code. You, on the other hand, only had to throw in a few ternary operators to manage the different sizes. I guess in the hands of some novices, it could come out messy.

Actually, since you brought up the resolution, in Android we indicate sizes in "density-independent pixels" and "scale-independent pixels." Does Qt have something to offer for that?

Yes, definitely. For managing high DPI screens, Qt versions 5.6 and later offer global attributes that are configured in the QCoreApplication, as well as environmental variables that allow you to activate and deactivate scaling, or configure it independently for specific screens. Also, implementation of this kind of scaling is going to depend on the characteristics of the various OS (retina screens on macOS, different levels of zoom on Windows, etc.) or else use the values configured by the developer if manual control is needed (for more restricted OS, for example, dedicated Linux boards).

Now that I'm thinking about all of it, if I am right all that JS definitely wasn't generated by Design Studio. And I'm getting suspicious because I see a lot of it in your dynamic screens. I'm not sure I believe you when you say that all that was automated. I think you went back over it yourself.

Component Display > Qt 1 // Android 0

3/ Discography Selection and Downloading

3.1/ Searching for files in the device filesystem

An existing, system-agnostic, QML component (FileDialog) allows you to integrate a native file explorer into the operating system to select songs. It's very, very easy.

With Android, one app can delegate an action (in this case, the folder selection) to any other app that can process it. That's how the "share" function in your photo software works, for example. It couldn't be more decoupled.

File searching > Qt 1 // Android 1

3.2/ Importing Files

For the back end module, Qt offers classes for browsing the directories and files (QDir, QFile, QFileInfo, etc. in the QtCore module) in the file system to find the music file paths that can be read by the app. This feature was actually integrated into the C++ standard in the C++17 version, as an `std::filesystem` module in its standard library.

Classes for browsing the file tree (java.nio.file) have been available in Java since at least version 1.2 (java.io.File*). It's not even a topic for discussion anymore.

Importing Files > Qt 1 // Android 1

4/ Presentation and Filtering of the Discography Representation Model

4.1/ Modeling and Filtering Songs

To manipulate data models, Qt offers a generic class, QAbstractListModel, that lets developers connect data getters with attributes (like artist, album, length, etc.). Besides the QAbstractListModel, you can also use a QSortFilterProxyModel. I assigned them particular attributes (one attribute for each) and a filter condition. This kind of model is integrated directly into the lists' QML components, allowing you to filter by masking which has a minimal cost in relation to performance.

I saw that in your code, it's like a wordier version of Linq (in C#). But whatever it is, I haven't seen anything like it in Java or Android. I eventually used Streams and old fashioned sorting and filtering.

It's an approach that's actually a lot like SQL views. It improves performance in terms of access to filtered models by avoiding costly operations on the initial list every time the interface wants to access it. As a bonus, multiple QSortFilterProxyModel can be chained for further filtering.

Modeling and Filtering Songs
> Qt 1 // Android 0

4.2/ Graphics

When it comes to UI graphics, the QML components make decoupling and reuse easier, allowing list displays to independently define the container (including its height, position, screen fill, etc.), the model (here, our QSortFilterProxyModel defined previously in the backend module, directly accessible by and compatible with the QML component) and the delegated component that displays each of the elements in the list.

As we already said, in Android, screens are defined by their structure and the static data in the XML files. They're just as decoupled and reusable as your QMLs. The dynamic components you get with JavaScript are managed by the element selector, which makes it possible to apply characteristics (border size, colors, fonts, background, etc.) to the components according to their state, the goal being to minimize programmatic manipulation.

Display > Qt 1 // Android 1

5/ Analysis of Media Content Metadata

The Qt Multimedia module offers classes (QMediaPlayer) dedicated to handling media files (in our case, audio files tagged with complementary information/metadata). These classes permit asynchronous reading and processing of metadata, which then serve as a discography model (as explained above).

The “android.media” package provides all you need to read media files and explore their metadata. But actually, I see that your app doesn’t display album covers, even though that’s metadata.

Chalk one up to you. Since the metadata that contains the album cover directly contains the raw data for a PNG image, displaying the images is a little more complicated. Because QML Image components only accept resource URIs, you have to set up an ImageProvider on the back end to be able to turn QML URIs into QImages from the metadata. That isn’t as easy as just providing the bitmap of the image directly to the component.

Analysis of Media Content Metadata > Qt 0.5 // Android 1

6/ Reading Audio Media

Reading audio is done by media classes from the Qt framework, so it’s entirely managed by the back end module (which interacts with the operating system’s audio drivers via the Qt framework).

This creates two important features:

- First of all, reading audio doesn’t depend on the state of the interface, which means that the music can be playing in the background while the user browses the rest of the app.

- Second, the functions can be decoupled, so the back end module can define the API that the interface accesses to display media information (title, album, total length, etc.) and the controls for the music that’s playing (play/pause, next/last track, volume control, etc.).

This way of doing is system agnostic, but it requires the OS to expose audio and video drivers that are supported by Qt.

As I said before, I like to use the “android.media” package, specifically with the MediaPlayer class that lays out everything needed to play music and control the position and volume. I did still have to encapsulate it to manage a song list. That said, there is a more complete MediaPlayer in JetPack (androidx.media2), but it’s very new (the first version came out in September 2019). So it’s a tie?

Yes, this time the two approaches are equally efficient.

Reading Audio Media > Qt 1 // Android 1

7/ Controls and Accessibility

Although I didn't use them in this demonstration, the Qt framework does offer options to facilitate app accessibility. For QML components, accessibility can be configured as a group of added properties (https://doc.qt.io/qt-5/qml-qtquick-accessible.html), which allows you to describe the role, name, purpose of the control component, and other properties that make it easier to fill in fields and navigate a page or form.

Accessibility is completely managed by Android. You just have to fill in a few extra fields, for example, image descriptions or a contextual help. Android Studio gives you a warning if there's anything missing, which is really nice to help ensure accessibility.

Controls and Accessibility > Qt 1 // Android 1

8/ Internationalization

For the i18n, everything's managed in Android Studio without interrupting the flow of development. You write the translation key, Alt + Enter, you write the caption in the main language, and you're back in your code. You end up with a minimal XML that just has to be duplicated for the translators. During replacement, the translation closest to the locale is used, with fallback to the language family, then the default language. Obviously, since Android Studio integrates the XML files, any renamed keys are global.

The generation process for internationalization files is more manual with Qt, I have to concede this point. Generation is done by an external tool that parses the code files, detecting where to find translations marker based on syntactic analysis (both in QML, with a JavaScript function, and in C++). Under these conditions, changing a key requires a new analysis execution on the entire code base.

However, in the lifecycle of an application, translations will not be handled solely by the developers. For the translators, Qt provides a dedicated tool, Qt Linguist, that makes it easy for them to enter translations and see which ones still need to be filled, without having to delve in the more technical XML file.

Internationalization > Qt 1 // Android 1

9/ Ease of Development

Qt offers Qt Creator, a simple, lightweight IDE designed for developing in C++ that also integrates a local help database for the many components in the Qt framework. The IDE also facilitates the deployment of the app on the target device during development. It offers deployment and debugging methods through a local network (deployment via SSH or SFTP, and remote execution of the software on the target device, debugging using the GNU C++ GDB).

Android Studio is based on IntelliJ IDEA, the older brother of the Jetbrains IDE suite. In my humble opinion, it's the most complete IDE family on the market. It has a lot of advantages over your text editor:

- Transparent on-target deployment of the application
- On-target debugging that's perfectly integrated with adb
- A real usage search method (not a dressed-up "search across project")
- Powerful static analysis that adds tons of additional warnings
- Naming suggestions
- So much contextual auto-fill using "Alt + Enter" that you practically want to marry it.

Yeah, it's a bit more developer-friendly on Android Studio, but it's not like there is a complete lack of tools to work with Qt and C++.

On the generic tools' side, the C/C++ development suite is bulletproof on Linux (GCC, G++, GDB and its server variant, Valgrind and its variants, or even the LLVM tool suite – ClangFormat, ClangTidy, ClangD, etc. –), and those are partially integrated in Qt Creator, along with Qt's own profiler for QML interfaces.
On the syntactic and semantic side, Qt Creator takes the same route as Android Studio, slowly but surely, via the development and integration of LSPs (Language Server Protocols) that let you skip the additional development of analysis tools.

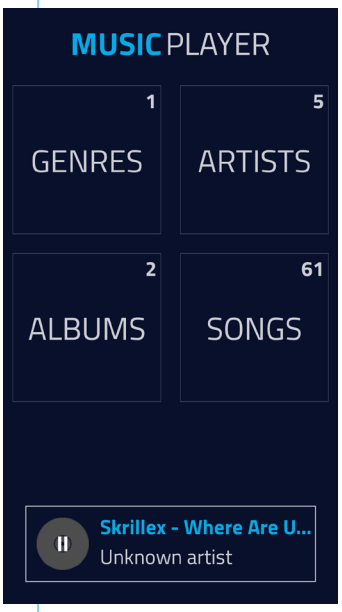
Additionally, the Qt application can be built as an Android .apk using Qt Creator. For that matter, it can target any embedded environment (e.g. iOS, RTOS).

Ease of Development > Qt 0 // Android 1

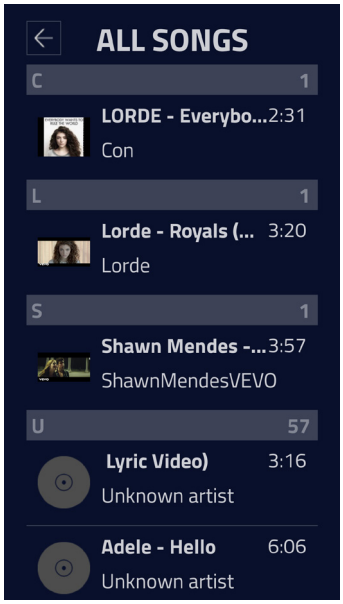
SOME SCREENSHOTS OF OUR MUSIC PLAYER APP ON A NXP i.MX 8 QUAD BOARD

Qt version

Home page



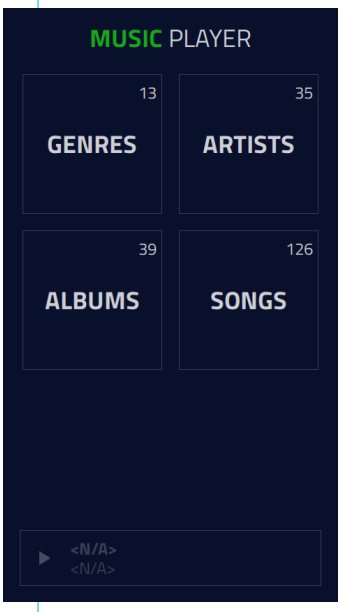
All songs page



Playing song page

Android version

Home page



All songs page

Playing song page

KEY TAKEAWAYS FROM OUR ANDROID AND Qt COMPARISON

After feverish debate on each subject, here are our final conclusions:

• Interface and UX for mobile applications

Qt has the advantage here. Both frameworks implement best practices (separation of the interface and business logic, and a responsive design tool), but the flexibility of QML (via its JavaScript integration) makes it easier to create dynamic interfaces while limiting wordiness in sources.

• Manipulation of media files

There's no clear winner here. Both frameworks offer all the tools that seem necessary for file access, locally or via remote flow, as well as the metadata in the media files.

• Comfort and Ease of Development

The Android development environment wins out over Qt and QtCreator, thanks to its more significant developer assistance and better abstraction of targeted deployment and debugging problems.

However, Qt offers Qt Design Studio, a tool that effortlessly connects graphic design and QML interface creation. There's no equivalent to this tool in the Android environment.

• Performance

The responsiveness of both applications' interfaces is good. In fact, they are very similar, except for the song list display, which functions progressively in Android and appears "instantly" in Qt. Without getting into the debate over the pure performance of C++ compared to Java, this does show the efficiency of Qt's filtered model system.

On this particular project, Qt wins, mainly because of its sorting and filtering of the data model. But the real answer is "it depends." There's no clear winner, it all depends on your priorities for your own project.

In this article, we've compared the functions and development "comfort" of the Android and Qt frameworks.

To get a more complete picture, it would be interesting to compare Qt and Android in terms of app performance for the user, particularly its fluidity (mp3 parsing, list display on different screens, etc.). That could even be the subject of another duel.

If you want to access our code, click here. *https://github.com/Witekio/qt-and-android-whitepaper-apps*

Part 2
Recap

	Qt	Android
ARCHITECTURAL BREAKDOWN	1	1
MOBILE STYLE INTERFACE		
From design to implementation	1	0
Internal navigation	1	1
Component display	1	0
DISCOGRAPHY SELECTION AND DOWNLOADING		
File searching	1	1
Importing files	1	1
PRESENTATION AND FILTERING		
Modeling and filtering songs	1	0
Display	1	1
ANALYSIS OF MEDIA CONTENT METADATA	0,5	1
READING AUDIO MEDIA	1	1
CONTROLS AND ACCESSIBILITY	1	1
INTERNATIONALIZATION	1	1
EASE OF DEVELOPMENT	0	1
TOTAL	**11,5**	**10**

18+ YEARS
SOFTWARE SYSTEM
EXPERTISE

5 OFFICES IN EUROPE
AND USA

100+ EMBEDDED
AND IOT SOFTWARE
ENGINEERS

200 INNOVATIVE
DEVICES PROJECTS/YEAR

Witekio

Paris • Lyon • Frankfurt • Bristol • Seattle