University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

glime (GLue pinetIME): an abstraction library over system APIs for operating systems of the PineTime watch

Andrei Boghean

January 16, 2025

Abstract

The aim of this dissertation project is to investigate and demonstrate a viable improvement to the platform of the PineTime smartwatch, specifically in it's currently confusing and difficult app instalation and usage process.

To achieve this, this project determined that the most appropriate improvement to be made lies in helping to remove the isolation of applications to specific PineTime operating systems. after review of literature demonstrating similar portability of applications between varying operating systems, this project settled on the use of an abstraction library over system API calls.

this approach was then shown to function in a restricted use-case, being usable to implement a simple watch application displaying useful string constants. evaluation also hints that the abstraction library is not detramental with respect to native development.

these results however can not be generalised on account of the small sample size of 7, but this dissertation project does still demonstrate that the approach is technically viable when applied to the PineTime with support for 2 operating systems InfiniTime and wasp-os.

Acknowledgements

I would like to acknowledge my supervisor, Yutian Tang, for agreeing to supervise this project and, despite having no obligation to, agreeing to entertain my naive idea to make the PineTime more usable, and for giving me the freedom to develop this into the project I was originally envisioning while helping me stay on track with the scope and offering valuable advice despite how confusing the project is.

I would also like to acknoledge my friends, family, and those arnound me for supporting me in this with endless advice and guidance. Lastly, further thanks are dedicated to Dr Pepper and other caffinated beverages for supporting me in this with the willpower and energy to complete this dissertation to the best I could have.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Andrei Boghean Date: 17 January 2025

Contents

1 Introduction

1.1 project specification

1.1.1 motivation

The PineTime is an affordable FOSS smart watch developed and sold as part of the PINE64 project. The PINE64 project is an initiative to deliver ARM and RISC-V devices with an emphasis on fostering a rich development community around these devices. Thanks to their developer-focused mission, the PineTime has been allowed to slowly evolve over time through various disjoint developer efforts and contributions, and by now has amassed a wide collection of unique operating systems and applications for use on the device.

With the independent development of operating systems disjoint from others on the PineTime platform, there naturally arises a degree of inconsistency between the design and implementation of operating systems on the PineTime. This unintentional disregard for interoperability is particularly evident in the underlying technologies used and is exacerbated by the lack of exhaustive documentation.

the tightly-knit community of the PineTime fosters software that is designed by and for others in the community, and leads to highly convenient software for a knowledgeable software developer in the community whom is aware of the software landscape for the PineTime and can navigate it properly, but can tend to lack accessibility for a more amateur individual interested in the PineTime and it's rich software efforts.

With so many different operating systems and development processes, this then means that trying to explore any other operating system or application besides the stock can then be a very daunting task that effectively gatekeeps developer efforts from the average user.

1.1.2 aims

This project should then strive to fulfil the following aims:

1. Provide a solution to simplify the process of installing and using others' apps on the PineTime smartwatch.
2. Design a system with adequate support for available software but remain adaptable enough for future OSes not currently on the PineTime landscape.
3. Achieve a minimum level of versatility to ensure the project is practically usable rather than being a proof of concept.
4. Strive to achieve these aims not just on one OS but rather across multiple host operating systems of the PineTime to avoid facilitating any form of vendor lock-in where any OS is significantly more featureful.

1.2 definition of terms

throughout this dissertation, there is a selection of terminology that has been adopted to aid explanation and conciseness. These terms are listed and explained here.

- adopting operating system : an existing operating system being retroactively modified with support for the abstraction library
- OS maintainer : generally refers to any entity with significant ongoing contributions to an operating system. This may refer to either an individual developer working on their passion project or a member of the wider development team behind an OS, e.g., a member of InfiniTimeOrg.
- independent developer : an unrelated user of a PineTime watch with development experience trying to develop an application using the abstraction library
- abstraction library : refers to the library produced as part of this project that defines an abstract interface for use across operating systems, designed to be exchanged with different implementations according to the OS.
- portable application : refers to an application created with the intent to be usable across multiple operating systems with minimal effort, i.e. an application that can be "ported" easily.
- externApp / external app / portable app : an application written solely with the use of the abstraction library that is being developed for execution for any operating system using the abstraction library but is "external" to a specific OS.
- wrapper app / wrapApp : a wrapper application introduced to an operating system by an OS maintainer who is supporting the library, usually introduced to "wrap" an external application.
- gateway : the code boundary between two layers in an operating system, usually used to describe specifically the boundary between native system code implementing an abstraction library, and C code also supporting the abstraction library

2 Background

The field this project relates to is well-established and has many papers that've similarly attempted to bridge some sort of gap between 2 or more closed-source development platforms. The most popular and immediate example of this would be between the open source and developer-friendly Android platform and the closed-source, overly-restrictive iOS platform.

2.1 solutions for a open ↔ closed transition (android ↔ iOS) on different hardware

2.1.1 context

Android - is a free, open source mobile operating system founded and maintained by Google. Due to its developer accessibility, its application marketplaces and other general developer projects are larger in number, at the expense of quality. By extension, the open source aspect has allowed the operating system to become the de facto on most non-Apple mobile devices, from the highest-end to the lowest.

iOS - is a closed-source operating system available only on Apple devices, mainly iPhones, with particularly stringent freedom provided to the user with regards to customisation and modification of the OS. It features very tight restrictions not just in the interest of security but also in an effort to enforce Apple's control over iPhone devices. On account of this control, iOS is not required to run on a wide array of devices and is, as a result, less versatile, which extends to its marketplaces that feature a smaller quantity of high-quality applications.

This abundance of adoption for Android but small pool of high-quality desirable applications on iOS can motivate a need for portability for applications between operating systems.

2.1.2 Android → iOS (using translation layer + abstraction library)

Puder (2010) aims to take 2 such devices that have fairly similar hardware but which differ greatly in their native application development models (Android-based Nexus One and IOS-based iPhone 3GS), and bridge that software gap between them.

The specific approach adopted here begins with "XMLVM"; a byte code level cross-compiler that enables the translation of an Android application to the iPhone, removing the need for two codebases to be maintained.

Their XMLVM framework designates a base OS, namely Android, for its wider device support and general freedom on the platform, and henceforth translates apps to the alternative OS, IOS, using a cross-compilation toolchain from Java source code to dex and then up to Objective C.

Naturally, this conversion of code can not be reasonably applied to system/hardware interactions since they operate on different hardware.

To address this, 2 solutions are considered:

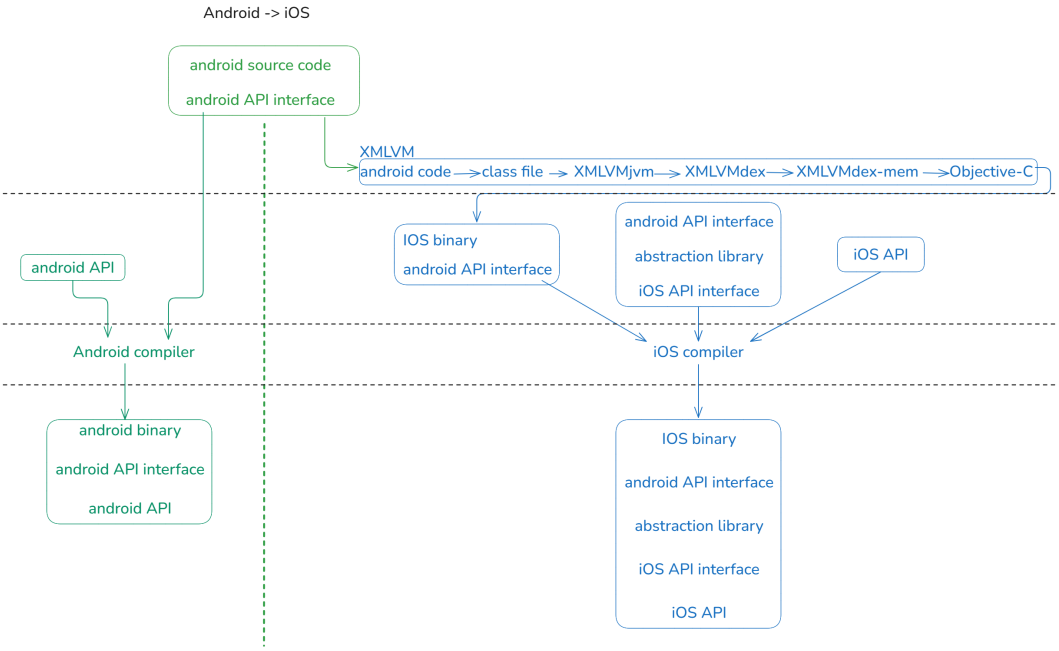
Firstly, modifying the application and directly translating Android-specific actions to their IOS equivalents. This approach ensures minimal additional overhead in the converted application but would be tedious to implement, potentially requiring a deep analysis of every individual application needing conversion in order to understand its specific and potentially nuanced usage of the API to learn what functionality would need preservation after translation.

The alternative approach considered is an API mapping, via a compatibility library, where the original Android application is unchanged, but the underlying API implementation is swapped out with an IOS-specific variant that exposes the same API but is implemented only using IOS-specific system calls.

In some sense, this compatibility layer acts as a virtualization layer, where the Android application is not aware that it is running on a 3rd-party platform. The Android compatibility layer only provides device-agnostic functionality such as layouting and falls to a "Common Layer" API for device-specific functions such as sensors. This Common Layer API serves to support various smartphones by necessitating that only the device-specific Common Layer API get re-implemented for new platforms.

In summary, this approach employs code translation for the majority of logic but accounts for OS-specific functionality via an abstraction layer and hardware-specific functionality with a "Common Layer API".

architecture diagram of this process:

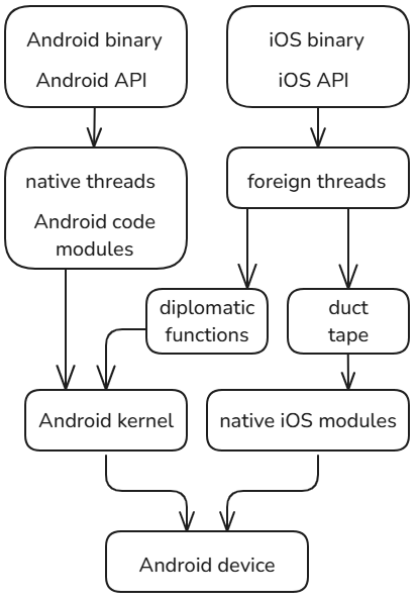


2.1.3 iOS → Android (using abstraction library + integrated iOS kernel code)

Andrus et al. (2014) shows a fairly similar but much more refined approach. It also uses a form of pre-processing to run apps, but in this case, rather than translating the application, the application is unencrypted and repackaged with the necessary files and then passed forward to the device, which again decompresses it for integration into the Android home screen.

Breakdown
Upon launching of the app, the application binary is executed natively using "personas", effectively threads with labels denoting whether the code it executes is foreign or native. This persona thread then carries out normal execution until a system call is encountered. System calls can then be trapped just the same as they are on native iOS since the modified Android kernel handles the call differently according to the classification of the thread.
Specifically with iOS, the kernel handles these calls with 2 techniques:
1. "duct tape", a compile-time code adaptation layer that permits the integration of unmodified foreign kernel code into the domestic kernel. This provides foreign iOS services natively within Android, allowing iOS apps to use them as if they are running on iOS.
2. "diplomatic functions", which allow the temporary switching of an iOS "persona" to Android, hence allowing iOS hardware interactions (such as graphics rendering) to be replaced with the equivalent native Android hardware interactions.

Architecture Diagram:



2.1.4 relevance

A significant amount of work in these examples serves just to deal with differences such as development environment (Java/Objective C), device specifications (Nexus One device / iPhone), and platform openness (permissive open source Android vs restrictive iOS). While these papers were a good starting point and established the usefulness of an abstraction library, the paradigm here is not properly matched with the dynamic between the very open PineTime OSes on the same hardware.

2.2 solutions for an open ← closed transition (Linux ← Windows) on the same hardware

2.2.1 context

Both these operating systems are designed to function on a wide variety of hardware, but the differences in their adoption and usage justify a conversion between Windows applications to Linux.

Windows Windows is a closed-source operating system developed by Microsoft. It has been well-established in the consumer PC industry for a significant amount of time and has been allowed to take over with relatively little competition from no one but Apple, which does not have sufficiently affordable devices to achieve the numbers necessary to challenge Microsoft. McKenzie and Shughart (1998)

Linux Linux is a free and open source Computer operating system that is particularly adaptable in its ability to run on most hardware including mobile phones, watches, laptops, and desktop computers. This has led to widespread adoption of Linux for most non-consumer applications, eventually encroaching into the consumer space. as Linux slowly gained popularity as a desktop operating system, users committing to a full replacement of windows found themselves still

occasionally in need of windows applications as a symptom of how widespread Windows has become, motivating a need for portability of Windows applications to Linux. Orr (2003)

2.2.2 Linux<-Windows transition (using translation layer)

Amstadt and Johnson (1994) shows a product that aims to allow Windows applications to run under Linux. It's more relevant on account of the fact that there is less concern for hardware differences (since both OSes are usable on the same machine), instead only needing to remedy the core problem of differing system APIs; however, it still needs to deal with Windows's restrictive design.

WINE uses no code translation such as was shown in Puder (2010) and no abstraction library for system calls. Instead, translation is used for system calls only, and something similar to Andrus et al. (2014)'s approach is used to integrate existing Windows services into the Linux host.

This method is beneficial because of its reuse of existing implementations, possible thanks to the high degree of similarity between Windows and Linux in their nature as operating systems, but it does, however, come with the drawback that the foreign program being executed is subject to the performance quirks of the host OS.

This can be beneficial in some cases (the Windows filesystem can be effectively augmented with multithreading when it's running under wine) and detrimental in others.

Additionally, since the sole device involved is the host PC running Linux, the solution is executed entirely on the host device in question.
This is only really possible since the host platform is a PC fully capable of the live translation demanded by the wine approach and would likely not work on a mobile platform such as mobile or smartwatch.

2.3 solutions for an open ↔ open (MantisOS ↔ FreeRTOS) transition on the same hardware

2.3.1 uni-directional transition (using an abstraction library)

Oliver et al. (2010) is a paper focused on a more unknown application of OS api translation and hence requires some introduction. The primary devices involved are members of Wireless Sensor Networks (WSNs), which are generally underpowered on account of the hardware being a mere embedded system.

These devices are extremely open and easy to support, thus, a lot of these different devices accrue their own selection of OSes, which then means there appears a very disjointed set of hardware APIs you may pick from even for the same device.

This then means that development effort for any hardware interaction modules can't be easily reused for different applications.

The paper attempts to resolve this by unifying the available OS APIs, something it achieves with the simple idea of an "Operating System Abstraction Layer (OSAL)". In particular, the OSAL works by defining a subset of OS primitives which adequately fulfil the most common needs of a potential application developer, while remaining simple enough to be supportable by the varying underlying OS APIs. The OSAL is used exclusively by the running application, and any "system primitives" or other API function calls are then translated by the layer into native equivalents.

2.4 current process of app installation

This project asks for a "solution to simplify the process of installing apps on the PineTime smart watch". Before any improvements can be made, some context must be established regarding what the currently existing process is like.

2.4.1 deciding an OS.

naturally, the workflow for installing an additional app to your PineTime watch can differ according to which operating system you're using.

InfiniTime - InfiniTime is the stock operating system for the PineTime and is likely what most users will be using. It is written in C++ and builds upon a freeRTOS kernel, using the lvgl graphics library for most rendering.

Wasp-OS - Wasp-OS is the second most popular OS available for the PineTime. It features drastically worse performance and battery time but is significantly more user-friendly on account of it being built in MicroPython.

other OSes - Other OSes available for the PineTime include pinetime-rust-mynewt, pinetime-zephyr, and PineTime-apps (among others). However, these alternative options are mainly passion projects. They do not appear to be in widespread use by the userbase (yet?), nor do they support popularity since most are lacking documentation and user-friendly instructions.

2.4.2 finding a useful App.

If a PineTime user may ever wish to install another app to their watch, they must first decide what app in particular they're installing. In the current state, there is not really any go-to online "market" the user may browse. Instead, they are limited to what they may find in the online community spread across GitHub, Discord, IRC, Matrix, Telegram, the Forum, and Reddit.

Luckily, these communities are very well-established and contain rich amounts of content, granted the user is willing to search through them.

InfiniTime in particular has a pseudo-market of applications in the form of pull requests to the OS' repository tagged with the "new app" label.

Wasp-OS on the other hand, has a similar label in their pull requests, but the community has not been submitting apps as PRs to Wasp-OS, so this list is useless in this regard. There does, however, still exist user-made apps that are available online, but they are much less visible than apps are on InfiniTime and must be "hunted" for.

other OSes however, do not face this problem since they have not been significantly forked and extended by the community, and most of the interesting work is still by the original developer and is therefore easily available in the host repository.

2.4.3 app integration

applications are typically tied to a particular repository, where the modified fork of the original OS is solely tailored for a single application the external developer has decided to create. Any non-beginner user is likely to have their own existing clone of their watch's OS, which they wish to extend with someone else's application, instead of having to entirely discard their existing download (presumably containing some other application the user has used) in favour of the repository with the other app. In this case, that then means the user must investigate the desired application and understand how to replicate its changes within their own clone of the OS.

2.4.4 current PineTime OS download and build process

After finding a new application, the next step for the user is to download and build the operating system containing the new application.

InfiniTime on InfiniTime, this first requires cloning the operating system, setting up dependencies (ARM-GCC cross-compiler, NRF52 SDK, the relevant python modules required, CMake, and the lv_font_conv node module) *build and program* (n.d.).
Subsequently, the user (presumed to have existing knowledge of Linux, cmake, and make) must properly configure the system with the correct dependencies in the correct locations, the correct compilation flags, etc. and then build.

Wasp-OS on Wasp-OS, the process is, in fact, very similar. First, the OS must be cloned and then dependencies established (arm Cortex-M4 toolchain, python, the relevant python packages, and various system packages). *Building wasp-os from source* (n.d.)
Compared to InfiniTime, Wasp-OS features a slightly simpler build system that's entirely made with make. This makes the build process easier to learn and slightly easier to re-build. This doesn't entirely remove the complications, however, since some degree of setup and make knowledge is still required by the user.

other OSes On other OSes, the build process is generally similar. More complex OSes feature a cmake system, and simpler ones may directly use make.

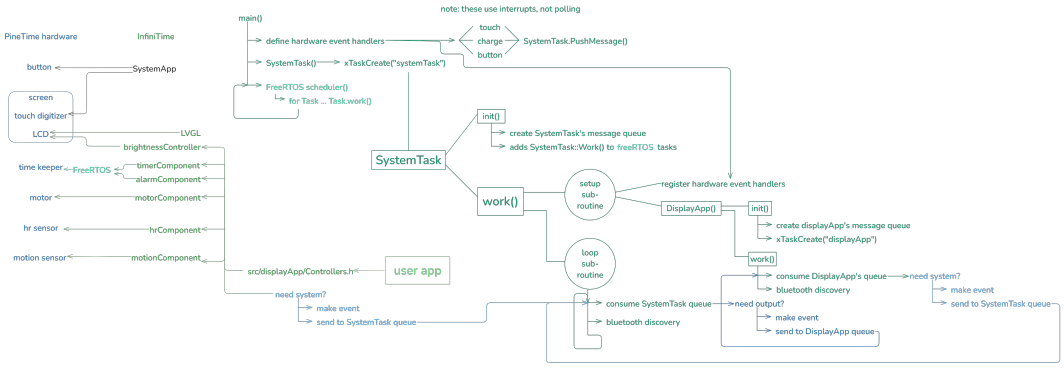
2.4.5 watch flashing workflow?

After having decided on an app and built the operating system now containing it, all that remains is to flash it to a PineTime unit.

This process is relatively simple and is the same regardless of which operating system has been chosen. First, the user must decide on the particular flashing platform (Android, iPhone, Windows, Linux) and then select a suitable DFU flashing application available. There is adequate documentation not only on the official PineTime wiki *PineTime Companion Apps* (n.d.), but also across repositories of different PineTime OSes which one can reference.

After this decision, the process can be as simple as loading the OS zip file to your device, putting your watch into flash mode (if its current OS has one), and flashing the DFU image.

2.5 InfiniTime architecture diagram



2.6 InfiniTime architecture breakdown

2.6.1 FreeRTOS

The general underlying architecture InfiniTime relies heavily on FreeRTOS, particularly its built in scheduler and timing.

Specifically, InfiniTime relies on the scheduler to organise any and all "Tasks" and "Timers" running on the device. These are FreeRTOS constructs which represent either a code module scheduled to run on the device, or a module scheduled to run within a regular timing interval. Excluding the kernel's setup at the beginning, these constructs are used for all execution on the device, including the stopwatch or heart rate applications serviced by Timers, its core services (explained subsequently) provided by Tasks.

2.6.2 2 main InfiniTime Tasks

The PineTime system operates using 2 fundamental Tasks: the System Task and the Display Task (also referred to as DisplayApp).

Upon execution of the OS's main entry point, the kernel configures and prepares the hardware for usage and then instantiates the first task, SystemTask. After this, this system calls on the FreeRTOS scheduler, passing the execution thread to it.

The scheduler then schedules the only task it has at this point, SystemTask, which goes on to execute its Work() method. The Work() method is comprised of 2 main subroutines of sorts; there's the initial setup logic which initalises hardware and prepares necessary data structures, including the set-up and registration of DisplayApp() as a task.

SystemTask setup
At the beginning of SystemTask::Work(), the PineTime's hardware, drivers, and management objects are set up and prepared for usage. This includes registering handlers for interrupts, button presses, touch screen events, etc., which add an appropriate message to SystemTask's queue as necessary whenever they execute. After the setup, an infinite loop is entered that fulfils messages in the SystemTask queue and periodically handles Bluetooth low energy discovery.

DisplayApp setup
Similar to SystemTask, DisplayApp also has its own message queue. This queue is added to by either the system task or user applications that are requesting some sort of job (e.g. turning off screen sleep).

summary - effectively, SystemTask handles how InfiniTime interacts with its hardware, and DisplayApp handles *when* and *for what* these interactions happen.

2.6.3 App interactions and DisplayApp architecture and components

If the system task needs Display-like services, such as outputting a new notification to the user, it pushes a message to DisplayApp accordingly.

To carry out any system interaction, an application needs to access kernel-like functionality, which it can do by similarly passing an appropriate message communicating its intent to SystemApp.

2.6.4 components

InfiniTime features "components" that provide particular logic to an application. Components are more privileged than user apps and are typically reserved for system interactions such as hardware sensor readings. Components are initialized a single time, together with DisplayApp,

and are reused across any application. Applications have free access to any and all components and use these instead of ever trying to handle system interactions themselves directly.

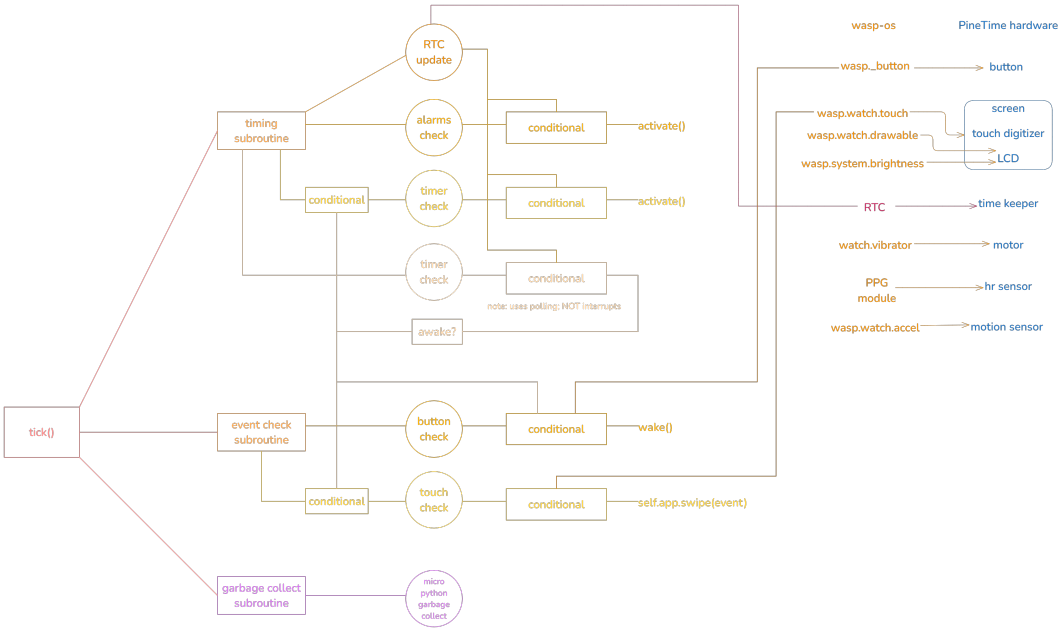
There is no apparent barrier forbidding the use of System-like functions from a regular app, but it's more beneficial to create reusable components.

input/output boundaries As previously mentioned, DisplayApp dictates the flow of execution most, if not all, of the time and does so according to what actions are indicated by user interaction events. SystemTask itself handles whether a user interaction event exists in the first place, and then relays this to DisplayApp for further decision making by adding an appropriate event to DisplayApp's queue. For example, when a touch event happens, SystemTask detects it, gives DisplayApp a message, DisplayApp then asks the Touchhandler component for further information, and then DisplayApp decides what type of action (swipe left, swipe right, ..., touch) should be triggered accordingly.

Note that there is almost a distinction between kernel space and user space. A boundary wherein Input/Output is permitted and effectively SystemTask serves the Input, with DisplayApp serving the Output.

hiding persistence Since applications are freed from memory upon closing of an app, they can not maintain any persistent memory on their own. To deal with this, applications which need it can either use an existing component to provide needed state (e.g. a timer that continues running even after the watch enters sleep) or create their own component that provides the required state.

2.7 Wasp-OS architecture diagram



2.8 Wasp-OS architecture breakdown

2.8.1 MicroPython

The majority of the complex OS-like functionality in Wasp-OS is provided by the MicroPython runtime. Similar to InfiniTime, Wasp-OS relies on MicroPython as its underlying runtime to

provide memory management, code execution, process scheduling, and other kernel-like services to the Wasp-OS main tick.

MicroPython is a more restricted but optimised version of the standard Python language that is purpose-built for running on microcontrollers and other embedded systems.

2.8.2 main tick

The system's main "loop" is defined entirely within a single _tick() method. This tick function is the only module ever scheduled with MicroPython's scheduler. It serves to provide all synchronicity for the device, handle button, touch, and charging events, and manage the sleep state.

It starts by triggering an update to its real-time counter object, which simply tracks the watch's time.

It uses this to then carry out all of its time-related checks, including checking the earliest alarm for whether it's time to activate and comparing time since the last tick() to determine if any timer requests by the current application should be triggered.

It then performs manual checks for the presence of any button or touch events, actioning on them if present.

Lastly, the routine compares watch uptime with the sleep timer and conditionally enters sleep mode, before finishing up with a call prompting MicroPython's garbage collector to run.

These steps are conditional to when the watch is awake. If it is not, the tick logic is limited to the alarm check and an additional check for a button press or a change in the charging state, either of which trigger the watch to wake.

2.8.3 input/output

Wasp-OS itself only handles UI rendering, other input/output, the drivers thereof, and the application logic tying everything together.

Its drawing logic is entirely custom. Wasp-OS's source code contains different modules according to the target device, which is swapped out during the build process. this means that the correct drivers can be used for different devices, but also allows the exchange of the micropython system with a regular python system for execution on a typical computer, useful for development.

Going back to drawing - Wasp-OS does not use any graphics libraries for rendering, instead opting to provide all drawing logic itself. This then means it's drawing logic is relatively unrefined and basic, putting a lot of effort on the application developer's side for achieving any complex visuals. However, this does not mean the drawing logic is inadequate; in fact it's very versatile and, where it lacks, can be easily worked around thanks to the freedom available.

In addition, the Wasp-OS libraries for hardware interaction, such as the PPG or the brightness controller, try to be as simple and easy to use as possible. In most cases, it's as simple as setting a variable or calling a single function with 1 or 2 arguments.

2.8.4 code boundaries

Further to this, Wasp-OS, in fact, does not establish any "boundaries" in logic. It's entirely reasonable and within design guidance to carry out even direct hardware interactions within an application's code, giving a lot of power to the application developer.

This then allows applications to be more centralised within their own file since they can achieve everything they need to without moving to other parts of the operating system.

3 Analysis/Requirements

3.1 OS API abstraction

The main objective of this dissertation is to devise some method by which any user with some particular preference of OS could program an application that can run on their own preferred OS, but would not be exclusive to it and instead could be adapted for other operating systems with relative ease.

Since this problem is focused on establishing some form of interoperability between multiple distinct independently created operating systems, it means that requirements are particularly limited by implementation details. Analysis must therefore begin with an investigation into the systems this project will have to work with, specifically into the architecture and application development processes.

3.1.1 analysis

initial investigations then started with determining how to externally insert code into an arbitrary project.

As explained in the introduction, the general build process for PineTime OSes uses a make build system, where any source first must 1. be included in the project's files somewhere, 2. must be communicated to the build system that the file exists, and 3. must in fact be used by the project.

After a new file (an app) is successfully added to the project, it must then go through the build process specific to the language. Generally, this means conversion from code to assembly, and then linking together into a binary.

insert binary
To provide the most familiar process, ideally, the user would be the one to build the program themselves into a final binary, which this project would then insert into the OS. This is why, first, a method for inserting binary directly was investigated.

The premise was to compile the binary as is standard with the gcc arm compiler, and then link it directly into the build process for any system. This would then have the benefit of requiring a single compilation step, which would generate a binary theoretically linkable to any other operating system provided the OS is modified to expect the binary.

A proof of concept was created which successfully re-created the last steps of the InfiniTime build process, with modifications to include a pre-compiled C binary. Including this into wasp-os, on the other hand, was too complex. The most sensible way to include a pre-built binary in a micropython project is with an mpy library as eventually explained in 5.2.2, however, the build system of wasp-os was not built to expect to ever deal with mpy libraries containing pre-compiled binary. This eventually led to issues in "freezing" the code from the binary mpy file, ultimately having no obvious easy solution. Furthermore, even if this method may have allowed for a very portable and convenient app representation (single binary), it would go against one of this project's aims of supporting already available software, since a deep knowledge of binary code execution would likely be necessary, something potentially doable by the core OS developers but certainly out of scope for the experience of the average maintainer or external developer.

integrate code
with direct binary inclusion being out of scope, we may move one level further up and consider inserting the user's code into the existing build system.

One such approach that may attempt to achieve this would be one that uses purpose-built code translation such as the system discussed by Puder (2010). Such a system would require the user's source code for an application, after which it would then translate its source code to the language of another system for inclusion. This system would then also have to deal with translating system API calls, but this is a separate point of consideration.

Such a system may work in a restricted use case between Objective-C and Java as was successfully demonstrated and would likely also work in our restricted use case considering just InfiniTime and wasp-os, however, it would be incredibly hard to extend for additional operating systems. Since the system intends to support conversion between any supported OS to any other, that then means it must support 2^n mappings with respect to the amount of supported languages i.e. for c++, micropython, rust, that would mean c++->micropython, c++->rust, micropython->rust, and also the reverse conversions. This is a violation of one of the project's aims to allow the *easy* extension of support to additional operating systems, not to mention that all of this translation would still be on top of the existing work necessary to handle API mappings.

insert code
The last remaining option is to insert the user's application code directly into the underlying application. This option is similar to the "code integration" from earlier, but differs in that it restricts the user's application language to a single one. this then removes the problem of explosive translation, but still means there is a new translation required for every particular target OS language the project supports.

Luckily, however, this can be dealt with by exploiting the nature of the target device. It is an embedded device, and the industry of embedded devices has slowly been drawn to implementing in C for it's low level compatibility. As a symptom of this, most other embed-friendly programming environments such as C++, micropython, rust, etc. are all particularly friendly with C and provide significant support for running C submodules as part of the final platform with a significant degree of interoperability.

If the chosen language for the developed application is then affixed to C, this allows the project to leverage C's interoperability to integrate it into multiple other platforms easily.

3.1.2 requirements

Requirement analysis can begin with restricting the scope of this project to just the 2 main operating systems of the PineTime - InfiniTime and wasp-os. This is because they're the only mainstream ones and are currently the only operating systems on the watch subject to frequent migrations of the user-base *SleepTk InfiniTime porting bounty* (n.d.). The requirements will, therefore, require the MVP to be fully demonstrable for all 3 MVP applications, whether running on either PineTime or wasp-os.

Puder (2010) states that only 15% of system APIs needed to be translated before most general applications became runnable with their solution, which is also backed up by Dunford et al. (2014). Hence, for this project, a restricted set of applications was decided on for support, with the understanding that working versions of these applications would extend to most other applications as well.

Considering the fundamental use cases for the PineTime watch and balancing these to aim to cover a wide variety of system API usages, it was decided to target 3 main applications: 1. a flashlight app making use of screen brightness, colours, text, and swipe/tap events 2. a drawing app to demonstrate custom rendering techniques and granular tap events with x/y positioning, and 3. a heartrate application making use of system timers and heartrate sensor interaction.

Further to this, it was eventually decided to additionally include a 4th application - a background timer. The reason for this was because of an implementation detail within InfiniTime's architecture, which meant timers that run in the background are a special case and thus it was deemed important to explicitly ensure they function.

The API calls necessary for the 3 initial apps were considered part of the minimum viable product, and those of the 4th app were a stretch goal considered nice to have.

3.2 other room for improvement

in addition to the project's primary goal of a general cross-os solution, there is other room for improvements that can be made throughout the whole of the app installation pipeline.

3.2.1 potential improvements in app discovery

The immediate idea for this step is an over-arching storefront for the discovery of user applications, potentially one that may make applications not only more discoverable but could also collate different applications from different operating systems.

However, creating this would at best provide users with a platform they can easily discover applications for their current OS. Users would still be restricted to applications specifically developed for their OS, and this hypothetical platform could not really improve the build and installation process.

Hence, this is excluded from the requirements and scope of this project.

3.2.2 potential improvements in the OS build process

a suite of OS-specific scripts could be created for each operating system, however this would mean the project must provide a distinct developer effort for every operating system chosen, and would also be a flawed effort because the work created could likely not be extended to further operating systems, unless an extreme development effort creates a universally adaptable system, which would be unreasonable.

Assistance scripts for building, however, are without doubt going to be a part of this project not just for convenience of development, but also to assist in the evaluation process which is carried out by unexperienced subjects.

Therefore, as part of the optional requirements, this project may produce convenience scripts for setting up and building both InfiniTime and wasp-os, and also setting up and integrating this project's modifications within the OSes.

3.2.3 potential improvements for integrating existing apps

as already established, the main focus of this project will be in creating some form of conceptual barrier between the underlying operating system and any user application, with the aim of making user applications more portable and more supportive of cross-platform behaviour.

4 Design

4.1 abstraction API architecture

The overall design settled on is conceptually simple and follows this structure:

1. There is an openly advertised API interface
2. A user developer creates an application which implements their app solely making use of this API interface.
3. There is a corresponding implementation to this API interface on a target OS.
    - The specific implementation differs across OSes, but the interface stays the same.
    - This allows portability of the application. Since it does not use any OS-specific details but rather only the OS-agnostic API interface, the implementation can simply be swapped out with another that is compatible with a different OS.
4. the application is combined with the interface, the corresponding OS-specific implementation, and the OS itself.

The hosting operating system naturally must be modified to expect this. generally, this means creating a ghost application within the OS that can be launched by the watch user, which will then delegate execution to the portable application after the ghost application carries out the required setup/preparation.

4.1.1 inclusion of external applications

The particular method for how an application should integrate into an operating system is a detail highly dependent on the platforms involved in the problem specification, in this case, the very open InfiniTime and Wasp-OS operating systems, that are both designed to run on the same hardware. as a result, the specifics of the most reasonable way to integrate an external application has already been covered in the problem analysis (3.1.1), supported by research from the background (2).

4.1.2 system API handling

However, it's not yet been discussed how to handle API calls for system functions across operating systems. what approach is used here does not threaten the project scope, so it's reduced to a design decision rather than a component of the requirements.

In the background literature, solutions generally either 1. convert API calls from one OS to another Amstadt and Johnson (1994) 2. include components of the other system WITHIN the host OS, essentially bringing the foreign API call into the host system Andrus et al. (2014) or 3. convert API calls from an abstract API to the specific underlying system Puder (2010) and Oliver et al. (2010)

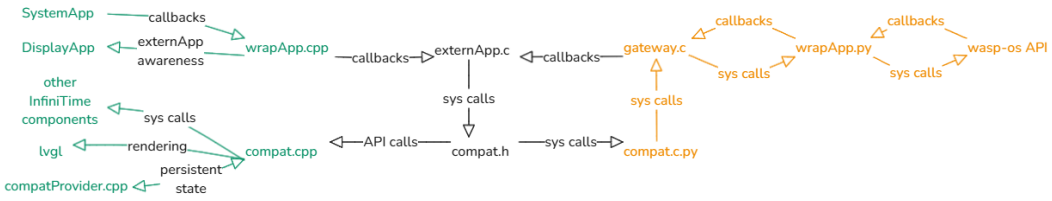
API call conversion
Similar to why code translation was discarded as a viable approach for application integration, using API call conversion will also require translations to be set up between every possible OS supported. What's worse, since each OS has it's own unique API, the effect would be even more pronounced than with language translation, where if the OS being considered is already written in a language the project hypothetically already supported for a different OS, a distinct translation solution would not be necessary because it already exists.

component inclusion
The modification of a host OS to include system components of another would provide the most familiarity since the foreign component isn't emulated or translated but rather literally included within the host. This would, however, still be problematic when scaling with multiple OSes because it would likely quickly approach the limits of the storage capacity of the small embedded device this project is targeting.

abstract API
This option comprises the designing and implementation of an OS API from the ground up, which provides the common services usually found on PineTime OSes. When another OS is added to the list of supported platforms, the process would be as simple as creating an implementation to the abstraction library that uses native system calls. This implementation process should hypothetically be simple not only since the OS developer is thoroughly familiar with their own OS but also since different OSes written for the same hardware are likely to end up with fairly similar OS functions despite their differences in development, thanks to the hardware platform being the same. What's more, this abstraction API definition shouldn't need to be fully exhaustive since it's been found that 15% of the host system's API is already adequate for a majority of usable applications Puder (2010). Lastly, with this approach, scaling with more operating systems should not need the project to be further modified with any translation methods or external components, since the abstraction library does not use any OS specific features, and "adding support" to an OS only requires modification of the OS itself.

system API handling summary
As a result of the inefficient scaling that API call conversion or external component inclusion demonstrates as you support more operating systems, together with the developer independence that an abstraction API provides (once developed, further support for an OS should not require modification on the part of the abstraction library), an abstraction library was chosen as the best approach for this case.

4.1.3 overall diagram



4.2 abstraction API interface

The abstraction API interface itself, the specification which operating systems implement and applications use for OS interraction, had to be designed carefully to strike a balance between simplicity of use and versatility to permit complex applications.

The API had a few notable design decisions to act on. Design decisions that most could be considered implementation details, but that pose distinct requirements when specifically applied

to the platform of wearable watches such as dealing with touch/swipe events. Since open source developers are presumably not highly invested and do not have the resources for bespoke systems. Most independently designed operating systems are still likely to use existing well-established designs/approaches where available.

4.2.1 paradigm for rendering of elements

One of the most immediate design decisions that needs to be made when developing an operating system is how to orchestrate interactions with your operating system's UI toolkit. Naturally, a given system will have some method of drawing to the screen. This will, in all cases, contain the primitive of a single pixel. Some systems (and, in fact, most) use functions to codify common drawings such as squares, circles, labels, buttons, etc. Further to this, some systems may even bind the properties of these created elements to programming constructs e.g. uses C++ classes with attributes for item positioning, colour, text, etc.

not every operating system provides a full GUI library such as this. some provide widgets where necessary (buttons and selectors) and do not bother in other cases such as with basic squares and labels. to properly handle this, the abstraction library needs to choose a level of robustness which it provides. this level needs to find a balance between zero rendering assistance and a full widget toolkit.

object-based GUI The more complex OS, InfiniTime, uses the lvgl graphics library for its rendering. This library provides an exhaustive UI toolkit comprised of widgets and full bindings for their properties, meaning that graphical elements such as text, buttons, symbols, etc. only need to be created once. The background system handles continuous re-rendering, layering, and object mutation.

As far as the user is concerned, they only need to pick and create the widget they desire. they start with a simple draw call that creates a blank object, and then modify the widget's parameters to achieve their desired look. they need to make no further changes unless they have a need e.g. if they wish to re-orient the item they can simply do so by mutating its position.

single-use? Wasp-OS, on the other hand only provides graphical assistance up to showing primitive graphical elements on the screen.

For the user, this means they can use the builtins for placing labels, squares, etc. but if they wish to adjust their properties they need to once again execute the draw call with changed parameters.

this draw call will then of course draw the new element with it's new parameters, but the old element still persists in the video buffer. the user needs to either accept that it is there, or manually clear the screen before they draw in order to clear the outdated render.

this process becomes even more complicated when combining elements. if changing a single element, the user is not able to selectively clear the part of the buffer relating to that object. they must clear the whole screen, and then redraw everything, including any other elements that were also on the screen.

the object management that is provided by the graphics library in InfiniTime is not present, so it is all offloaded to the user developing the app.

HOWEVER, such objects are actually provided by Wasp-OS when the object's implementation by definition requires state. this mainly applies to components such as buttons and selectors. in this case, the OS returns an object but still does not bind any customisation to it. yes, the position and values can be changed, but visual properties can not.

design decision ultimately, a tradeoff was made to have only stateful or complex objects be bound to objects. This means all graphical elements still have to be manually cleared and re-rendered during adjustment, but complex widgets that can't avoid an object representation (buttons, selectors, etc.) may provide render functions for convenience.

This was decided because passion project operating systems (the most common for an open source platform such as this) are unlikely to go through the trouble of integrating someone else's graphics library, since the point of such projects is for the developer to enjoy the processes of creating code themselves. By this logic, InfiniTime is presumed to be an outlier in its usage of fully controllable widgets, and this should not be reflected in the abstraction library.

this also makes it friendlier to potential OSes adopting the project, since this reduces the work when implementing graphical objects in the OS's variant of the abstraction library.

4.2.2 design pattern for interacting with alarm interrupts

On the topic of events, most, if not all, platforms use handlers for events. Some platforms, however, use static method overrides to define event handlers, whereas others use a function pointer.

handler override specifically, using a handler override starts with a class provided by the operating system that represents a user app. some of the methods defined in this class relate to handling events, so the application developer can then override these methods with their own logic for responding to events, when they create the subclass for their specific app.

handler pointer On the other hand, the operating system may simply ask for a pointer or a reference to any function with the required signature. To use this, the user puts their event handler logic in such a function and then passes the function to an event registration method.

design decision With handler overrides, it's slightly more inconvenient to register multiple handlers for an event because you can technically only register a single handler. You can however program this to delegate events to other functions accordingly.

Inversely, with handler pointers, it's much easier to register multiple handlers to the same event queue, but it's also easier to lose control of what is registering event handlers, since multiple sources may register events without it being obvious.

In the end, passing handler pointers was settled on because being able to register multiple listeners for events is more powerful but also more intuitive and therefore, likely the more common approach. Furthermore, adapting a handler-override OS to mimic handler-pointers is much more convoluted than adapting a handler-override OS to mimic handler-pointers, since a class data structure would need to be introduced, and the handler paradigm would need to be adapted to work in an object-oriented nature.

4.2.3 hardware interactions (specifically heartrate)

The hardware for the PineTime uses a hrs3300 heart rate sensor. For proper operation, this sensor should ideally be polled at 25Hz *HRS3300 Heart Rate Sensor* (n.d.)

In order to take a bpm measurement, anything executing on the pinetime must: 1. initialise the sensor. 2. Record measurements at a rate of 25Hz. 3. analyse the measurement trend to extrapolate the final BPM measurement.

While this process is fixed, different operating systems may split it between user and kernel space differently.

some may provide only drivers for the heartrate sensor, and require the user carry out the full process themselves. some may also provide sensor initialisation, and some may even record measurements for you.

it is reasonable to expect OS assistance in initialising the heartrate sensor, but not anything further. however, from investigations, OSes typically provide trend analysis and it is only measurement recording that is or isnt provided.

OS-handled measurements The InfiniTime operating system has in its implementation a "component" representing the heart rate monitor which, when activated, uses it's internal heart rate driver to initialise and handle heart rate measurement and processing. All the user application needs to do is start the service, and then they are free to poll for the current heart rate as they please.

user-handled measurements Wasp-OS on the other hand provides only the driver and processing. the user application is tasked with setting up a timer routine to poll the heart rate at a specified interval.

design decision While OS-handled measurements is more straightforward for the user, user-handled measurements allows more freedom and lets the user application access the full range of data measurements. this then means that a heart rate app can be more featurefull e.g. it can graph the PPG data.

furthermore, since OSes still provide functionality to analyse the measurement data even if they don't provide measurement, letting the user take measurements should not be much more complex.

However, there is an implementation detail in InfiniTime that does not permit this. The underlying measurement functionality is isolated to the kernel space and can not be directly used by a regular application.

since InfiniTime is a major PineTime OS, the abstraction library is required to support it and thus has to make this allowance here.

This is why the choice was made to implement OS-handled measurements rather than user-handled, despite user-handled being more appropriate.

while this does mean any OS that follows user-handled must implement OS-handled when they integrate support for this project, it is still the case that implementing measurements is conceptually trivial so should not be difficult for the majority of OSes that ever adopt this project.

Furthermore, this design could eventually be modified to allow the user to sidestep OS-handled with their own user-handled measurements.

5 Implementation

5.1 high-level overview

This project features 2 definitions of an "implementation": Firstly, there is the hypothetical implementation process that occurs long past this project is completed. It is the process of an independent OS maintainer attempting to add support for this project's abstraction library to an adopting OS.

5.1.1 steps

This implementation workflow specific to the OS maintainer is as follows:

1. Resolve how to insert arbitrary code into the operating system
The first step to retroactively modifying an operating system with support for this project's abstraction library is to determine how to insert arbitrary C code. This is necessary to allow the system to even understand the abstraction library, and as a process, it can vary in complexity according to the languages used within the host platform.

2. facilitate System -> externApp interaction.
in order to include an external application, this application must be understood by the operating system to be an application, however the portable application is not structured to be understood by InfiniTime, Wasp-OS, and much less any other operating system, so any adopting operating system needs modification for it to expect and understand portable applications. This is typically done by the use of a "wrapper" application to delegate execution.

3. facilitate externApp-induced System -> externApp interaction
In addition to the basic System → externApp interactions handled in step 2, there is also more complex communication, such as that which is induced by the externApp itself (as is the case when setting up any sort of callback).

Such mediation can not be handled by the core component of the abstraction library, since the core only mediates interactions externApp → System, hence responsibility falls here to mediate callbacks.

Typically, this is also achieved via the wrapApp.

Note that it would be possible to modify the host operating system to interact directly with the externApp, bypassing the need for a wrapApp entirely both in steps 2 and 3, but achieving this would potentially require significant rewrite of the host OS. not just to introduce the expectation for an external app, but also to modify how native applications are handled. They would potentially require porting to use the abstraction library instead of native system calls.

In order to remain minimally intrusive to the host operating system's code, this project's implementations of the abstraction library on InfiniTime and Wasp-OS choose to keep the wrapApp mediator, in interest of maintaining parity with the likely actions of a hypothetical independent developer that is retroactively modifying an OS they didn't develop to support portable applications, with the intent of being as minimally disruptive as possible.

4. Implement the abstraction library for this OS
finally, it remains to implement the abstraction library for the adopting operating system. This step is the most involved but does not need to be completed exhaustively.

By the nature of software, a majority of applications developed using a UI toolkit do not necessarily use the full power of the toolkit, and in fact, most functionality remains unused in an 80/20 split Dunford et al. (2014) Gittens and Godwin (2007). This can be taken advantage of by an implementer of the abstraction library to selectively support a subset of a full UI framework, for example, only containing features they deem important for their specific use case and expanding only as necessary.

For example, if an OS is purpose-built for heart rate monitoring, it may only implement features relevant to monitoring heart rate, meaning that the "20%" of a full UI framework that this particular OS may be using is only that which is specific to it. This would not be expected to cause troubles for end users because, by way of the 80/20 rule, only this specific subset is actually needed by the OS regardless, and other logic is out of scope or unneeded. This is also supported by Puder (2010)

5.1.2 implementation interweaving with design

Ideally, these above steps are completed for each OS being supported independently without consideration between them, however, the development process in this project was closely linked to the design process since ad hoc implementation was the most sensible way to ensure a design decision was possible.

In practice, this dissertation splits the simpler implementation definition laid out above into a "preparation" and an actual "implementation" stage.

The preparation stage still features the first 3 steps and essentially comprises the base proof of concept for this project. First, it was accomplished with InfiniTime and then Wasp-OS.

The implementation stage, now combined with design, is the phase that features the concrete definitions and implementations for the abstraction API, e.g. the notion of a "place_label" function, its InfiniTime implementation, and its Wasp-OS implementation.

5.2 "preparation" phase

5.2.1 InfiniTime preparation

inserting arbitrary code - Since the project settled on using C for the language of the portable application, inserting the abstraction library's files and portable application is trivial in the InfiniTime C++ system Stroustrup (2002). Specifically, this meant adding the abstraction library header and implementation in the root of the source and then adding the wrapper app much like any other, making sure to register these new files in the CMake configs. Eventually in the development cycle, this also meant adding the compatibility "component" - that is, the portion of the compatibility implementation that is designed as a component (because it is needed to provide component-exclusive functionality) and is therefore included in the build process together with other components rather than apps.

facilitate System -> externApp interaction - In its simplest form, this wrapper application consists of a wrapApp class that extends Screen, overriding only the constructor. Within this constructor, the abstract application is imported, and the extern main that it defines is then executed, completing the "handoff".

As the abstraction library develops, however, there is increasing complexity within the wrapApp. This is because, for most functionality, InfiniTime assumes the application does not need it and,

therefore, does not provide it (examples of this being touch events or components such as the battery or motor component). In its final form, the wrapApp not only overrides both versions of OnTouchEvent with dummy implementations but also provides logic to the abstraction library implementation that it can use to replace the dummy with actual touch event handling logic which the portable application wishes to run.

Further, wrapApp also obtains all InfiniTime components ("app controllers") and makes these available to the compatibility library implementation as well.

facilitate externApp-induced System -> externApp interaction - During initualisation of the abstraction library, the wrapApp passes a pointer of a pointer of a function to the abstraction library. this pointer can be used by the abstraction library implementation to pass arbitrary function pointers up to the wrapApp. the wrapApp can then use these pointers whenever it's delegating an event callback dow nto the external application.

these variables, by default, feature dummy functions with no logic. when the external application registers an event handler, the default method is discarded and overwritten.

this technique is used for all callback-like functionality in the abstraction library, including touch, swipe, timer, and background timer events.

5.2.2 Wasp-OS preparation

inserting arbitrary code to facilitate externApp -> System interactions Wasp-OS's main system is built on top of the micropython runtime. This runtime features 2 officially supported methods for extending it with additional logic *Extending MicroPython in C* (n.d.).
that is; 1. MicroPython external C modules, and 2. native machine code in .mpy files.

Of these two methods, the more beneficial one would be the integration of native machine code because it does not require code to be written in C but instead could theoretically accept any arbitrary language as long as it compiles to the appropriate architecture (in this case, ARMv7).

This means an external application could be independently compiled and built into a standalone module, and then be included into Wasp-OS's build system post-build of the application.

Upon creation of an mpy module containing native machine code, however, it was discovered that the Wasp-OS build system does not support the inclusion of such modules. Proceeding with this would then have presumably required modification to the system, which disagrees with aim 2 of this project.

That is because, in consideration of an independent programmer attempting to retroactively modify an existing OS they have little experience with, they similarly would prefer to avoid modifying the Wasp-OS build system which is understood to be difficult, so this project should prefer to avoid establishing such a time-consuming expectation that would otherwise increase the difficulty of adapting the project for other future OSes not currently on the PineTime.

The external C module approach, on the other hand, does not provide the same difficulty because external C modules are included within the micropython runtime itself, not Wasp-OS, and micropython has a much more robust and well-documented build system.

Further, to allow the flow of execution from the application to the Wasp-OS system (i.e. allow the abstraction library to use system calls), some connection needs to be established between the abstraction library implementation and the Wasp-OS system.

All interaction between C and Python must go through the micrpython API, but there were several possible ways this could've been achieved.

The most intuitive solution would be for the implementation library to use the micropython API to directly call system functions, however due to how the micropython API works, any

execution of a micropython function from C requires that the C function have a pointer to the python function. this is certainly doable, but requires an additional pointer for each function. this is not practical at scale.

Instead, a "delegate" method was implemented in the wrapApp to facilitate the flow of execution from C->Python. This delegate takes an argument indicating the action being requested by the abstraction library implementation, and an additional **args parameter passing an arbitrary amount of additional parameters necessary for the action.

Hence, the whole process generally looks like:

1. the externApp requests e.g. place_label with an x and y
2. the abstraction library implementation allocates a string literally containing "place_label", and additionally space for the x and y variables.
3. the abstraction library uses the pre-communicated pointer to the delegate function to call it, passing the 3 arguments.
4. the delegate matches the string to the desired operation, and carries it out on behalf of the abstraction library implementation.

Due to this system, most logic within the Wasp-OS implementation of the abstraction library in fact sits within wrapApp's delegate function, with most functions in the implementation of compat.h simply forwarding the function call to the delegate.

Ultimately, the externApp and compatibility library are made available on Wasp-OS as follows: First, a "gateway" micropython module is created, which itself includes the compat.h header. At compile-time of micropython, this header is linked with its OS-specific implementation, and gateway.c. This forms a micropython module "gateway", that any micropython applications running on Wasp-OS can subsequently include and execute functions from.

facilitate System -> externApp interaction The Wasp-OS wrapper application is slightly more contrived with respect to InfiniTime. To properly pass execution to the external application, the Wasp-OS wrapApp needs to route it through the gateway module, meaning any and all access of C code from Python must be planned out in advance because it must be added to the gateway module's interface at compile-time.

facilitate externApp-induced System -> externApp interaction There are some cases (event callbacks) where a C->Python action eventually triggers a Python->C action. In this situation, Python->C must be aware of the C function it must call for the callback, but the interface for the gateway module is defined statically and built during compile-time.

This means there is no direct way to advertise an arbitrary C function as one callable from Python. To work around this, the function pointer pointer technique previously demonstrated in InfiniTime is used to store functions within the gateway.

The gateway can then advertise a single function usable for any callback, which can then delegate execution to the appropriate externApp function stored within the gateway.

5.3 "implementation" phase

The design and implementation processes were closely linked since the design process needed to ensure that any design decisions were possible on both operating systems involved (and others).

This meant that the general process for considering the addition of a feature was

1. Look at existing applications on the OS which use the same feature
2. Do the same for the other OS
3. Create preliminary function interfaces that equally reflect how the feature works on both OSes

4. Attempt an implementation
5. Refine accordingly

This process was carried out on the 3 PineTime applications previously established as part of the MVP. Each application was investigated for it's use of the underlying system APIs, and then each system API feature was iteratively added to the abstraction library following the above process.

flashlight application:
The first of the three MVP applications to be converted was the flashlight application. From initial inspection, the InfiniTime and Wasp-OS flashlight apps use labels, swipe gestures, brightness control, and background colours.

In pursuing this implementation, the first function added to the library was show int. this function served an important purpose in development since it permitted the communication of basic changes and values in code, so it was kept in the library for future help during development if anyone may ever wish to add support to an OS. Second to this was place_label. This is a function that is, in theory, relatively simple in logic, but demonstrated the first small challenge in implementation. The approach used here would set the precedent for the rest of UI interaction with this project, so it needed to be a thought-out decision. Following the considerations laid out earlier in 4.2.1, it was decided to follow Wasp-OS's simpler manual re-draw paradigm.

This meant the implementation here must dumb down InfiniTime's (or rather LVGL's) complex mapping of UI elements to C++ objects. In order to achieve this, the implementation still makes use of InfiniTime's built-in UI toolkit but discards any objects produced by InfiniTime as representations of UI elements.

Further parity is achieved by adding a clear screen function to the library that paints the screen black on Wasp-OS and frees all existing LVGL objects from memory in InfiniTime, something made possible by LVGL's lv obj clean.

With this so far, the flashlight application's state alone can be communicated. For the state to be able to change (on or off), some user input is required - specifically swipe gestures. Both InfiniTime and Wasp-OS provide swipe gestures via a class method that the application must override with their own event-handling logic, similar even to the extent that tap and swipe gestures are separate methods. This means the gap can be bridged with a default event handler implementation that stores a pointer to a function. This function, by default, does nothing, but can be overridden using "register global eventListener" that modifies the function pointer variable and inserts the user's desired handler instead. This approach is suitable for both operating systems.

Lastly, in order for the flashlight state (whether it's on or off) to have any impact, screen brightness control and background colour must follow. Brightness can be implemented in InfiniTime by translating the compatibility library's Low/Med/Hi enum to the native one appropriately and passing this to the Set method of the brightness controller, made available thanks to the wrapApp. Similarly, on Wasp-OS, the enum is translated to an integer and then assigned to the wasp.system.brightness variable.

Colours, on the other hand are slightly more complicated. Graphical elements in InfiniTime all have their own default colours which get used, so if an application desires a particular colour, it must be overridden manually. In Wasp-OS, on the other hand, the application only has to specify the current foreground and background colours, and any subsequent rendering calls make their best effort to incorporate these colours into rendered items. To mirror this logic on the InfiniTime side, the implementation maintains an internal variable tracking the colours currently set by the user. Upon any creation of a graphical element, the implementation then has to manually ensure the elements are set to the colour dictated by the internal variable. This means that any existing rendering elements had to be updated to make use of colours, and any future graphical methods will need to properly support colour.

5.3.1 Paint application:

The second MVP application is a paint application that lets the user draw with taps or use swipes to change colour. Building on top of features achieved from support of the flashlight app, this app additionally requires "register global eventListener xy", and "draw rect".

To support the tap event variants which feature x/y positions, the existing logic for global events can be reused.

Draw rect on the other hand, is much more complex. While Wasp-OS features a general-purpose rectangle drawing function that can be used with minimal difficulty, InfiniTime's native drawing application used a grid of colours, effectively an in-memory image, for the stroke of the pen. For each tap, the application would render this image at the tap position. This is suitable enough for a paint app, but it does not scale well with the size of the rectangle since it needs to allocate the whole image in memory. The first workaround considered for this was the use of LVGL graphics objects that get assigned the currently selected colour. This method worked for a small amount of method calls, but when used in a paint application with tens or even hundreds of strokes, the memory fills up quickly and eventually crashes the watch. Instead, InfiniTime's native method was adapted to, instead of allocating the entire image buffer at once for bulk drawing, allocate a single pixel and then place this manually at every position. This method is much more memory efficient because it uses LVGL to flush to the display buffer without any graphics objects, but it becomes slow when drawing a large rectangle since, in the worst case, this method flushes the whole display buffer pixel by pixel.

5.3.2 heart rate application:

The last application part of the MVP was the heart rate application. For proper heart rate function, this application requires start read hr, stop read hr, and get hr bpm. It also needs a "register_timer_interrupt" for the application to be able to regularly measure heart rate data and provide a live update of the current heart rate.

Starting with the heart rate functions - on the InfiniTime side, this was as simple as calling Start(), Stop(), and HeartRate() on the heartRateController.

On the Wasp-OS side, this was more convoluted since data processing is not provided by Wasp-OS's PPG component. First, an instance of the wrapApp class had to be obtained, then the heart rate sensor is enabled, and last, the wrapApp's _hrdata variable is populated with a first measurement. Pinging hr data needed a simple _hrdata.preprocesss() call, and stopping hr data only needed the clearing of the _hrdata variable.

Despite the apparent simplicity, there was particular difficulty in this because, previously, the delegate had no access to an instance of wrapApp, so it itself had no state where it could store _hrdata. Any state such as colour info or handler pointers would either be embedded within the Wasp-OS system (i.e. wasp.watch.drawable._bgfg) or in the C side in the gateway, which has its own persistent state. Measurement data of course, could not go within wasp because wasp does not implement assistance with measurement processing, but it could've been reasonably placed within the gateway. Placing this in the gateway would be simple but would require conversion of the measurement logic to C and may defeat the purpose of having an abstraction library. What's more, it's almost certain that future features may similarly require some form of state, and setting the precedent to something so tedious may discourage exhaustive implementation of the spec if the Wasp-OS implementation is ever further developed.

For start_read_hr's heart rate polling routine and for the regular re-rendering of the current heart rate value, the abstraction library must provide timers to serve functionality at regular intervals.

As is the pattern, InfiniTime demonstrates the preferable implementation with support for multiple timers running on the system together. Wasp-OS does not have the same versatility, and needs to be augmented with logic to multiplex timers into one, mimicking InfiniTime.

Specifically, this was done by tracking all timers so far within the gateway application and requesting the Wasp-OS system for a tick duration corresponding to the greatest common divisor of all of the timers. This then allows the gateway to keep a running total of time each task has been waiting, and then execute a task whenever it's running total reaches it's requested task period.

5.3.3 stopwatch application:

With the heart rate app working, the project technically reaches its MVP state outlined in the requirements, but there are still developmental concerns with regard to InfiniTime's component system. Specifically concerning the lack of application state in InfiniTime, where applications are destroyed when they lose focus, such as during watch shutdown. This may pose a problem for apps such as alarm clock apps or sleep trackers that need long-running background timers to carry out their alarm check or motion data processing.

Hence why it was decided to additionally pursue the implementation of a stopwatch app, to investigate how the library implementation may work with an InfiniTime component.

The stopwatch app requires 3 additional features in the abstraction library: selector widgets, button widgets, and background timers.

Selector and button widgets could be relatively easily implemented with the techniques demonstrated so far, so they were not a focus of this development. Similarly, thanks to the simplicity of Wasp-OS's tick() routine, converting a timer into a long-running background timer is as simple as exchanging the timer() call with an alarm() call. In InfiniTime, long-running background timers require a dedicated component that can serve as the persistent state for the timer object. There is, in fact, already a timer component that is used by InfiniTime's built-in timer and alarm apps, but the convention on InfiniTime is to create one component to serve one application, so these components can not be simply reused. Instead, a dedicated component needs to be created and then linked to the abstraction library.

This dedicated component also needs to track the timers it has created because, otherwise, it does not necessarily know the source of a timer. Whenn faced with 2 distinct timer requests, the component can not know if a request is either from the timer creation call of an application, but over 2 separate instances of the app where it executed once, the watch turned off, and the app eventually executed again, or if the requests are for timers with distinct purposes, such as a heart rate app starting one timer for polling and one timer for showing the BPM within the same execution. For this reason, timer creation is additionally specified with a timer ID, allowing the component to return an existing timer from a previous application instance or create a brand-new timer if no timer with that ID has been historically requested.

Since this timer ID is only a detail necessary in InfiniTime (Wasp-OS does not necessitate components, so this management is not necessary), the timer ID is only used by the abstraction library and is never seen by the external application.

Ultimately, the timer flow of execution is as follows: the external application requests a timer. The abstraction library passes this to the compatProvider component, together with a timerID. The compat provider either creates this timer or returns an existing timer if there is one with that same ID. This timer is then passed back through to the chain, eventually giving the external application an abstract void* that represents the timer. This can be used to carry out other timer actions, such as stopping or checking remaining time.

6 Evaluation

6.1 plan

The overall plan for the evaluation is to subject experiments to 2 different variables: firstly, whether they're using InfiniTime or Wasp-OS, and secondly, whether they're carrying out the experiment while using the project or without it (the control group).

The evaluation is carried out as a between-subjects experiment.

Its format starts with a pre-experiment survey explaining the details of the evaluation and some preliminary questions, followed by the experiment itself and lastly, a post-experiment questionnaire.

6.1.1 pre-experiment survey

The questions of the pre-experiment survey aim to understand the following specific attributes of the subject:

1. their experience levels in linux, C, C++, and micropython. This should inform the expectation for how long the participant should take and should ideally have minimal influence on the time to develop, presuming the project is suitably usable for newcomers.

2. their prior experience developing on mobile platforms. This information should give an indication of how informed the participant's answers are in other sections of the form, notably where the user is asked whether they've experienced difficulties targeting multiple operating systems for any mobile platform.

3. their prior experience with smartwatches. this should indicate how well-informed the user's answer is to the subsequent question of whether the user has ever felt a distinct need for some niche smartwatch application. e.g. if the user is experienced and gives little answer, that indicates their experience with smartwatches is relatively pain-free and convenient.

6.1.2 experiments

the main task presented to the subject during any experiment is based around the persona of an engineer who has found themselves frequently in need of various scientific constants throughout day-to-day life. This persona is then motivated to create an application for the pinetime which may display a selection of string constants conveying the scientific values desired. The specific constants may be found in (A.1).

under both experiments
The participant is presented with a Manjaro virtual machine that has been pre-prepared with the 3 relevant repositories (glime, InfiniTime, and Wasp-OS), together with the necessary dependencies and libraries.

The participant is also supplied supplementary build scripts created for evaluation purposes to hasten the process. This is because the project's primary focus is on the development process,

not build, and helped with avoiding an unnecessarily long evaluation. Whenn using InfiniTime, the participant debugs their application with the use of InfiniEmu, a dedicated emulator for InfiniTime. this is also set up as part of preparation.

For Wasp-OS, the participant hands over the machine to the experiment supervisor, who flashes the produced image to a physical PineTime.

main experiment
Under the main experiment, the subject is presented with glime and modified InfiniTime/Wasp-OS documentation, to be used as guidance throughout the experiment. The expectation is that most information is learned from these documents, and experiment supervisor intervention should be minimal.

control group experiment
Under the control experiment, the subject is provided only with the original InfiniTime/Wasp-OS documentation providing guidance on app development.

6.1.3 experiment survey

The experiment survey varies across the standard experiment and the control one.

under both experiments The post-experiment survey aims to learn what platform(s) the participant used, how successful they were under what timeframe, and lastly open ended questions with regards to their success.

main experiment The main experiment form additionally tries to understand the participant's troubles specifically with the abstraction library, whether they felt it was too restrictive or lacking in any degree.

control group experiment The control group experiment form, on the other hand, investigates whether the platform was overly complex, confusing, or otherwise irritating to use.

6.1.4 evaluation justification with respect to project aims

to properly evaluate whether this project achieved it's aims, this evaluation primarily focusing on the development experience for a typical developer new to the PineTime platform.

This was chosen since the project's approach to achieving these aims is one which operates during the development stage. If the project is effective, this should make it clear whether aim 3 was met.

aim 2. and 4. is evaluated by the between-subjects design of the experiment, allowing participants to pick between either InfiniTime or Wasp-OS, meaning any differences in support between either OS should appear in the experiment results.

Aim 1. is not directly evaluated as part of this experiment, on the expectation that if app development with this project is preferable to the standard development process for an operating system, then the repertoire of apps written with this project should eventually (theoretically) grow to the point where a majority of applications available for the PineTime are written with support for an abstraction library, which should then mean an easier installation process across the board.

the control group's purpose is to provide a frame of reference for the difficulty of the task without using the project, to gauge exactly how much of an improvement the project provides over the regular experience developing "natively".

6.2 results

6.2.1 post-experiment survey

main group
There was a 50/50 split in participants between choosing InfiniTime and Wasp-OS. Note that this was not by design but purely by coincidence.

All participants had successfully created some form of application that executed, and all participants subjectively believed they had completed the objective to a satisfactory degree.

Participants took 47, 56, 60, 60, and 37 minutes, respectively.

In issues with building or debugging, 4/5 participants in the main group cited some form of inexperience with C for difficulties in this regard.
One participant cited complaints with the naming of the event listener register.
One participant described the github(s) documentation as unclear, and another user explicitly stated the compilation process was well-documented.

In general issues, 2 participants mentioned inexperience with C. One participant discovered that buttons were not fully implemented on the Wasp-OS side, and one participant experienced general issues with label positioning and display. one other participant also mentions an issue with not knowing the height of text, disallowing proper placement of the labels.

3/5 participants state there were no other functions they were expecting to see for this exercise. one participant again mentions the lack of button implementation on Wasp-OS, and one participant says they would like to see auto positioning of elements without the need for manual placement.

on the Likert scale:
4 participants believed the functions did not feel restrictive in their purpose, and one participant believed otherwise.
3 participants believed the functions provided did feel restrictive in their arguments, while 2 believed otherwise.
3 participants believed the functions provided were misleading in their method signature, while 1 disagreed and 1 felt neutral.
4 participants believed the functions provided did not feel incomplete, while 1 believed otherwise.
All 5 participants believed the functions provided did not feel inadequate.

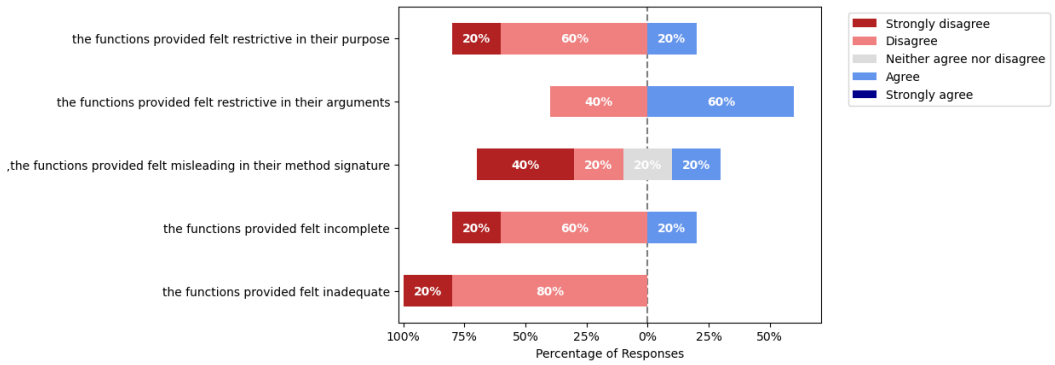


Figure 6.1: Likert responses from main evaluation group

1 participant additionally commented that they "felt that the function 'register_global_eventListener' should be related to touchCallback" Another participant additionally commented that they thought the current positioning of labels is enough to achieve small goals but more complex sizing and styling would be a nice feature to have.

in general, one participant stated they would prefer some re-usable components, potentially some sort of library,
one participant would prefer to have scrollable views where only a small window of the view is rendered at a time,
and one participant mentions a generic placement method that could automatically place elements without manual programmer alignment.

control group
By design, there was exactly 1 participant in the control group for InfiniTime and 1 participant in the control group for Wasp-OS.

All participants had successfully created some form of application that executed, and all participants subjectively believed they had completed the objective to a satisfactory degree.

Participants took 57 and 70 minutes.

On InfiniTime, debugging and building troubles relate to misleading documentation, rusty C++ experience, and oversized labels not fitting on the screen.
on Wasp-OS, debugging and building troubles related to poor documentation, overcrowded build output hiding errors, and further errors in reading documentation.

One participant filled out an optional commentary on general problems, where they mentioned troubles with Wasp-OS's simulator.

No responses were given on functions expected to see but not provided,

On the Likert questions, both participants were neutral with regards to the complexity of their respective development platforms
both participants disagreed that the platform felt intuitive (with strong disagreement for Wasp-OS)
The InfiniTime participant disagreed on the development platform feeling bloated, whereas the Wasp-OS participant felt neutral.
lastly, both participants agreed their respective development platforms felt restrictive.

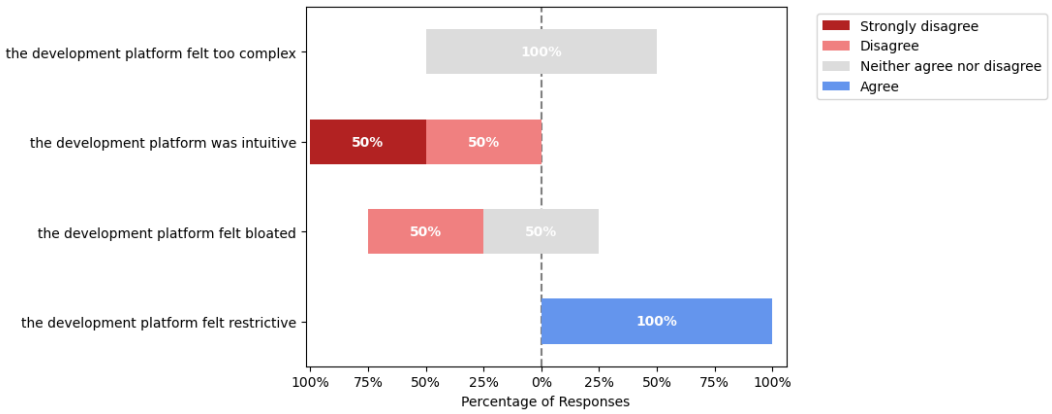


Figure 6.2: *Likert responses from control evaluation group*

In additional comments, the Wasp-OS participant described the platform as extremely unintuitive but not complex or bloated. The InfiniTime participant, on the other hand, wishes it'd be easier to place correctly sized bodies of text.

In commentary regarding frequent use, the Wasp-OS participant could consider repeated development on this platform but would be discouraged by the poor documentation. The InfiniTime

participant, on the other hand, states they would be content with developing for the platform regularly if it had more features, even if "it remained a little difficult to do so"

6.3 insights

1. provide a solution to simplify the process of installing and using others' apps on the PineTime smartwatch.
as previously mentioned, this evaluation does not directly assess whether the project meets aim 1. however, inspection of the times taken to completion of the application for the control group (57 and 60 minutes) versus those using the project (47, 56, 60, 60, 37) indicates that the app development process using the project is in fact generally more achievable with respect to the native process, which in turn should eventually make it easier for an individual to install someone else's application as the apps available for the PineTime slowly adopt the use of an abstraction library.

2. design a system with adequate support for available software but remain adaptable enough for future OSes not currently on the PineTime landscape.
The between-subject design of the experiment evaluating both InfiniTime and Wasp-OS makes a surface-level effort to evaluate the project on this aim, but it's not adequate for a definitive statement on whether this aim is met.

That is because problem analysis and requirements impose a reduced requirement to work with only 2 operating systems. This made the project achievable, but is detrimental to it's ability to demonstrate a versatile design that features transferability to other operating systems past InfiniTime and Wasp-OS.

However, this aim was a highly prioritised target in the design of this project, and the current experiment results can at least suggest that the project is at minimum somewhat close to achieving this aim, supported by one participant that tested their produced application on both operating systems, which demonstrates application portability is present and somewhat effective.

This may allow the conclusion that the project, at minimum, has somewhat met this aim.

3. achieve a minimum level of versatility to ensure the project is practically usable rather than being a proof of concept.
The extent to which this project met this aim is mixed.

feedback in the post-experiment survey indicates that the project is suitable for a basic level of development for PineTime applications, but is still lacking support in some areas e.g. in the versatility for placing/rendering labels.

Compared with the control group, however, a similar pattern was shown where the participants would've liked to see more versatility in general functionality.

as such, the lack of such functionality could be attributed as being a flaw of the underlying operating systems, which may be out of scope of this project to fix.

it still remains true that some parts of the specification for the abstraction library were not fully implemented, as discovered by one participant when attempting to use buttons, so this aim can not be considered a complete success, but a partial success here may still be adequate with respect to the minimum subset of features required for a UI toolkit to be practical.

4. strive to achieve these aims not just on one OS but rather across multiple host operating systems of the PineTime to avoid facilitating any form of vendor lock-in where any OS is significantly more featureful.
The extent to which this aim is met is indicated by the times participants took on each OS., that is 47 & 37 minutes on InfiniTime and 55 & 60 on Wasp-OS. Ideally, these times should be similar across operating systems, however, it is clear that participants generally took less time on

InfiniTime. This is contradictory to expectations, not only because the control group took more time on InfiniTime but also because InfiniTime is the more complex operating system.

This may be because InfiniTime participants were able to use the InfiniTime virtual machine available on the evaluation machine for debugging, whereas Wasp-OS participants did not have the same comfort and instead needed to offload the firmware file to a mobile device and then spend $\tilde{5}$ minutes waiting for it to flash to the watch.

regardless, these results together with the missing implementation for widgets on Wasp-OS, certainly indicate that InfiniTime is more well-supported by the project. this means this aim is not perfectly achived, however it's still true that both InfiniTime and Wasp-OS participants succeeded in making the application, something that is supported by the fact that the 3 main MVP applications are definitely fully supported on both operating systems.

7 Conclusion

7.1 Summary

This dissertation project aimed improve the user experience for the average PineTime user. to achieve this, a reduced aim was determined of providing some software solution for the PineTime watch that enabled the porting of individual applications to multiple operating systems with no need for rewrites, on the basis that this would reduce vendor lock-in to any particular OS.

Problem analysis suggests that the most sensible way to insert arbitrary code into embedded systems is through taking advantage of C's general interoperability with low-level embedded systems.

Further, background literature suggested that handling system APIs across operating systems when considering only open source operating systems running on the same hardware, is through the use of an abstraction library with a common interface but varying implementations according to the underlying OS.

Requirement analysis determined it would suffice to demonstrate this solution functioning only on the 2 main operating systems of the PineTime (InfiniTime and wasp-os), since they are not only the most dominant operating systems and thus set the standard for the whole platform, but are adequately different in implementation that success should prove the solution is generalisable to most operating systems thereafter.

A design was created which follows the requirements and takes inspiration from background papers, drawing out patterns between operating systems of the PineTime such as how events are handled or how hardware interactions occur, and then attempts to set appropriate precedent that should theoretically facilitate a simpler development process down the line both on the part of a OS maintainer and most importantly for an independent user of the PineTime using the abstraction library.

In implementation of this design, this dissertation demonstrates a minimum viable product functioning with 3 main applications (flashlight, heartrate, and paint apps), and an additional timer application as part of optional requirements, to investigate the viability of a full implementation with respect to InfiniTime's component architecture.

Evaluation of this work then showed that, at minimum, the resulting product could reasonably achieve the aims laid out to an acceptable standard, but was still unrefined and missing particular features and creature comforts that users were expecting. despite the lack of refinement, the evaluation still demonstrated the project was an effective proof of concept that was, most importantly, usable to a degree.

7.2 Reflection

7.2.1 design

The design of the abstraction API interface was created function-by-function, together at the same time as implementation. i.e. "place_label" was created in the library interface, and then it was

implemented on InfiniTime, and then wasp-os, and then the next function e.g. "register_event" would be next.

this piece-wise design meant that a general design was never pursued. this was potentially a flaw because creating an over-arching design top-down from the beggining would've simplified the implementation process, allowing it to be purely implementation without frequent design breaks in between.

most importantly, a general design is also necessary when trying to create what is partially a UI framework, because of how extensive the concept ofa UI framework is. such a tool that attempts to cover most possible use cases in UI can not possibly be created piece by piece without accruing a form of design debt that would eventually cause challenges.

while the design here was small enough to not make this obvious, it was definitely still noticed e.g. where the design of the touch interface reflects one-to-one how Infinitime and wasp-os use separate event handlers for swipe and touch, whereas an alternative method that combines both types of events may have been preferred.

also, while it's likely a design combining the two types of events is in fact unreasonable, the design was not even able to consider that option and understand the reasoning, which is a flaw of the design process.

7.2.2 implementation

Recall that the implementation phase was split up into a "preparation" and an "implementation+design" phase (referred to concisely as the implementation phase), which was each carried out once for each OS being supported. the preparation phase was unexpected to take as much time as it did, which left very little time for the implementation phase. as a result, the overall design of the abstraction API interface was severely lacking.

The design was able to reach an adequate level of support for user input having achieved touch+swipe events and buttons, but is particularly lacking in output. At present there is only basic label placement and coloured squares.

As part of the extra optional requirement, widget support was achieved only on the InfiniTime side, but not on wasp-os. This is a severe drawback since the native widgets are where most of InfiniTime and wasp-os's remaining power lies. If buttons, checkboxes, and radial selectors could've been implemented, then this project may have resulted in a respectably usable library, but in it's current state can only be used for basics.

7.2.3 Evaluation

this project featured several severe issues with it's evaluation. it featured several particularly broad aims, and these were all difficult to evaluate individually.

A solution to simplify app installation and usage could be effectively evaluated by an experiment tasking the user to install applications for the PineTime.

a system targeting support for multiple existing operating systems while maintaining adaptability for future operating systems, could be evaluated with an experiment tasking a participant to test the project across multiple existing operating systems, and then additionally task a participant to extend the project to an additional system.

whether the system is practically usable or not could be evaluated by tasking a user to implement a particularly complex application using the system, and then evaluate that whole process.

Whether the system achieves it's aims not just on one OS but on multiple could be evaluated by tasking a user to implement a simpler application but across multiple operating systems, and then

evaluate the application's success between target OSes.

this dissertation was naturally not able to carry out something on par with all of these evaluations individually. this was not only because it did not have the time nor the people available, but also needed to remain realistic with respect to how much of a participant's time can be taken, and also how many participants could be reasonably obtained.

as a result of all of this, the evaluation was restricted to a fairly broad one that could only lightly cover all of the aims.

7.3 Future work

Future work may include the following:

- more extensive and featureful abstraction API interface demonstrating e.g.
  - more extensive widget support
  - support for images/icons
  - support for BLE communication with PineTime companion apps
- better user-facing documentation of the abstraction API
- allow multiple applications on a watch to use the abstraction API
- implementation with a third OS to validate adaptability
- better evaluation(s)
- a deeper modification of host operating system to demonstrate an OS with no native API calls, exclusively providing those in the abstraction library.

A Appendices

A.1 evaluation ethics checklist

School of Computing Science
University of Glasgow

Ethics checklist form for 3rd/4th/5th year, and taught MSc projects

This form is only applicable for projects that use other people ('participants') for the collection of information, typically in getting comments about a system or a system design, getting information about how a system could be used, or evaluating a working system.

If no other people have been involved in the collection of information, then you do not need to complete this form.

If your evaluation does not comply with any one or more of the points below, please contact the Chair of the School of Computing Science Ethics Committee (matthew.chalmers@glasgow.ac.uk) for advice.

If your evaluation does comply with all the points below, please sign this form and submit it with your project.

---

1. Participants were not exposed to any risks greater than those encountered in their normal working life.

   Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback

2. The experimental materials were paper-based, or comprised software running on standard hardware.

   Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, laptops, iPads, mobile phones and common hand-held devices is considered non-standard.

3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.

   If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.

   Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.

4. No incentives were offered to the participants.

   The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

5. No information about the evaluation or materials was intentionally withheld from the participants.
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

6. No participant was under the age of 16.
   Parental consent is required for participants under the age of 16.

7. No participant has an impairment that may limit their understanding or communication.
   Additional consent is required for participants with impairments.

8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
   A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. All participants were informed that they could withdraw at any time.
   All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.

10. All participants have been informed of my contact details.
    All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.

11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
    The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. In cases where remote participants may withdraw from the experiment early and it is not possible to debrief them, the fact that doing so will result in their not being debriefed should be mentioned in the introductory text.

12. All the data collected from the participants is stored in an anonymous form.
    All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Project title abstraction over PineTime operating system APIs

Student's Name Andrei Boghean

Student Number 2645295b

Student's Signature Andrei Boghean

Supervisor's Signature G.Tang

Date 23/3/2025

A.2 simple evaluation application

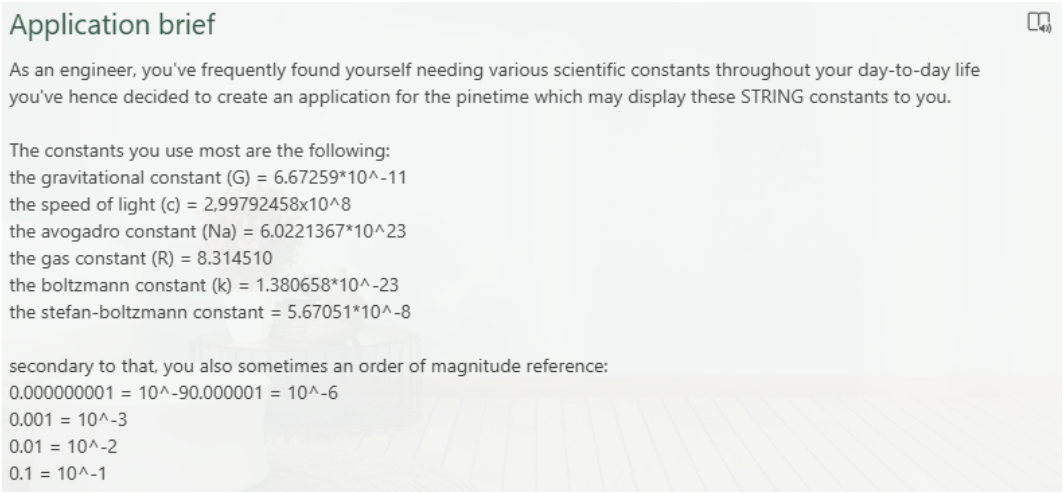


Figure A.1: *application brief for a simple PineTime application displaying useful string constants*

Bibliography

Amstadt, B. and Johnson, M. K. (1994), 'Wine', *Linux J.* **1994**(4es), 3–es.

Andrus, J., Van't Hof, A., AlDuaij, N., Dall, C., Viennot, N. and Nieh, J. (2014), Cider: native execution of ios apps on android, *in* 'Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems', ASPLOS '14, Association for Computing Machinery, New York, NY, USA, p. 367–382.
URL: *https://doi.org/10.1145/2541940.2541972*

build and program (n.d.).
URL: *https://github.com/InfiniTimeOrg/InfiniTime/blob/6a6981c91251b96cdf33fe9b094b0833b00ebd8f/doc/buildAndProgram.md*.

Building wasp-os from source (n.d.).
URL: *https://wasp-os.readthedocs.io/en/latest/install.html#building-wasp-os-from-source*.

Dunford, R., Su, Q., Tamang, E. and Wintour, A. (2014), 'The pareto principle', *The Plymouth Student Scientist* **7**(1), 140–148.

Extending MicroPython in C (n.d.).
URL: *https://docs.micropython.org/en/latest/develop/extendingmicropython.html*.

Gittens, M. and Godwin, D. (2007), 'A closer look at the pareto principle for software', *Software Quality Professional* **9**(4), 4.

HRS3300 Heart Rate Sensor (n.d.).
URL: *https://files.pine64.org/doc/datasheet/pinetime/HRS3300 Heart Rate Sensor.pdf*.

McKenzie, R. B. and Shughart, W. F. (1998), 'Is microsoft a monopolist?', *The Independent Review* **3**(2), 165–197.
URL: *http://www.jstor.org/stable/24561040*

Oliver, R. S., Shcherbakov, I. and Fohler, G. (2010), An operating system abstraction layer for portable applications in wireless sensor networks, *in* 'Proceedings of the 2010 ACM Symposium on Applied Computing', SAC '10, Association for Computing Machinery, New York, NY, USA, p. 742–748.
URL: *https://doi.org/10.1145/1774088.1774243*

Orr, B. (2003), 'Time for linux', *American Bankers Association. ABA Banking Journal* **95**(8), 51.

PineTime Companion Apps (n.d.).
URL: *https://wiki.pine64.org/wiki/PineTime#Companion_Apps*.

Puder, A. (2010), Cross-compiling android applications to the iphone, *in* 'Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java', PPPJ '10, Association for Computing Machinery, New York, NY, USA, p. 69–77.
URL: *https://doi.org/10.1145/1852761.1852772*

SleepTk InfiniTime porting bounty (n.d.).
URL: *https://github.com/thiswillbeyourgithub/SleepTk_pinetime_sleep_tracker/issues/13*.

Stroustrup, B. (2002), 'C and c++: A case for compatibility', **20**.