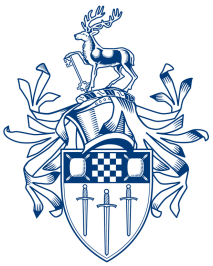

Bachelor's Dissertation

An Analysis of the Rust Programming Language for Embedded Systems

William Daniel

Faculty:	Department of Computer Science & Electronic Engineering
1st Supervisor:	Dr. Nick Frymann
2nd Supervisor:	Dr. Stella Kazamia
Date:	14/05/2025

Declaration

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

On this day, May 14th 2025.

William Daniel

Abstract

Embedded systems form the operational backbone of the Internet of Things, yet their pervasive reliance on C/C++ programming languages continually exposes them to critical memory safety vulnerabilities, which are responsible for a significant majority of security incidents and system failures. The Rust programming language is emerging as a transformative alternative, developed to provide compile-time memory safety guarantees without the performance degradation typically associated with garbage collection, positioning it as a compelling candidate for developing highly robust and secure embedded software. This dissertation critically assesses Rust's practical viability as a comprehensive replacement for C in the contemporary embedded systems domain.

The core hypothesis of this research is that Rust can not only match the low-level control and performance characteristics of C in typical embedded applications but can also demonstrably enhance software safety and developer productivity. To investigate this, an embedded project that simulates a real–time system—a morse code encoder and decoder utilising RP2040 microcontrollers, was implemented, first in C and subsequently migrated to Rust. This comparative development covered both bare-metal programming paradigms, for direct hardware control, and higher-level abstractions using Software Development Kits for C and Hardware Abstraction Layers for Rust. These distinct implementations were then subjected to a comparative analysis. This involved comprehensive static analysis of generated binaries to quantify memory footprints, static RAM and Flash benchmarks to identify code size contributors, dynamic performance benchmarking measuring key peripheral operation speeds and interrupt latencies, crucial for real-time systems and IoT engineering.

The empirical evidence gathered supports the hypothesis. It demonstrated that Rust's compile-time safety mechanisms, particularly its ownership and borrowing system, effectively preclude common C-family memory errors, thereby inherently enhancing software security. Performance analysis revealed that optimised Rust HALs can not only match but, in specific instances such as GPIO toggle speed and interrupt latency consistency, demonstrably exceed the performance of optimised C SDKs, while also offering superior static RAM efficiency. However, we experienced the steeper learning curve present for developers that are accustomed to C. Furthermore, while Rust's embedded ecosystem is experiencing rapid maturation and enthusiastic community support, it has not yet achieved the sheer breadth and depth of C's decades-old infrastructure and vendor support.

This study concludes that Rust is a decidedly viable and increasingly attractive paradigm for developing safer, more secure, and often equally, if not more, performant embedded systems. Its adoption represents a significant step towards mitigating persistent software vulnerabilities. The key contributions of this work include a detailed, evidence-based comparison of C and Rust across multiple abstraction levels on a contemporary microcontroller, practical insights into the migration process, and a nuanced evaluation of both technical and non-technical factors influencing Rust's adoption. Widespread industry transition will, however, depend on concerted efforts to flatten the learning curve through enhanced educational resources and continued strategic development of its ecosystem.

Acknowledgments

I'd like to first thank Nick for proofreading the thesis, providing me with advice while also letting me undertake this project in September. Thanks to Stella for providing me with an abundant amount of references to overlook for my literature review, and my friends and family who have supported me through my time at university.

Contents

List of Figures

List of Tables

List of Listings

Glossary

Bare-metal: Programming directly on hardware without an operating system.

Cyclic Redundancy Check: An error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data.

Executable and Linkable Format: A common standard file format for executables, object code, shared libraries, and core dumps.

Foreign Function Interface: A mechanism by which a program written in one programming language can call routines or make use of services written in another.

Flash Memory: An electronic non-volatile computer memory storage medium that can be electrically erased and reprogrammed.

General-Purpose Input/Output: A generic pin on an integrated circuit whose behavior—including whether it is an input or output pin—is controllable by the user at run time.

Hardware Abstraction Layer: Software layer that provides a consistent interface to hardware peripherals, abstracting away hardware-specific details.

Inter-Integrated Circuit: A synchronous, multi-master, multi-slave, packet switched, single-ended, serial computer bus.

Memory Protection Keys: A hardware feature on Intel CPUs that allows user-space applications to set protection attributes on pages without making system calls.

Mid-level Intermediate Representation: An intermediate representation used by the Rust compiler.

Modular Symbolic Execution: A formal verification method.

Monomorphization: The process in Rust where generic code is translated into specific code for each concrete type it's used with.

no_std: A Rust attribute indicating that the program will not link against the standard library, common in embedded development.

Peripheral Access Crate: In Rust, a crate providing low-level register definitions for microcontroller peripherals.

Real-Time Operating System: An operating system intended to serve real-time applications that process data as it comes in, typically without buffer delays.

Software Development Kit: A collection of software development tools in one installable package.

Static Application Security Testing: A set of technologies designed to analyze application source code, byte code and binaries for coding and design conditions that are indicative of security vulnerabilities.

Static Random-Access Memory: A type of semiconductor memory that uses bistable latching circuitry to store each bit.

Unikernel: A specialized, single-address-space machine image constructed by using library operating systems.

Abbreviations

ADC: Analog-to-Digital Converter
AST: Abstract Syntax Tree
CRC: Cyclic Redundancy Check
CWE: Common Weakness Enumeration
EABI: Embedded Application Binary Interface
ELF: Executable and Linkable Format
FIFO: First In First Out
FFI: Foreign Function Interface
GDB: GNU Debugger
GPIO: General Purpose Input-Output
HAL: Hardware Abstraction Layer
IDE: Integrated Development Environment
I^2C: Inter-Integrated Circuit
IoT: Internet of Things
ISR: Interrupt Service Routine
LTO: Link Time optimisation
MPK: Memory Protection Keys
NMI: Non-Maskable Interrupt
OCD: On-Chip Debugger
OE: Output Enable
PAC: Peripheral Access Crate
PWM: Pulse-Width Modulation
RTT: Real-Time Transfer
SDK: Software Development Kit
SSI: Synchronous Serial Interface
SPI: Serial Peripheral Interface
SVD: System View Description
UART: Universal Asynchronous Receiver-Transmitter
UB: Undefined Behavior
UF2: USB Flashing Format
VM: Virtual Machine
VTOR: Vector Table Offset Register
XIP: Execute-In-Place

1 Introduction

1.1 Preamble

The use of embedded devices has become prevalent as the internet begins to progressively scale upwards at an ever-increasing rate. The interconnection of devices globally through the *Internet of Things* (IoT) amplifies the need for these systems to be safe, secure and reliable. Mobile devices and smart infrastructure for transport, medical, automation and home appliances are only a few examples demonstrating its importance in the modern day. Disruption to any of these would have catastrophic consequences. The implementation of such systems requires low-level languages to interface with the embedded hardware layer of the microcontroller and connected peripherals.

Due to its capabilities, Rust is becoming increasingly popular for embedded systems, promising memory safety guarantees without runtime overhead through its unique ownership system. Developed by Mozilla Research, it represents a new paradigm shift by combining low-level control with high-level and zero-cost abstractions. With its type system guaranteeing the absence of data races, buffer overflows, stack overflows and accesses to uninitialized memory, a considerable amount of attention has been drawn to the language.

Despite these features, there exists a considerable amount of legacy code present within modern systems. Companies and educational institutions are reluctant to teach and replace code testing over years with something new, with embedded systems still heavily reliant on the C and C++ programming languages. While Rust is garnishing attention, the general community places it 19th on the TIOBE Index as of May 2025, with its popularity decreasing from the prior year and having a rating of 8.77% less than C [1]. It appears industries that heavily emphasise embedded development face the challenge of balancing the need for system reliability and security with the practical constraints of existing infrastructure. While C and C++ have served as the foundation for such development for decades, their inherent lack of memory safety guarantees has led to numerous security vulnerabilities and system failures; studies from Google and Microsoft Research reporting figures as high as 70% of all security vulnerabilities were caused by memory safety issues [2]. The recent CrowdStrike incident that affected 8.5 million systems and caused an estimated $10 billion in damages [3] illustrates how memory-related vulnerabilities in low-level programming can have catastrophic consequences. Despite advancements in testing and debugging tools, these issues remain persistent.

1.2 Aims and Objectives

This dissertation confronts these concerns directly, investigating Rust's practical viability as a successor to C within constrained embedded systems. While some studies have touched upon Rust's capabilities, a frequent oversight is a detailed, comparative analysis of real-time performance and nuanced software-hardware interactions against C in practical embedded scenarios —this investigation seeks to fill that void [4]. It will employ the migration of a feasible embedded project using the most common type of embedded device, a constrained microcontroller within a bare-metal and SDK/HAL environment written in C to Rust to highlight the technical and non–technical aspects when considering the migration and how the project can be used as a synonym for real–world applications when scaled up, primarily in the cybersecurity and defence sectors.

We aim to answer the following:

1. How effectively does Rust's compile-time safety mechanisms, particularly its ownership and borrowing system, prevent common memory-related vulnerabilities prevalent in C-based embedded development, and what is the practical impact on software robustness?
2. Does Rust, across different abstraction levels, achieve performance comparable to, or even exceeding, equivalent C implementations for typical embedded peripheral operations and real-time tasks on a resource-constrained microcontroller like the RP2040?
3. Can Rust meet the deterministic and real-time processing requirements of embedded systems as effectively as C, particularly concerning interrupt handling and predictable execution times?
4. What is the relative maturity, usability, and comprehensiveness of the tooling, libraries and community support for embedded Rust development on a modern microcontroller such as the RP2040 in 2025 when contrasted with the established C ecosystem? What are the implications of Rust's learning curve and its unique programming paradigms for developer productivity and the migration of existing C skill sets?
5. Synthesizing the findings on safety, performance, real-time capabilities, and ecosystem factors, what is the overall practical viability of Rust as a strategic replacement for C in future embedded systems development, particularly for applications demanding high security and reliability?

To comprehensively address these questions, this project pursues the following objectives:

1. Implement and migrate a bare-metal and morse transmission SDK project in C between RP2040 microcontrollers to Rust.
2. Conduct controlled experiments using the SDK project to evaluate safety and performance.
3. Compare both languages in relation to the bare-metal and SDK implementations, focusing on the technical and non-technical aspects.
4. Evaluate the results of the controlled experiment to determine Rust’s advantages and challenges in regards to safety and performance.
5. Conclude by discussing the findings, including safety, security, performance, design patterns, and lessons learned.
6. Offer recommendations on the feasibility of adopting Rust for embedded systems at scale.

1.3 Risks

The ecosystem for Rust in embedded development, though growing, remains less mature compared to C. Libraries like rp-hal and debugging tools such as probe-rs are still evolving, and developers often need to implement custom solutions for hardware interactions. Second, Rust's steep learning curve poses a barrier for developers accustomed to C, particularly in understanding concepts like ownership and lifetimes. Additionally, migrating legacy codebases to Rust is non-trivial, as direct translation is rarely feasible due to differences in language paradigms. Finally, while Rust’s emphasis on safety is a strength, it can make certain low-level optimisations more complex, particularly in time-sensitive tasks.

2

Literature Review

2.1 “C” is for Casualties

Huge security vulnerabilities arise when features within low–level programming languages are exploited or left unhandled. On the 19th of July 2024, the world witnessed the largest outage in history through the CrowdStrike incident that went on to affect 8.5 million systems [3] and many industries—airlines, airports, banks, hotels, hospitals, manufacturing, stock markets, broadcasting, gas stations, retail stores—as were governmental services, such as emergency services and websites, with damage estimates to be at least $10 billion. An internal post-incident investigation revealed one of the issues arising from a missing runtime array bounds check for input fields on channel files. Specifically, the vulnerability was in how their driver handled an IOCTL request in regards to array index validation. In C, the length of arrays must be treated and checked separately. However, the length was not checked before access, raising an out-of-bounds exception. While the incident was not solely this the fault of this bound check in C, and third-party proprietary software operating in kernel space is inherently dangerous, the point is to illustrate that Rust in such situations is better at being able to make an underlying error paths visible to be exposed to review and coding convention, a fault that the CrowdStrike team were unable to catch before moving to production [5]. Large companies such as CrowdStrike are assumed to always be actively attempting to write safe and secure code. While testing is number one priority before roll-out, it makes it much easier for non-obvious null dereference or use-after-free errors within large code bases in C, even using best practices. It is clear that regardless of high investments into improvement, such problems are not solved by supporting the developers through coding guidelines, finding issues through code analysis or other measures.

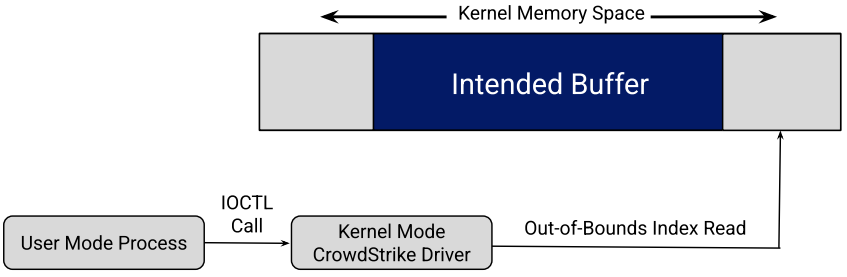

Figure 1: CWE-125 Out-of-Bounds Read

Eliminating memory errors is important in all software systems. The same standard is to be applied to embedded systems. Cases like this explain the pursuit of a safer, more resilient environment. Systems programming, an area with substantial embedded development, has intrigued The Department of Defense as they begin to take an interest in the conversion of C to Rust for software development with the announcement of the TRACTOR project [6]. The Defense Advanced Research Projects Agency (DARPA) has recognized the potential in Rust’s features, citing that the software engineering community had reached a consensus and that it was “not enough” to rely on bug-finding tools. In the defence sector’s own words, the preferred approach is to use safe programming languages that can reject unsafe programs at compile time, preventing the emergence of memory safety issues. This makes sense in the context of safety-critical embedded systems. While all the concepts are still in their early stages of

development, it is clear to see a future where the programmer is stress-free from issues the compiler should be handling. Question is, why isn't this already the norm? At a glance, Rust appears well-suited for embedded systems, offering strong safety guarantees while maintaining performance comparable to C. However, the migration faces some technical challenges.

2.2 Ecosystem

Existing Rust support is heavily centered on ARM Cortex-M MCUs, with architectures such as AVR and RISC-V still remaining underrepresented. HAL crates are available for only 32% of MCU families, and peripheral access crates exist for 37%. This leaves significant gaps in software support, particularly for non-ARM architectures [6]. Furthermore, many crates rely on unsafe wrapper implementations around C libraries, inheriting vulnerabilities that compromise their robustness. Tooling for Rust in embedded systems also faces notable challenges. SAST tools, critical for identifying security vulnerabilities, struggle with embedded idioms and no_std configurations. High false-positive rates (40%-90%) and incompatibilities with embedded build processes limit their usability. Interoperability remains a significant hurdle, as most Rust embedded crates rely on Foreign Function Interface (FFI) calls, but approximately 70% of these crates use types incompatible with C, making integration labor-intensive. While tools like bindgen simplify Rust-to-C function calls, C-to-Rust transitions require extensive manual effort to engineer compatible wrappers [7].

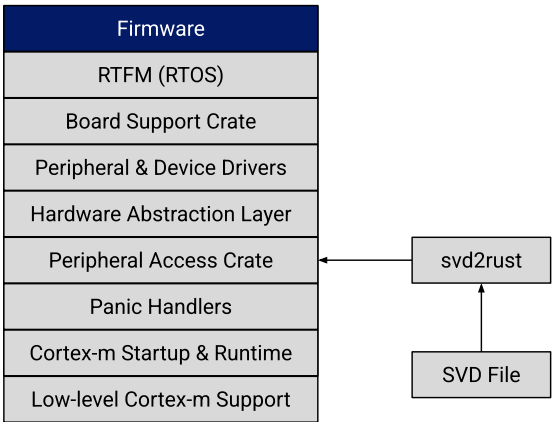

Figure 2: The Embedded Rust Ecosystem

Standard libraries are important in any software-driven environment. They allow developers to focus on the unique aspects of their application rather than reimplementing common features from scratch. Studies reveal that only ~10% of all Rust crates support no_std environments as of 2024 [7], suggesting limitations in library availability. Even so, many core embedded development needs are already well-served by mature no_std crates, with frameworks like embedded-hal providing robust foundations for hardware abstraction. Embedded-hal has evolved to offer standardized traits and implementations for common peripheral types, enabling portable driver development across different microcontrollers [8]. Embedded developers typically want to remove dependency code, provide their own core data structures and work away from dynamic memory allocation. The attribute #![no_std] is used to provide this functionality instead. Furthermore, a significant portion of standard library crates were never relevant for embedded use cases, making their lack of no_std compatibility a non-issue for practical embedded development. The slow progression of Rust's adoption in industry can be partially attributed to these challenges; libraries in Rust are in constant development, often having to be built from the ground up only increasing the chance for unsafe library mistakes. Automated tools such as Aunor have emerged to address this by converting standard libraries

into no_std-compatible versions at scale [9]. Though these conversion tools fail to possess a consistent track record. c2rust similarly falls short, failing on 93.8% of tested embedded C codebases due to their inability to handle non-standard compiler flags and aforementioned no_std requirements [7]. Developer feedback underscores the need for better documentation. Current resources lack sufficient examples tailored to embedded systems, making it harder for developers to adopt Rust in this domain. Surveys indicate that 72% of developers prefer purpose-built no_std libraries over converted ones [7], reflecting the practical difficulties being encountered.

Despite this, the Rust community has made significant progress in addressing ecosystem gaps. Tools like Cargo, Rust's build system and package manager, simplify dependency management and facilitate reproducible builds. Rust's compiler, rustc, plays a pivotal role in reinforcing the language's appeal for embedded systems with its detailed and actionable error messages that reduce debugging time. Unlike traditional C compilers like GCC or Clang, which focus solely on optimisation, rustc prioritizes safety alongside its ownership model and borrow checker. It's reliance on LLVM ensures that it generates optimised machine code.

2.3 Performance

Rust's ability to enforce safety at compile time significantly reduces the likelihood of errors arising. Plus since Rust utilizes LLVM, any performance improvements in LLVM also carry over to Rust [10]. Unlike C and C++, Rust allows for reordering struct and enum elements to reduce the sizes of structures in memory, for better memory alignment, and to improve cache access efficiency [11][12]. Performance testing shows implementations within 1-2% of equivalent C code [13],with most of Rust's safety guarantees imposing no runtime overhead. The exception is array bounds checking, though its impact on performance is typically minimal due to modern processor branch predictions [13][14]. These metrics and a handful of similar studies seem to primarily focus on general-case scenarios with algorithms rather than worst-case execution times, and do not consider timing-critical safety issues, such as those related to real-time constraints or interrupt handling, which are generally more important to consider for embedded systems operating in real time [15][16][17]. In this case, C's simpler memory model often offers more predictable behavior that programmers can easily discern.

Binary size is an important consideration as on average flash sizes typically reach, at most, 512 KB on modern microcontrollers, with a few extending into the MB ranges. Rust statically linking its standard library can cause even a basic "Hello World" program to reach a binary size of 1.6-3.2 MB compared to C's 300-400 KB footprint [18] [19], giving the impression that Rust is unsuited for embedded deployment. Though this only results from naive use of the language. Stanford University identified monomorphization, inefficient derivations such as #[derive(Debug)]), implicit data structures such as vtables, and limited compiler optimisations as key contributors to this issue [19]. Replacing `#[derive(Debug)]` with custom implementations reduced binary size by 112 bytes per instance in a test case. By extension the same can be said with custom panic handler implementations. Monomorphization alone accounted for a 26% size increase in a TockOS kernel binary due to duplicate code generated for generics. Hidden data, such as static initializers, also led to unnecessary memory allocation, particularly for types with non-zero default values. By following size-reduction principles, including minimising generic scopes and using dynamic dispatch selectively, the authors reduced an industrial Rust firmware binary by 19% and an open-source kernel by 26% [20]. Other methods include stripping debug symbols and using the panic=abort flag to eliminate unwinding machinery. This can be still identified as a constraint, as a programmer has to take

more steps to get their code fit on flash in Rust compared to languages such as C. Tools like cargo-bloat and third-party libraries such as μfmt have emerged to help developers in this regard [21].

2.4 Safety

Rust isn't perfect. Unsafe Rust exists as a subset of the language that bypasses some of the safety guarantees. This is denoted with the unsafe keyword that allows developers to write blocks of code that perform five key operations: dereferencing raw pointers, calling unsafe functions, implementing unsafe traits, accessing mutable static variables, and accessing fields of unions [22], key operations in embedded development when dealing with low-level register manipulation, direct memory access, and device driver implementations. Brown and Carnegie Mellon University's mixed-methods study of 19 developers and 160 survey respondents found that 77% of developers used unsafe code as they perceived no viable safer alternative against the restrictive compiler, with 88% of respondents attempting to encapsulate unsafe code beneath safe APIs (88% of respondents) though only 23% were consistently confident their encapsulations were sound in all situations [23]. Larger sample sizes for the study targeted at embedded developers should be intended for future work. Fortunately, Modular Symbolic Execution (MSE), a formal verification method proposed by the RustBelt framework addresses these risks by leveraging formal semantics to verify soundness of unsafe blocks. By interpreting Rust's types in Separation Logic, MSE ensures memory safety and functional correctness while verifying safe abstractions [24]. This and tools like Miri [25] that interpret Rust's Mid-level Intermediate Representation (MIR) for undefined behavior detection are a step in the right direction despite some clear disadvantages. There's just one issue; Miri is designed to find undefined behavior in safe Rust code. Currently, these tools aren't well-suited for testing embedded applications as they are unable to can't simulate hardware without replication of a virtual setup, doesn't understand typical entry points like those defined by cortex and fails to operate well within no_std environments. Furthermore, they'd require manual effort to annotate unsafe code imposing a cognitive burden on developers and struggle to integrate into their workflow. Given Rust's lack of a memory model, strides are still yet to be made in formal verification. With the separation of MIR and hardware being too far apart, attempts at moving safe Rust code further down the layer of abstraction are being attempted, with λ_{Rust} being an example. While still an active area of research, it is plausible that in the future that MSE and other related formal checks become integrated into CI pipelines or IDEs to streamline the integration of formal verification and foster safer embedded programming practices [26].

$$
\begin{aligned}
&Path \ni p ::= \ x \mid p.n \\
&\quad Val \ni v ::= \ false \mid true \mid z \mid \ell \mid funrec\, f\ (x)\ ret\ k := \ F \\
&\quad\ Instr \ni I ::= \ v \mid p \mid p1 \ + \ p2 \mid p1 \ - \ p2 \mid p1 \leq p2 \mid p1 \ = \ p2 \mid new(n) \mid delete(n,\ p) \\
&\qquad\qquad \mid *p \mid p1 := \ p2 \mid p1 := n * p2 \mid p\ inj\ i{:}{=}{=}\ () \mid p1 inj\ i{:}{=}{=}\ p2 \mid p1 inj\ i{:}{=}{=}\ n * p2 \mid \ldots
\end{aligned}
$$

$$
\begin{aligned}
&FuncBody \ni F ::= \ let\ x \ = \ I\ in\ F \mid letcont\ k(x) := \ F1\ in\ F2 \mid newlft;\ F \mid endlft;\ F \\
&\qquad\qquad \mid if\ p\ then\ F1\ else\ F2 \mid case\ *p\ of\ F \mid jump\ k(x) \mid call\ f\ (x)\ ret
\end{aligned}
$$

Listing 1: RustBelt's Formal Syntax of λ_{Rust}

The presence of these trade-off issues has led to the research of new methods to better establish C to Rust as a viable migration. Unikernels are lightweight, specialized operating systems that combine application code and OS functionality into a single VM image. With origins in cloud computing, they are also particularly well-suited for embedded systems, often compared to

RTOS in their ability to minimise guest OS overhead. Unishyper, the first embedded unikernel to achieve thread-level memory isolation between user code and kernel code and isolation between different user applications, is an example of the action being taken to address these concerns. Based on Intel Memory Protection Keys (MPK), the implementation uses hardware-assisted page-granular memory access control to achieve isolation between different threads while maintaining single address space efficiency through the Zone memory isolation mechanism, reducing context-switch overheads [27]. Zones are defined as isolation domains, uniquely identified by Zone IDs, which correspond to Intel MPK protection keys. Threads are assigned ZoneKeys, restricting memory access to their allocated regions. Based on the DWARF standard [28], it uses an unwind module to gracefully handle any runtime failures, a useful feature in safety-critical applications that ensures fault isolation.

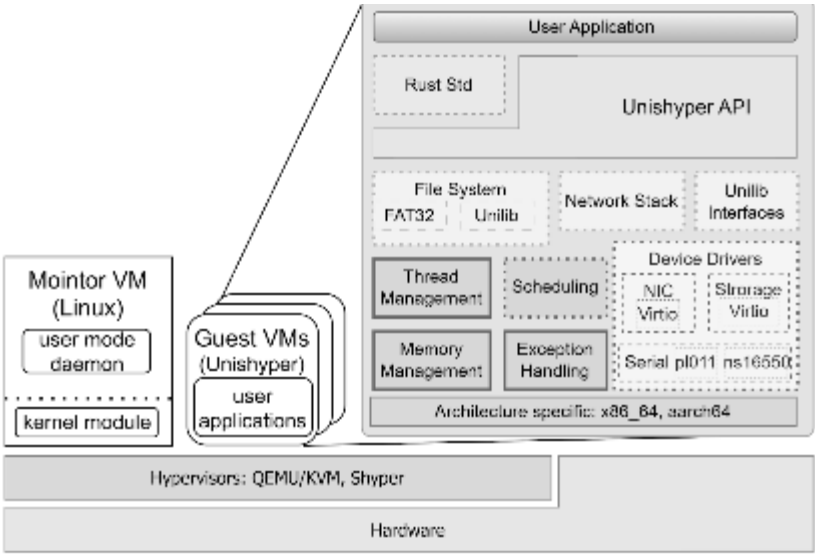

Figure 3: An Overview of the Unishyper Unikernel

Unishyper represents a step forward in embedded memory safety for Rust, but it still fails to address mechanisms to isolate unsafe Rust code from safe code without relying on hardware. Unikernels as a concept face notable criticisms that challenge their viability in production systems. By design, they merge application and operating system functionality, running entirely in the microprocessor's privileged mode. This eliminates the traditional boundary between user and kernel space, reducing overhead but introducing notable drawbacks. For instance, unikernels often lack essential debugging tools, such as ps, netstat and tcpdump making troubleshooting production issues difficult using these interfaces. While unikernels do provide a smaller attack surface, their single-address-space (SAS) model violates the principle of least privilege, meaning that any application vulnerability can compromise the entire system [29].

The evidence suggests that while Rust offers compelling advantages for embedded development, successful adoption requires careful consideration of significant technical and ecosystem limitations. Its future in embedded systems likely depends on continued maturation of the ecosystem, particularly in areas of formal verification and real-time guarantees. When compared to established embedded development approaches in C, Ada, and modern alternatives like MicroPython, Rust occupies the middle ground. It advertises stronger safety guarantees than C and better performance than MicroPython, but lacks Ada's mature certification pathways and C's mature ecosystem.

3

Programming the µC

The focus of this chapter is to lay the foundation on the design rationale behind key decisions, the technical challenges and how these implementation choices were designed to create a fair basis for the comparative evaluations detailed in subsequent chapters.. The overarching goal was to ensure that both the C and Rust implementations were functionally equivalent where possible, developed with best practices appropriate to each language and abstraction level, and sufficiently instrumented to yield meaningful comparative data. The first sequence of objectives is to get the microcontroller to blink an LED and fire a passive buzzer through a button press in a bare-metal environment. Implementations were undertaken within both bare–metal and SDK/HAL environments.

The hardware used is the Raspberry Pi Pico, which contains the dual-core ARM Cortex-M0+ RP2040 microcontroller [30]. The board itself includes a user-programmable LED, GPIO pins for external connections, and a USB port that can be used for both programming and debugging. A second Pico is used for debugging through a Serial Wire Debug (SWD) interface and to play the role of the receiver.

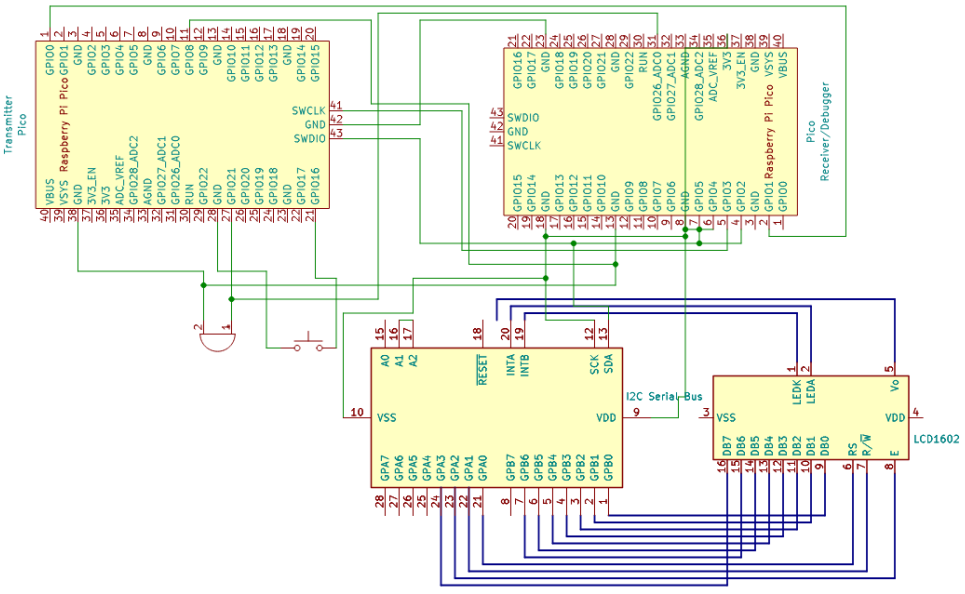

Figure 4: Breadboard Schematic

On the breadboard are wired connections to GPIO16 for a pull–up resistor button and GPIO21 for a passive buzzer [31]. Despite not being used in this project, we have a 7–segment display, two potentiometers, a spare button and two LEDs. These are not present in the schematic. Connections from the 7-segment to the receiver Pico have been made where we have pin 0 to the 7-segment pin A, GPIO pin 1 to pin B, and so on. The VCC pin for the segment is connected to the Pico's 3.3V power pin and the GND pin to the Pico's GND pin. Functionality of the 7-segment was handed over to the LCD1602 that comes integrated with an asynchronous serial bus over I^2C [32] with physical connections to the LCD's GND pin and the Pico's GND pin, the

LCD's VCC pin to the Pico's VBUS (3.3V) and the LCD's SDA pin to GPIO0 and the LCD's SCK/SCL to GPIO1 respectively.

A 3-pin wire connects SWCLK and SWDIO pins from the target to the debug Pico's, leaving the other to be connected to GND. Now the microcontroller can now be connected via a USB port. Firstly, we need to get by the `picoprobe.uf2` firmware from the Raspberry Pi GitHub repository by running `git clone https://github.com/raspberrypi/picoprobe` and mounting the built firmware to the debug Pico. To ensure the firmware is built for the debug Pico, it's important to run `cmake -G "MinGW Makefiles" -DDEBUG_ON_PICO=ON` prior to running `make -j4`.

To use the debugger on Windows we need a special driver to have OpenOCD communicate with the picoprobe firmware and it requires us to download Zadig. On this program we select Picoprobe from the drop-down menu and select libusb-win32 as the new driver and install it [33]. In a separate terminal, we can use the Open On-Chip Debugger (OpenOCD) tool to connect to the debug Pico through the following command:

```
> openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg -c "adapter speed
5000"
```

Listing 2: OpenOCD Terminal Configuration

Within this open terminal, OpenOCD provides port 4444 to execute manual commands using Tcl, a GNU Debugger (GDB) server on port 3333 for the GNU debugger to connect to and an RPC server on port 6666 for automated Tcl scripting. For debugging on OpenOCD throughout this project, port 3333 was used.

3.1 Using the C Language

Development is done entirely on Windows on the C23 standard. To get started we needed to be able to compile C and C and C++ code for the RP2040. Installation of the GNU Arm Embedded Toolchain allowed for this as it contains the compiler needed. Once installed and the path is added to the list of environment variables, we are able to make calls such as `arm-none-eabi-gcc` [34]. When building Pico projects, we need to compile the `elf2uf2` and `pioasm` tools from source. These tools run on our computer and require us to have a native compiler and linker. MinGW is a collection of open-source utilities, such as compilers and linkers that allow us to build applications on Windows for these sorts of purposes. Once unzipped, the following is entered into the terminal:

```
> echo mingw32-make %* > C:\VSARM\mingw\mingw64\bin\make.bat
> echo "alias make=mingw32-make.exe" >> ~/.bashrc source ~/.bashrc
```

Listing 3: Batch File Generation

This creates a wrapper batch file that will call the `mingw32-make` tool whenever `make` is typed into a Windows terminal. After ensuring that `mingw64 bin` is set as an environmental variable, the second command will allow the Pico to be built from anywhere [35].

3.1.1 Boot-Sequence

The RP2040 follows a two-stage boot process. The first stage separates the boot sequence into the "hardware-controlled" section which happens before the processors begin executing the boot ROM, and the "processor-controlled" section during which processor cores 0 and 1 begin to

execute the boot ROM [36]. The first stage was not important for us since its job is to safely turn on the minimum number of peripherals required for the processors to begin executing code, and not actually executing the code. This leads us to writing the second stage responsible for this, which lives at the beginning of our program, then runs after the boot ROM:

```
#include <stdint.h>
#include <stdbool.h>

/* Base address for XIP (Execute In Place) peripheral */
#define XIP_BASE                    (0×10000000)

/* SSI (Synchronous Serial Interface) Registers
 * Used for communication with external flash memory */
#define SSI_BASE            (0×18000000)
#define SSI_CTRLR0          (*(volatile uint32_t *) (SSI_BASE + 0×000))
#define SSI_SSIENR          (*(volatile uint32_t *) (SSI_BASE + 0×008))
#define SSI_BAUDR           (*(volatile uint32_t *) (SSI_BASE + 0×014))
#define SSI_SR              (*(volatile uint32_t *) (SSI_BASE + 0×028))
#define SSI_DR0             (*(volatile uint32_t *) (SSI_BASE + 0×060))
#define SSI_SPI_CTRLR0      (*(volatile uint32_t *) (SSI_BASE + 0×0f4))

/* ARM Cortex-M0+ Core Registers */
#define M0PLUS_BASE                 (0×e0000000)
#define M0PLUS_VTOR                 (*(volatile uint32_t *) (M0PLUS_BASE + 0×ed08))

/* Boot stage 2 entry point
 * This function is placed in .boot2 section by our linker */
__attribute__((section(".boot2"))) void bootStage2(void) {  ...  }
```

Listing 4: `bootStage2.c`

All RP2040 components fit on a single chip. To work with them, we need to be able to access them. Our first action is defining our registers so we can control our peripherals. These components live at fixed addresses and we fetch them as defined in the datasheet so that they can be accessed easily. As a prerequisite, we start off by instructing the compiler to put the `bootStage2()` function into a linked section called `.boot2` so we can put it at the beginning of flash memory [38]. For this function to work:

1. It has to be 252 bytes in size.
2. It must exist at the start of the flash memory.
3. The code in boot ROM should see it as valid.

Our program's code is stored in external flash memory. When the RP2040 boots, its boot ROM checks the first 256 bytes of flash memory with all the possible combinations of SSI clock phase and polarity to verify its validity [36], with the last 4 bytes being a CRC. If valid, our boot code is copied into SRAM and executed. The project uses an open-source library, `CRCpp`, to calculate this CRC. We define a `crc32.cpp` file that will load the contents of our resultant binary file into a 252 byte long array. Then we calculate the CRC32 checksum using this library with specific parameters corresponding to the CRC-32/MPEG-2 algorithm.

The bottom half of the code is omitted for simplicity. Using the registers, it performs several functions:

1. It configures the SSI controller to communicate with the external flash memory.
2. It skips explicit XIP cache configuration as the datasheet confirms it's enabled by default on page 128 [39].

3. It sets up the Vector Table Offset Register to point to our vector table in flash (`XIP_BASE + 0x100`). Loads the initial stack pointer value from the first entry and jumps to the reset handler function using the second entry.

This third function is an important design decision. The RP2040 does not follow the standard boot process that many ARM-based microcontrollers do; it contains the hard coded vector table within its boot ROM [40]. So this configuration allows us to use our user provided vector table to exist at a specific location instead for the next implementation step. At this stage, the processor cannot yet execute code from flash directly because flash memory is not yet accessible via the Execute-In-Place interface to allow the processor to fetch instructions. The idea is simple; convert the read access requests from the processor into SPI commands, acting as a Bus-To-SPI translation layer between the RP2040's processor and flash memory that can communicate over SPI [41]. This was the next goal to implement. Here, we use Standard SPI Mode 0 where the clock is set as idle low and data is sampled on the rising edge. The `0x03` read data instruction is used to fetch instructions from flash memory. The Execute-In-Place functionality is provided via SSI. It was possible to configure the use of Double-SPI or Quad-SPI for higher-speed memory access. The RP2040 can fetch two or four bits per clock cycle this way. But for the purpose of this project, the standard single SPI is more than enough.

3.1.2 Initialisation

The next process involves writing startup code that is invoked when the microcontroller is reset before jumping to the main function. To do this, a few components need to be understood first; the vector tables that store handlers and the linker script.

```
#include <stdint.h>
#include <stdbool.h>

/* Function pointer type for vector table entries */
typedef void (*vectFunc) (void);

/* External stack pointer symbol defined by the linker script */
extern uint32_t _sstack;

/* Core handler function declarations */
__attribute__((noreturn)) void defaultHandler();
__attribute__((noreturn)) void resetHandler();

/* Core exception handler declarations - weak aliases to defaultHandler */
void nmiHandler       () __attribute__((weak, alias("defaultHandler")));
void hardFaultHandler  () __attribute__((weak, alias("defaultHandler")));
void svCallHandler    () __attribute__((weak, alias("defaultHandler")));
void pendSvHandler    () __attribute__((weak, alias("defaultHandler")));
void sysTickHandler   () __attribute__((weak, alias("defaultHandler")));
void ioIrqBank0       () __attribute__((weak, alias("defaultHandler")));

/* Main program entry point declaration */
extern int main(void);

/* Vector table definition */
const vectFunc vector[] __attribute__((section(".vector"))) =
{
        /* Core System Handler Vectors */
        (vectFunc)(&_sstack),  /* Initial Stack Pointer value */
        resetHandler,          /* Reset Handler */
        nmiHandler,            /* Non-Maskable Interrupt Handler */
        hardFaultHandler,      /* Hard Fault Handler */
        0,                     /* Reserved */
... }
```

Listing 5: `startup.c`

The interrupt vector or trap table is a data structure that contains the addresses of all the exception and interrupt handlers [42]. Per ARM convention, the first element of the vector table is the main stack pointer, and the second element of the vector table is the address of the reset handler [36] and from then on we follow with pointers to various exception handlers like NMI and hard fault. This table is placed at a specific location in flash memory by the linker script, and the VTOR register is configured to point to it during boot. When the RP2040 encounters an exception or interrupt, it automatically looks up the corresponding handler address in this table and jumps to it. For this to work, the microcontroller needs to be aware of the structure of memory. It's the job of our linker script to organize the object files generated by the compiler in accordance with the memory layout for the target microcontroller as specified generally by the RP2040's datasheet with a few notable exceptions; while the implementation utilizes the RP2040's complete SRAM capacity of 264KB, the RP2040 physically organizes this as six independent banks, with four 64KB banks and two 4KB banks. However, we treat it as a single contiguous memory region for what we're doing.

`linker.ld` begins with the ENTRY directive that identifies `bootStage2.c` as the entry point of our program. The MEMORY section defines two key regions:

- FLASH: Starting at address 0x10000000 with 2MB capacity, read-executable only.
- SRAM: Starting at address 0x20000000 with 264KB capacity, read-write-executable.

The SECTIONS directive then organizes our code and data:

- `boot2`: Placed at the beginning of flash memory, enforcing a 256-byte size. This accommodates the stage 2 bootloader and its CRC, which the RP2040 requires in the first 256 bytes of flash. We ensure this by padding to exactly 252 bytes, with the remaining 4 bytes being the CRC.
- `text`: Contains our vector table, code, and read-only data. The vector table is placed first, ensuring it's at a known offset from the beginning of flash.
- `data`: Contains initialised variables that must be in RAM during execution, but their initial values are stored in flash. We use the `AT > flash` directive to instruct the linker to place these values in flash memory immediately after the .text section, while the runtime location (`> sram`) indicates where the data should be copied during startup.
- `bss`: Contains uninitialised variables that must be zeroed during startup.
- `stack`: Placed at the end of RAM, growing downward. This mimics ARM architecture. It starts with a higher address and the value of the stack pointer decreases as the stack grows. The `_sstack` symbol marks the initial stack pointer referenced in the vector table.

It is in RAM that we store our program variables with values that get updated as the program runs. Because RAM is volatile, it must perform this initialisation every time the system boots up. This initialisation is the responsibility of our reset handler.

```
void resetHandler()
{
        extern uint32_t _etext;       /* End of .text section (in flash) */
        extern uint32_t _sdata;       /* Start of .data section (in RAM) */
        extern uint32_t _edata;       /* End of .data section (in RAM) */
        extern uint32_t _sbss;        /* Start of .bss section (in RAM) */
        extern uint32_t _ebss;        /* End of .bss section (in RAM) */

        /* Copy initialized data from flash to RAM */
```

```
        uint32_t *src = &_etext;      /* Source is end of code in flash */
        uint32_t *dst = &_sdata;      /* Destination is start of data in RAM */

        while (dst < &_edata) {
        *dst++ = *src++;
        }

        /* Zero out the BSS section */
        dst = &_sbss;
        while (dst < &_ebss) {
        *dst++ = 0;
        }

        /*  Call the main program */
        main();

        /* If main ever returns, stay in infinite loop */
        while(true);
}

void defaultHandler()
{
        while (true) {
        __asm volatile("wfi");  /* Wait for interrupt - low power state */
        }
}
```

Listing 6: `resetHandler()` and `defaultHandler()`

The reset handler copies relevant sections of data from flash memory into RAM, initialises other sections of RAM, and then executes a platform entry. This platform entry involves branching to the main function in our case as the peripherals have already been set up. Since Cortex-M0+ processors are required to be energy efficient, as their use case is for embedded systems, the ARMv6-M architecture allows us to call `__asm volatile("wfi")` to enable the processor to enter a low power state until the interrupt is triggered from the button press [43].

The default handler also serves as a catch-all for unimplemented exception handlers. All exception handlers that aren't explicitly implemented are aliased to this default handler using the `__attribute__((weak, alias("defaultHandler")))` directive. This attribute is used so that we can declare our own interrupt function that will enable the LED and speaker.

3.1.3 main.c

Our implementation needs a way to interact with external devices. The RP2040 provides 30 GPIO (General Purpose Input/Output) pins that can be configured to flash an LED and sound off a buzzer. These are managed through several hardware register blocks as structs:

- Single-cycle IO: Provides fast, single-cycle access for reading and writing GPIO pins.
- IO Bank 0: Handles GPIO pin function selection and interrupt configuration.
- Pads Bank 0: Controls electrical properties of the pins like pull-up/pull-down resistors.

Our main contains our own defined `delay(uint32_t count)` through looping a `NOP` assembly instruction. We admit the use of this busy–wait loop is computationally ineffective for the CPU. However, it is a simple implementation provided that we otherwise do not have a RTOS setup for this project to provide sleep queues for more efficient behaviour of this kind. Optimisations and critical–time saving methods are not of relevance within this project at this stage.

These blocks are used within our static inline functions `gpio_set_function()`, `gpio_set_dir()`, gpio_set_pulls() and gpio_put():

- `gpio_set_function(uint32_t pin, uint32_t function)`: Configures the function of a GPIO pin by writing directly to the control register for the specified pin. This sets the pin's operating mode using `io->gpio[pin].ctrl = function`.
- `gpio_set_dir(uint32_t pin, bool out)`: Sets a pin as input or output by manipulating the output enable (OE) register in the SIO block. If out is true, it sets the corresponding bit in the `gpio_oe_set` register; otherwise, it clears the bit using the gpio_oe_clr register through `if (out) sio->gpio_oe_set = 1U << pin; else sio->gpio_oe_clr = 1U << pin;`
- `gpio_set_pulls(uint32_t pin, bool up, bool down)`: Configures pull-up/pull-down resistors by setting specific bits in the pad control register for that pin. Bit 3 controls pull-up, bit 2 controls pull-down, and bit 6 enables the input buffer using `pads->gpio[pin] = (up ? (1U << 3) : 0) | (down ? (1U << 2) : 0) | (1U << 6);`
- `gpio_put(uint32_t pin, bool value)`: Sets a pin's output value by using the SIO's atomic set or clear registers, which allow modifying a single pin without affecting others. If value is true, it sets the pin high using `gpio_out_set`; otherwise, it sets the pin low using `gpio_out_clr`: `if (value) { sio->gpio_out_set = 1U << pin; }; else { sio->gpio_out_clr = 1U << pin; };`

We've set the button to trigger an interrupt on a rising edge rather than continuously polling the button state using `if (io->proc0_ints[BUTTON_PIN / 8] & (GPIO_INT_EDGE_HIGH << (4 * (BUTTON_PIN % 8))))`. In this statement:

1. `proc0_ints[BUTTON_PIN / 8]` gets the right interrupt status register.
2. Use `BUTTON_PIN % 8` to get the remainder to find its position within the register.
3. `<< (4 * (BUTTON_PIN % 8))` shifts bits to the right position. Each pin uses 4 bits.

Then our main function runs and lists out the following instructions:

1. Reset the IO Bank 0 peripheral to ensure it starts in a clean state:
```
volatile uint32_t* RESETS_RESET = (volatile uint32_t*)(RESETS_BASE + 0x0);
*resets_reset &= ~(1U << 5);
while ((*resets_reset & (1U << 5)) != 0) {}
```
2. Configure the button (GPIO16) as an input with pull-up resistor.
```
gpio_set_function(BUTTON_PIN, GPIO_FUNC_SIO);
gpio_set_dir(BUTTON_PIN, false);
gpio_set_pulls(BUTTON_PIN, true, false);
```
3. Configure the LED (GPIO25) as an output:
```
gpio_set_function(LED_PIN, GPIO_FUNC_SIO);
gpio_set_dir(LED_PIN, true);
```
4. Configure the speaker (GPIO21) as an output:
```
gpio_set_function(SPEAKER_PIN, GPIO_FUNC_SIO);
gpio_set_dir(SPEAKER_PIN, true);
```
5. Setup button interrupt for rising edge detection:
```
io->intr[BUTTON_PIN / 8] = 0xF << (4 * (BUTTON_PIN % 8));
io->proc0_inte[BUTTON_PIN / 8] |= GPIO_INT_EDGE_HIGH << (4 * (BUTTON_PIN % 8));
NVIC_ISER = 1U << IO_BANK0_IRQ
```
6. Run a startup test pattern to verify functionality.

7. Sleep while waiting for interrupts:
   `while (1) { __asm volatile("wfi");}`

When the button is pressed, the `ioIrqBank0()` interrupt handler activates the LED and speaker, waits for a short period, then deactivates them. This creates a brief visual and audible feedback when the button is pressed. The handler also clears the interrupt flag to acknowledge the event.

3.1.4 Libraries

The project depends on two external libraries which are incorporated as subdirectories:

- `CRCpp`: Used to calculate CRC32 checksum for the bootloader section.
- `UF2`: Required to convert binary to UF2 format for flashing.

To flash the main file, we need to build the project. The Makefile contains definitions to configure the compiler and linker to use the correct CPU target and standard library settings. It also describes the steps that must be taken to generate the final binaries. Its benefit is that it documents the process of building firmware in its entirety.

The compiler flags configure the compiler with these settings:

- `mcpu=cortex-m0plus`: Specifies the Cortex–M0+ processor target which is used in the RP2040.
- `-g`: Includes debug information for debugging on the host machine.

The linker flags specify `nostdlib` to avoid linking against the standard C library since we're implementing our own minimal runtime.

Several key targets are defined in the Makefile:

1. The bootStage2 target compiles our second-stage bootloader and generates an object dump for inspection.
2. Individual object files are compiled from C source files.
3. The CRC calculation generates a C file containing the CRC checksum required by the RP2040 bootloader.
4. The final ELF file links all object files with the bootloader and CRC.
5. The UF2 file is generated using Microsoft's UF2 utility for easy deployment to the RP2040 via USB mass storage.

The UF2 converter sets the following parameters:

- `-b`: 0x10000000 as the XIP flash base address where the code will be loaded.
- `-f`: 0xe48bff56 as the identifier for the Raspberry Pi Pico.
- `-c`: The input binary file.
- `-o`: The output UF2 file.

Now we run `make` or `mingw32-make`. Once built, holding down the BOOTSEL button on the RP2040 and inserting the USB to our machine will allow us to transfer the output `uf2` file to our mass storage device. When the button is pressed, the LED and speaker will be registered as a HIGH with an input of 3.3V and toggle on.

3.2 Using the Rust Language

The implementation follows the same principles as the C version. Despite this, our final `uf2` image does not work when loaded, and only operates under a debug session in OpenOCD. While working on the project, this issue was narrowed down to three possibilities; there's an underlying issue with the linker, the CRC and consequently the second stage checksum fails, the boot ROM can't find our .boot2 code or the `elfuf2` crate version we were running had problems.

We begin by installing Cargo, Rust's package manager using:

```
> curl -sSf https://static.rust-lang.org/rustup.sh | sh
```

Listing 7: Installing Cargo

This gets us to the current stable release of Rust for our platform along with the latest Cargo and saves us the time of building from source. The project uses Cargo 1.87.0-Nightly.

3.2.1 Boot-Sequence

The implementation for booting remains similar with only minor adjustments following the porting of our code from C to Rust:

```
#[link_section = ".boot2"]
#[no_mangle]
pub unsafe extern "C" fn bootStage2() → ! {
        // Disable SSI to configure it
        ptr::write_volatile(SSI_SSIENR, 0);
        // Set clock divider
        ptr::write_volatile(SSI_BAUDR, 4);
                  ...
       // Set the stack pointer
        core::arch::asm!("msr MSP, {0}", in(reg) stack_pointer);
                  ...
        // Ensure we're actually going to jump to a valid location in flash
        if reset_handler ≥ XIP_BASE && reset_handler < (XIP_BASE + 0×1000000) {
        // Jump to reset handler
        core::arch::asm!("bx {0}", in(reg) reset_handler, options(noreturn));
        }

        // If we get here, something went wrong - enter default handler behavior
        loop {
           core::arch::asm!("wfi");
        }
}
```

Listing 8: `bootStage2()`

Attributes exist in Rust with the `bootStage2()` function being marked with `#[link_section = ".boot2"]` and `#[no_mangle]`. `#[no_mangle]` is persistent throughout the codebase to prevent our functions and static variables from being mangled when compiled. RFC 2603 defines mangling as to prevent collisions during compile–time among other reasons [44]. Despite this being unsafe, we tell the compiler to turn this off so that it is easier to link our resultant `elf`. Here, `bootstage2()` is cast as a diverging function, marked with `→ !`, implying this function never returns. By looking at the RP2040 datasheet to look at the different bits of `CTRLR0` we can figure out what should go in each register. A bit lower down, we use assembly code.

This is necessary when setting the stack pointer and jumping to the reset handler since both Rust and C don't contain any direct functions to access ARM special purpose registers.

In Rust, there is no volatile modifier like in C. Instead, unsafe volatile read and write primitives were used. When performing write operations to memory-mapped peripheral addresses, we use `ptr::write_volatile`. This way, the compiler knows to not reorder our values or eliminate duplicate accesses to the same register for performance reasons. All values set are self-explanatory, though the value for the clock divider is significant here. The RP2040 datasheet mentions that the frequency of the SPI CLK signal is the function of f_{SSI_CLK} and the 16-bit SCKDV value in the BAUDR register [45]. All it takes is for us to know the desired SPI CLK frequency, found in the flash memory datasheet, stating that the maximum SPI clock frequency is 33 MHz for the Read Data Instruction [46]. Therefore:

$$\text{SCKDV} = \frac{f_{SSI\_CLK}}{f_{SCLK\_OUT}} = \frac{125\ MHz}{33\ MHz} = 3.7878 \approx 4$$

By default, our RP2040 boots up with the clock signal from the Ring Oscillator operating at 6.5 MHz [47]. While not being as accurate as other sources such as the Crystal Oscillator or Relaxation Oscillators since it can change due to conditions like voltage and temperature, the objective is to remain minimal for our implementation to still work. ROSC is still efficient enough for us at this point. Using GDB, we can see our boot code jump and reach line 13 in Listing 7 successfully:

```
(gdb) break resetHandler
Breakpoint 1 at 0×10000288
(gdb) continue
Continuing.

Thread 1 "rp2040.core0" hit Breakpoint 1, 0×10000288 in resetHandler ()
```

Listing 9: Debugging `boot2Stage.rs`

3.2.2 Initialisation

The startup code had differences that arose due to compile-time errors, notably with the vector table. Its implementation was replaced multiple times, including with the use of union structures and raw pointers to allow for compilation. Listing 10 in the appendix shows some of the different implementation types the table went through, yielding the outputs listed.

It's clear that attempting to port C-style code isn't so straight-forward, and inherent unsafe code continues to persist. Firsty, allowing arbitrary pointer-to-integer casts in constants leads to undefined behavior because function addresses aren't known at compile time, they're determined at link time [70]. In other words, when the program is running and executing. The borrow checker and type system can no longer provide guarantees and a work around is needed. Using `core::ptr::addr_of!` to create a raw pointer without creating an intermediate reference that could trigger Rust's aliasing rules is attempted [48]. Raw pointers can ignore the borrowing rules by having both immutable and mutable pointers or multiple mutable pointers to the same location, don't guarantee a valid pointer to memory, can be null and don't implement cleanup. This way, guaranteed safety is given up in exchange for the ability to interface with C-style code. But an error is thrown; temporaries can't have their address taken since they don't have a stable memory location [49] [50]. Despite being unsafe, an attempt to use unions brings the third and final error. The compiler complains that raw pointers cannot be safely shared between

threads as it may lead to two or more references being able to modify the same memory location at the same time, leading to data races, corruption or undefined behavior. While compatible in C, the ownership system prevents this at compile time and we are told to implement `Sync`, Rust's way of enforcing thread-safe shared references `&T` without relying on mutexes, locks or atomic types. It guarantees that if a type is `Sync`, we can safely share immutable references between threads without causing data races, as long as we ourselves can guarantee the vector table is read-only after initialisation and single-threaded [51]. In this manner, we are left with our final table:

```
#[repr(C)]
#[derive(Copy, Clone)]
union VectorTableEntry {
        handler: unsafe extern "C" fn() -> !,
        reserved: u32,
        stack_top: *const u32,
}

unsafe impl Sync for VectorTableEntry {}

#[link_section = ".vector_table"]
#[no_mangle]
pub static VECTOR_TABLE: [VectorTableEntry; 48] = [
        // Initial Stack Pointer
        VectorTableEntry { stack_top: unsafe { ptr::addr_of!(_sstack) } },

        // Core exception handlers
        VectorTableEntry { handler: resetHandler },
... ];
```

Listing 11: `VectorTableEntry` and `VECTOR_TABLE` with `Sync`

Here we explicitly mark our vector table with 48 entries. ARM reserves 16 of these entries and 32 are our external handlers [52]. This is slightly different to what we did in C, where our vector table only initialised the bare minimum. Our union is designed to hold one of three possible types of data at the same memory location: an interrupt handler function pointer, a reserved 32-bit integer, or a pointer to the top of the stack. Since all fields share memory, only the field used during initialization contains a meaningful value for that specific entry. To facilitate the use of these entries within the vector table, we use `#[derive(Copy, Clone)]`. Clone is a trait that allows us to explicitly create a duplicate of our table. Rust's compiler needs to place each of these VectorTableEntry values into the memory allocated for the static array and implementing the `Copy` trait makes this process trivial. Now, it gets duplicated by a simple bitwise copy of its memory and provides guarantees for us. The compiler can determine the exact bit pattern for each entry and lay them out sequentially in the static memory region. Without `Copy` our union is considered a "move-only" type. This implies that when we use a value, ownership is transferred [53].

The compiler helped us manage to reduce the amount of unsafe code blocks used significantly. We begin to understand the philosophy of the language; the goal is to keep nudging the programmer to follow its rules for the benefit of the program. Now, we only have our union marked as unsafe. Unions are inherently unsafe as Rust can't guarantee the type of the data currently being stored in the union instance. Here, the `#[repr(C)]` attribute ensures that we have the same size and alignment as an equivalent C union declaration in the C language for our target platform as Rust does not provide any guarantees on this otherwise [53]. However, it's clear to discern that in–experience with Rust can result in repeated use of unsafe code blocks in–order to satisfy compilation, something that should be avoided at all costs in general. The

migration from C can cause us to stumble from bad habits that we are otherwise hoping to avoid.

3.2.3 main.rs

Functionally, main remains consistent through the migration. To attempt to reduce bloat in the main function, our first implementation defined helper functions `gpio_put()`, `gpio_set_pull()` and `gpio_set_dir()`. This failed and to simplify our development, the migration was made direct and replicated what we had done in C. Before moving on with the issues with the `uf2`, the main function and its associated function calls to the handlers went through debugging. We use GDB and input the following:

```
(gdb) break ioIrqBank0
Breakpoint 1 at 0×10000358
(gdb) continue
Continuing.

Thread 1 "rp2040.core0" hit Breakpoint 2, lib::transmit::main () at src/main.rs:138
138  *resets_reset &= !(1u32 << 5);
```

Listing 12: Debugging `main.rs`

Stepping through, we can observe the LED and buzzer turn on at line 174:

```
168  *NVIC_ISER = 1u32 << IO_BANK0_IRQ;  /* Enable interrupt in NVIC */
(gdb) step
173  (*sio()).gpio_out_set = (1u32 << LED_PIN) | (1u32 << SPEAKER_PIN);
(gdb) step
174  delay(100000);
```

Listing 13: LED and Buzzer Enable

Our first breakpoint was enabled at `ioIrqBank0` in order to test the button press. While GDB is continuing our debug session and pressing down on the button, we observe the following:

```
(gdb) continue
Continuing.

Thread 1 "rp2040.core0" hit Breakpoint 1, lib::transmit::ioIrqBank0 () at src/main.rs:113
113 if (*io()).proc0_ints[pin_index as usize] & (GPIO_INT_EDGE_HIGH << pin_offset) ≠ 0 {

(gdb) bt
#0  lib::transmit::ioIrqBank0 () at src/main.rs:113
#1  0×100002e8 in ioIrqBank0Handler ()
#2  <signal handler called>
#3  0×100003fc in lib::transmit::main () at src/transmit.rs:189
```

Listing 14: `ioIrqBank0` Signal Handler Call

Backtracing the breakpoint shows us the call stack for `ioIrqBank0`. All of this confirms our Rust program is functional through a remote debug session.

3.2.4 lib.rs

In Rust we define our project structure with a `lib.rs` file. This file functions similarly to a header file in C, exposing and organizing the various modules of our project:

```rust
#![no_std]
#![no_main]
#![feature(linkage)]

use core::panic::PanicInfo;

#[cfg(not(any(feature = "boot2", feature = "startup", feature = "main")))]
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
        loop {}
}
```

Listing 15: `lib.rs`

The `#![no_std]` attribute informs the compiler that our project won't be using the standard library, which is necessary for bare-metal development on the RP2040. Similarly, `#![no_main]` indicates we don't use the standard entry point and instead define our own. `#![feature(linkage)]` enables unstable features of Rust's linker, which we need for our custom linking process. Here is where we also define our panic handler, our function that acts upon an unrecoverable error. It allows our program to terminate immediately and provide feedback. The `#[cfg()]` attributes are Rust's conditional compilation directives, similar to C's `#ifdef`. Each feature enables a different module of our project.

3.2.5 Crates

Cargo, Rust's package manager, uses "crates" to manage dependencies. Our `Cargo.toml` file defines the project configuration and dependencies. It contains three main dependencies, and one build dependency:

- `cortex-m`: For low-level access to the Cortex-M processors,
- `cortex-m-rt`: For the runtime components needed for the ARM Cortex-M processors,
- `panic-halt`: For our panic handler,
- `crc`: For calculating the CRC checksum.

For both release and development profiles, we use `"panic = "abort"` to ensure the program halts on panic rather than attempting to unwind the stack. We explicitly disabled Link Time Optimisation (LTO) and set `debug = true` for both profiles to allow for debugging symbols to be present when using GDB. Optimisations are kept minimal using `opt-level = 0` to prevent the compiler from optimising anything out during development.

3.3 Using the SDK

Here we aim to simulate a more realistic embedded development environment with the Pico SDK. A full, fair language comparison could not have been made otherwise. The code used within the section is referenced in Appendix A.

We begin by running `git clone https://github.com/raspberrypi/pico-sdk.git`, updating all nested submodules within the repository and placing it somewhere within our computer's file system. To make use of the SDK in our project, the `pico_sdk_import.cmake` file is dragged and dropped into the root of our directory tree. Within a bare-metal environment, we were constrained to developing our own Makefile. This was painfully long, and the syntax of Make was not on our side. CMake is a meta-build system that will automatically generate the

Makefile, pull in the relevant linked libraries and SDK, define preprocessor macros for each executable and set up our directories for execution. Our final implementation worked with three key components; a `main.h` file and our `transmitter.c` and `receiver.c` files that were separately loaded into the two Picos.

3.3.1 transmitter.c

The transmitter module manages the transmission of morse code, initialising and configuring the `BUTTON_PIN` through GPIO16, the LED through GPIO 25 as `PICO_DEFAULT_LED_PIN` and `SPEAKER_PIN` through GPIO21 using `gpio_init()` and `gpio_set_dir()`. Dots are recognized as push-downs on the button that last 250 ms and dashes are for 750 ms durations. The concept is simple; using busy-wait loops, if the button is pressed for these given intervals with some debounce, we transmit a dot or dash by enabling the buzzer, the LED and passing the message through UART. They generate different frequencies, 800 Hz and 400 Hz respectively to provide audible differentiation. At some point during development, issues with matching the synchronisation behind the transmitting Pico and receiving Pico arose. At the beginning of initialisation, we send over a distinct morse code with the LED to calibrate the timing thresholds automatically and confirm both systems are ready.

The system uses a state machine architecture to manage the input detection and transmission logic; idle, button pressed, dot transmission, dash transmission, and gap detection. The transmitter implements three distinct types of gaps—intra-character (150 ms), inter-character and word gaps which allows for proper parsing of the morse code sequences on the receiver end.

3.3.2 receiver.c

Originally, the receiver centered around picking up the dots and dashes through differentiating distinct HIGH values over a given threshold through the `ADC_PIN` on GPIO26. This worsened the aforementioned synchronisation delays between the transmitter and receiver while making it more complicated than it should have been to discern between dots and dashes. It was inconsistent and not to mention the random noise being observed through the serial port. Instead, both the Picos were made to communicate through UART using `UART_TX_PIN` and `UART_RX_PIN`. The receiver centers around adaptive signal processing, timing through interrupt-driven edge detection and tracking of inter-symbol gaps and decoding dots and dashes, or word boundaries with a time of 1750 ms. The final responsibility of the receiver is to output the decoded morse data, in real–time, to a I^2C LCD1602 module.

Similar to the transmitter, the receiver implements a state machine with three primary states: idle, in-character, and in-word. This approach effectively handles the complexity of morse decoding by maintaining context awareness throughout the communication process. Character recognition utilizes a dictionary-based lookup system that maps the received morse patterns to their corresponding alphanumeric characters. For resilience against timing inconsistencies, the system implements dual decoding mechanisms; explicit gap markers received through UART ('C' for character gaps and 'W' for word gaps) and automatic timeout-based detection.

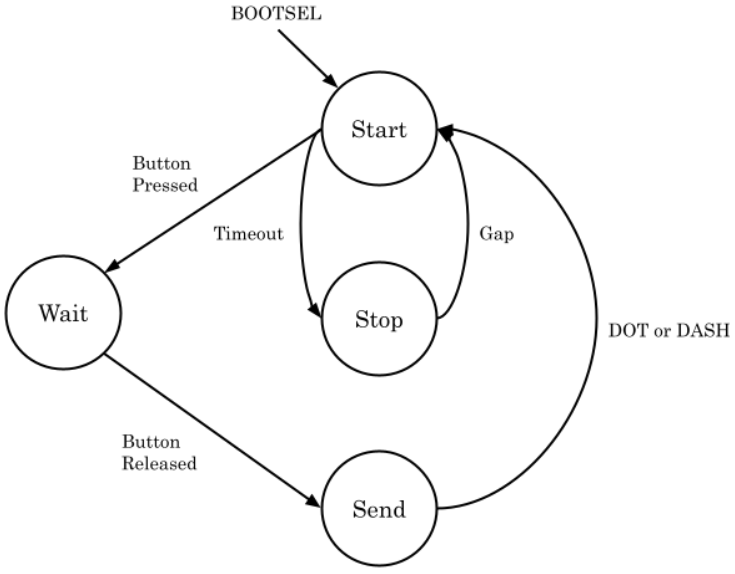

Figure 5: Transmitter State Machine

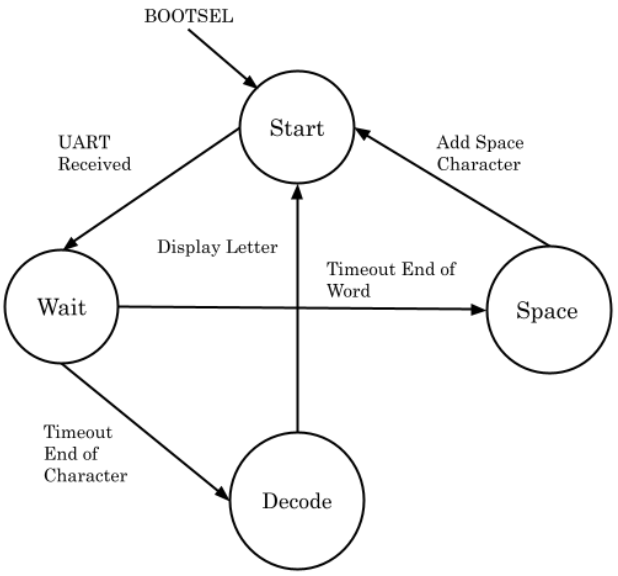

Figure 6: Receiver State Machine

3.3.3 Testing and Validation

To test and validate both the transmitter and receiver, we use the integrated serial monitor within VSCode to watch both Picos communicated over UART with their exchanges being logged through the serial ports via USB.

COM3 – USB Serial Port	Baud Rate 115200	COM4 – USB Serial Port	Baud Rate 115200

```
─── Reopened serial port COM3 ───
UART Initialized
Transmitter initialized.
Starting morse code transmission
Sent: Starting morse transmission ...
Ready for input
Sent: Transmitting sync pattern ...
                ...
Sent: Sync pattern transmitted
Sent: .
Sent: -
```

```
─── Reopened serial port COM4 ───
Starting morse code receiver
Initializing I2C for LCD ...
Scanning I2C bus for devices ...
Using SDA_PIN: 4, SCL_PIN: 5
I2C device found at address 0×27
Found LCD at expected address 0×27
LCD initialization complete
LCD successfully initialized
Receiver initialized and ready
Starting morse reception ...
Received signal: .
Decoded: E
LCD Display: E
Decoded character: E (.)
Received signal: -
Decoded: T
LCD Display: T
Decoded character: T (-)
```

Listing 16: Testing `transmitter.c` and `receiver.c`

There's just one issue imposed when using the SDK, and it comes due to performance overhead. The button press is slow unlike when the project was operating in a bare-metal environment, making it difficult to transmit a letter containing a longer string of morse. The issue persists and it highlights the issue with using a HAL for something like a safety critical system, like two computers communicating in real-time over some encrypted channel. In this case, it is too easy to send off the letters E or T as they only require single or double morse characters without the trade off of longer delays, a feature that communication of this sort can't depend on constantly. Our LCD module had issues with circuitry during implementation with regards to its connection

to VCC. Many references stated that its positive supply voltage should be set to 5V, correlating to a connection to pin 40 on the Pico. When this was attempted for the first time, the backlight kept flashing without being able to remain consistently on [54]. Research showed that the panel must be driven at 3.3V instead [55].

3.4 Using the HAL

For the rust implementation we take advantage of `rp-hal`, high-level drivers for the Raspberry Silicon RP2040 microcontroller and its associated boards [85]. This repository is cloned to the root of our project directory. From it, we get access to a provided linker script, `memory.x` that we are required to implement. It also provides proven and tested examples of code. Features from the `embedded-hal` crate are also used to make our code work.

The project root directory contained the essential configuration files: `Cargo.toml` for dependency management and project metadata and `.cargo/config.toml` for build configurations specific to the ARM Cortex-M target. At the beginning, build issues were encountered– a `/src/bin` had not been implemented and had been all placed in `src.` Unlike our bare-metal approach, we organized the subdirectories to maintain separation between different components. The `lib.rs` file served as the central module that exported our shared functionality, while the `bin` directory contained multiple executable targets for different aspects of our project.

3.4.1 transmitter.rs

The core of the transmitter is implemented as a struct with `pub struct Transmitter`, encapsulating all necessary hardware interfaces and state management. Our C code ended up being less lines of code compared to our Rust code. This is because `transmitter.rs` went under a lot of testing and additions during the migration attempt, such as manual PAC watchdog and clock initialisation. We also had to explicitly define our bootloader in each file using `pub static BOOT2: [u8; 256] = rp2040_boot2::BOOT_LOADER_GENERIC_03H` [58].

Many key design decisions remain consistent following the migration, though they were met with difficulties unlike when implementing the C code. Unsurprisingly, using the HAL compared to the Pico SDK was much more difficult as being unfamiliar with declaration of mutable variables and Rust's types and parameter rules. For example, we mistakenly use `<PushPull>` as a concrete type– a fully specified type that the compiler knows exactly how to represent in memory like `i32`. Unlike C, Rust supports generics that prevent type duplication [59]. This way, Rust encodes the pin configuration in the type system. This means the compiler can verify that we're using pins correctly. We can't accidentally read from an output pin or write to an input pin.

```
error[E0412]: cannot find type `Output` in this scope
  ⟶ src\transmitter.rs:46:30
   |
46 |           led_pin: Pin<Gpio25, Output<PushPull>, PullUp>,
   |                            ^^^^^^ not found in this scope
   |
help: there is an enum variant `rp2040_hal::gpio::DynSioConfig::Output` and 1 other; try
using the variant's enum
```

```
error[E0412]: cannot find type `PushPull` in this scope
  ⟶ src\receiver.rs:46:37
   |
```

```
46 |           led_pin: Pin<Gpio25, Output<PushPull>, PullUp>,
   |                                 ^^^^^^^^ not found in this scope
   |
help: you might be missing a type parameter
   |
37 | impl<UART, I2cBus, P, PushPull> Receiver<UART, I2cBus, P>
   |                   ++++++++++
```

Listing 17: `error[E0412]`

Thankfully, Rust's compiler was extremely useful with its help messages when it came to managing these issues. Many of them boiled down to improper package definitions and mismanagement of the project structure rather than suffering with the features of the language or the implementation itself.

3.4.2 receiver.rs

Like `transmitter.rs`, the receiver and its global variables are encapsulated within a struct with each method operating only on its own internal state. Our morse decoding algorithm uses fixed-size buffers with compile-time capacity validation through Rust's `heapless` crate which provides fixed-capacity implementations of standard collections like `String` and `Vec` that do not require dynamic memory allocation. The rest of the migration was straight-forward. During the receivers implementation, we encountered the first run-in with the borrow-checker:

```
error[E0502]: cannot borrow `*self` as mutable because it is also borrowed as immutable
   ⟶ src\receiver.rs:221:9
       |
221 |           self.lcd_print(&self.display_message[start_pos..]);
       |           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
       |           |       |       |
       |           |       |       immutable borrow occurs here
       |           |       immutable borrow later used by call
       |           mutable borrow occurs here
```

Listing 18: `error[E0502]`

This error occurred because our `lcd_print()` function requires a mutable reference to `self` while simultaneously trying to borrow `self.display_message` immutably. The solution was to use splitting borrows, a method that allows separate handling of different struct fields by creating an intermediate string.

3.4.3 Testing and Validation

Before testing our work, the project had to be built. This is where the implementation began to encounter issues with linking. We had defined a linker script taken from `rp-hal` yet encountered errors with it:

```
= note: some arguments are omitted. use `--verbose` to show all linker arguments
= note: rust-lld: error: section '.boot2' will not fit in region 'BOOT2': overflowed by
256 bytes
rust-lld: error: section .boot2 virtual address range overlaps with .vector_table
        >>> .boot2 range is [0×10000000, 0×100001FF]
        >>> .vector_table range is [0×10000100, 0×100001BF]

rust-lld: error: section .boot2 load address range overlaps with .vector_table
        >>> .boot2 range is [0×10000000, 0×100001FF]
        >>> .vector_table range is [0×10000100, 0×100001BF]

error: could not compile `morse_rsdk` (bin "transmitter") due to 1 previous error
```

Listing 19: `rust-lld` Error

Within the MEMORY section our second stage boot code existed at `0×10000000` with a length of 0x100 bytes, with flash being offset by `0×200` from the second stage and length `2048K - 0×100`. This issue occurred due to being under the assumption that the Pico we'd be operating on had 2048 KB of flash. To fix this issue the length of the second stage boot code was increased to 0x300 and the length of flash was updated to `2048K - 0×300`. Additionally, an attempt to use the `probe-rs run --chip RP2040` which we defined as our runner was met with a persistent error:

```
> cargo run --release --bin transmitter
                                        ...
WARN probe_rs::architecture::arm::core::armv6m: The core is in locked up status as a
result of an unrecoverable exception
Erasing ✓ 100% [####################]   8.00 KiB @  37.16 KiB/s (took 0s)
Programming ✓ 100% [####################]   8.00 KiB @  32.37 KiB/s (took 0s)
Finished in 0.48s
Error: The core is locked up.
error: process didn't exit successfully: `probe-rs run --chip RP2040
target\thumbv6m-none-eabi\release\transmitter` (exit code: 1)
```

Listing 20: probe-rs ARM Core Lock Error

Trying to diagnose this error was difficult. On page 207, The ARM ® v6-M Architecture Reference Manual states that a processor can exit Lockup state in the following ways [60]:

- If locked up at priority –1 and an NMI exception occurs, the NMI is activated as normal. The NMI return link is the address used for the Lockup state.
- A System reset occurs. This exits Lockup state and resets the system as normal."
- A halt command from a halt mode debug agent is issued.

Sometimes a system reset would occur where our runner would work but then on a second run, lock the core in a similar manner. The issue either arises due to our transmitter code through an invalid clock setup, hard fault, or infinite loop despite not being picked up at compile-time–an unlikely situation or through a hardware issue. Double checking on OpenOCD rules this out. Attempting to force a clean reset using `probe-rs run --connect-under-reset --chip RP2040` gives us our final warning and error:

```
> probe-rs run --connect-under-reset --chip RP2040
target/thumbv6m-none-eabi/release/transmitter

WARN probe_rs::session: Timeout while deasserting hardware reset pin. This indicates that
the reset pin is not properly connected. Please check your hardware setup.
```

```
Error: Connecting to the chip was unsuccessful.
Caused by: A timeout occurred.
```

Listing 21: `probe-rs` Timeout Error

No viable fix could be found. Furthermore, attempting to load both `transmitter.rs` and `receiver.rs` as `uf2` images with the `elf2uf2` didn't seem to work either. Suspecting this to be a bug within `probe-rs`, OpenOCD was used for testing instead.

```
(gdb) monitor arm semihosting enable
semihosting is enabled
(gdb) monitor arm semihosting_fileio enable
semihosting fileio is enabled
(gdb) break main
(gdb) continue
Continuing.
Transmitter initialized
Starting morse transmission...
Ready for input
Transmitting sync pattern...
            ...
Sync pattern transmitted
```

```
(gdb) continue
Continuing.
(gdb) info threads
Id   Target Id
Frame
* 1  Thread 1 "rp2040.core0" (Name:
rp2040.core0, state: debug-request)
0x10008db4 in
rp2040_hal::i2c::I2C<rp2040_pac::I2C0..
```

Listing 22: Testing `transmitter.rs` and `receiver.rs`

We verify that our transmitter code works as intended. The sync pattern is transmitted with the dots and dashes clearly being discerned. Pressing the button during run-time enables the LED, buzzer and outputs a dot or dash depending on the interval the push down is held for. Attempting to validate our receiver is trickier. Firstly, we only have two Picos with one acting as the debugger and the other loaded with the receiver code. This leaves the COM3 serial busy being used by GDB while COM4 failed to initialise. Instead, we inspect the processes and the registers to validate the jobs being assigned to threads during execution. Listing 21 shows core 0 establishing a worker thread to set up our I²C controller for the LCD1602. It goes on to initialise our GPIOs and UART peripherals before trampolining back to our main function. By setting a breakpoint at `receiver.init();` we witness our LED enable, though the LCD display remains blank. The program gets to the point of communicating with the LCD, but fails to receive a response back. The full debug output is cited in Appendix C. Watching the call stack from this breakpoint shows us the aforementioned processes taken by our program:

1. An I²C transmission check from the HAL. Determining whether the FIFO buffer is full before moving on via `I2C::tx_fifo_full()`.
2. Since the FIFO buffer isn't full, a write instruction to the controller using `I2C::write()` is sent.
3. Our programming calls the write byte `fn` to the LCD via `Receiver::lcd_write_byte()`.
4. Enable pulse, send and print our data to the LCD.

No bytes are being specified for the write calls because our UART is waiting for data to be sent. With no access to a terminal for this input, our program is left waiting. Despite this we can confirm initialisation occurs but not as intended. We are unable to conclusively prove full LCD functionality, morse decoding and the final characters being displayed.

4 Methodology

This chapter details the systematic approach employed to compare the C and Rust programming languages for embedded systems development on the Raspberry Pi Pico. It outlines the research design, the strategy used to ensure comparable implementations, the specific procedures for data collection and measurement across various metrics, the tools and environment utilised, and the analytical approach taken to address the research questions posed in Chapter 1.

4.1 Version Control

GitHub was used as the primary version control software. The source code for this project can be cloned by typing the following into the terminal:

```
> git clone https://github.com/TremorPulse/morse-rp2040
> git clone https://github.com/TremorPulse/rp2040-bare_metal
```

Listing 23: Cloning the Project

4.2 Research Criteria

The core of this research employs a comparative experimental design. The primary goal is to evaluate the suitability of Rust as a suited replacement to C for embedded systems development by implementing the same target application in both languages. The project targets a constrained microcontroller. To assess the impact of abstraction levels, two distinct implementations were undertaken for each language. The first was through a bare-metal environment, focusing on direct hardware register manipulation with minimal external libraries. This involved writing custom startup code, vector tables, and peripheral control code. The second was using the Pico SDK and Rust HALs, higher-level abstractions to simplify development and perform data collection.

Starting off on the lowest level with no abstraction not only aids personal development objectives, but will help to highlight the differences of the programming languages in Chapter 5 that might otherwise be difficult using just the SDK and HALs alone. The same can be said for using higher level abstractions– this way, both unsafe and safe Rust can be evaluated. The development process itself, including challenges and language feature usage, was documented qualitatively in Chapter 3.

4.3 Research Considerations

Our research used a selection of tools for the project and was primarily dictated by the standard development environments for C and Rust in the embedded ARM context. We clarified this in Chapter 3.

For C development, the GNU Arm Embedded Toolchain was used. This toolchain is the industry standard for compiling C code for ARM microcontrollers, particularly for "bare-metal" environments or those using lightweight RTOSs. The `arm-none-eabi` target triplet signifies that

the generated code assumes no underlying OS and follows the ARM (EABI). Using this standard toolchain ensures access to mature, well-documented tools specifically designed for cross-compilation from a host PC, in this case, an x86 Windows to the ARM target. First of all, our RP2040 has limited processing power and memory. Running a full compiler on this device is not possible and unrealistic when researching with the focus on constrained embedded systems. The Pico is a bare-metal device and our host machine operates on Windows, so this method allows us to generate code for our specific target. In Rust, this was with `thumbv6m-none-eabi`.

Building for the C SDK project utilized CMake along with the Pico SDK's provided CMake modules through `pico_sdk_import.cmake`. CMake is the official build system generator for the Pico SDK; its use was necessary to correctly integrate the various SDK libraries and platform configurations automatically. For the C bare-metal project, a manual Makefile was employed. This approach was chosen for its explicit control it offers over the compilation and linking process, which is often preferred for bare-metal development where dependencies are minimal and direct control over our custom linker script was required. While alternatives like Ninja exist, It was the build style I was most familiar with from other C projects I've done.

For Rust development, the standard Cargo build system and package manager was used alongside the rustc compiler. Both are standard in Rust. Cargo is central to the Rust ecosystem and significantly simplifies embedded development compared to manual C toolchain management. It handles target specification. In our case, `thumbv6m-none-eabi` for the Cortex-M0+ dependency resolutions and fetching from the central crates.io repository alongside the `rp2040-hal`, `cortex-m-rt`, and `embedded-hal` crates discussed in Chapter 3. The cargo-binutils extension was installed to provide convenient cargo subcommands (cargo size, cargo nm, cargo readobj) that wrap the underlying LLVM tools (llvm-size, llvm-nm, llvm-readobj) used by rustc. This integration provides analysis capabilities comparable to the GNU binutils directly within the cargo workflow.

4.4 Control Variables

To ensure a fair comparison, several factors were controlled across the implementations. Firstly, all implementations targeted the ARM Cortex-M0+ on the Raspberry Pi Pico. The same hardware setup, including button inputs, LED/buzzer outputs, UART communication links, and an I2C LCD display, was used for functionally equivalent C and Rust SDK/HAL implementations. For the project, the fundamental requirements of the morse code application including button input detection, dot/dash timing, tone generation, UART transmission, reception/decoding logic and LCD output were kept consistent as best as possible across both C and Rust SDK/HAL versions. The bare-metal versions implemented a simpler subset with a button input to LED/buzzer output but maintained functional equivalence. Our development environment used the same host machine on Windows and debugging hardware setup was used through a debug Pico probe. When performing performance-related measurements, comparable optimisation levels were targeted during compilation: typically release/optimised builds with C using `-O3` and `-O0` and Rust using `opt-level = 0`, `opt-level = 3` and `opt-level = "z"` for Rust release profiles. Our hardware-related measurements in C and Rust both underwent tests for the same interfaces: ADC, GPIO, Interrupt, PWM and UART. Attempting to control the layout of memory in the SDK/HAL implementations was more tricky and failed. CMake automatically linked against the linker provided in the SDK. However, executing the Rust code necessitated the use of a custom linker script. However, two separate link scripts had to be provided when running `cargo build --bin benchmarks` compared to `cargo build --bin transmitter` and `cargo build --bin receiver`. Attempting to build `transmitter.rs` and

`receiver.rs` with the linker file for `benchmarks.rs` results in `.boot2` overlapping with the vector table and attempting to build `benchmarks.rs` with the linker file for `transmitter.rs` and `receiver.rs` results in an ARM related error occurring when running `probe-rs run --chip RP2040 target/thumbv6m-none-eabi/debug/benchmarks`. As a result, the final zip file containing the code required to run the project has `memory.x` for linking the transmitter and receiver code and `memory2.x` for linking the benchmark code.

The independent variables were the programming language and the level of abstractions used throughout the project.

4.5 Validation and Practices

Equivalence was validated through functional testing. For the bare-metal blink/buzzer application, visual and audible confirmation upon button press was used. For the SDK/HAL morse code application, successful transmission and reception of morse characters via UART, correct decoding, and display on the LCD served as validation.

Implementations aimed to follow conventional practices for each language. The C code used standard SDK functions or direct register access typical in C embedded development. The Rust code leveraged HAL traits, the ownership model, Result for error handling, and crates from the embedded ecosystem where appropriate, while resorting to unsafe for necessary low-level operations in the bare-metal version.

4.6 Data Collection

Data collection involved both static analysis of code artifacts and dynamic runtime measurements on the target hardware. We intend to determine how well Rust delivers on its guarantees in an embedded environment. In-order to assess these features, our migrated morse encoding project serves as the example program to demonstrate these metrics. From here on out we aim to explain how the measurements were performed then present the findings and discuss results in our evaluation, including our failures as remarked with tools such as Miri– we lack an OS to simulate our hardware and find it inconclusive to analyse memory during program execution. Instead, our language comparison intends to demonstrate the features present in Rust to achieve this guarantee.

During testing we determined the project wasn't intensive enough to measure processor cycle counts as a standalone metric for this investigation. Furthermore, the stack was dynamically allocated at run–time within our project hence profiling was made difficult as it wasn't visible in the binary while working on the project. Due to time constraints of the project, profiling the stack was left out. This project also does not assess compilation times; we find it as a lesser priority that doesn't meet our research criteria. It'd also be made difficult as controlling the amount of libraries used on both ends to simulate a fair analysis would have added complexity.

4.6.1 Static Artifacts

Static analysis involved examining the output files generated by the compilation and linking process, primarily the executable ELF, without running the code on the microcontroller. This method allows for objective comparisons of binary size, static memory allocation, and compilation overhead, independent of runtime conditions.

The following metrics were collected both for the SDK/HAL and bare–metal implementations:

1. Static RAM usage,
2. Flash memory size,
3. Largest functions and static variables.

To measure our static metrics, we define a `.ps` file with the responsibility of building and inspecting the metadata of the final executable binary produced. It does this by parsing the output of our `arm-none-eabi` commands using `Select-String` and regular expressions to find the lines corresponding to the `.text`, `.rodata`, `.data`, and `.bss` sections and extract their respective sizes in bytes. Flash memory was calculated as the sum of the sizes of the sections that are loaded into the microcontroller's flash memory. Specifically, our powershell script calculates it using `$rxFlashTotal = $textSize + $rodataSize + $dataSize`. These variables are found by running `arm-none-eabi-size`. While `.data` variables are used in RAM at runtime, their initial values must be stored in the flash image and copied to RAM during program startup. Therefore, it contributes to both flash footprint and initial RAM usage. The static RAM usage represents the amount of RAM allocated at program startup before any dynamic allocation occurs through the stack. It was calculated using `$rxRamTotal = $dataSize + $bssSize.` The `.bss` section requires space in RAM at runtime, but since its contents are all zeros, it doesn't need to occupy space in the flash image itself as the startup code simply zeroes this region in RAM. We know this from what we did with the default handler in our bare-metal implementation.

For what we did in bare–metal, there are a few things that our PowerShell script discerns differently compared to our SDK and HAL implementation; all code and read-only data goes to flash memory and the .text section already includes .rodata, so .rodata will be zero when we run the script compared to .data which is counted separately as it goes in both flash and RAM. Since we wrote our own second-stage boot code and vector table, we need to also include both of these files in our total flash size calculation.

4.6.2 Dynamic Analysis

Since we were developing in a bare-metal environment, tracking dynamic allocations and using the heap was not of importance to us. Firstly, using the heap on a 32-bit microcontroller with limited capacity and unpredictability in allocations is inherently unrealistic. This crosses off attempting to discuss smart heap or other memory allocation comparisons between the two languages. Furthermore, tracking memory during run-time using tools was not an option due to the bare-metal environment we were operating on. As such, we took an approach whereby we could still obtain reasonable comparison results. Dynamic analysis involved executing benchmark code directly on the RP2040 hardware to measure the runtime performance of specific operations that were used in both abstraction layers. Our original plan was to use the morse implementation as the base of our examination. Though plagued with issues and only two microcontrollers, this fell short of being a realistic idea. We still focus on the more feature-complete SDK/HAL abstract with a set of targeted micro-benchmarks designed to compare low-level peripheral access speed between C and Rust, as attempting to test the bare–metal implementations at run–time with the Rust's code's apparent issues wasn't ideal and was left out. The intention behind this is to analyse Rust's ability to meet real–time requirements and see if it's comparable or even at times better than the industry standard that is the C language. The C code in `benchmarks.c` served as the reference for these micro-benchmarks, with equivalent tests implemented in Rust within `benchmarks.rs`. The

original plan to evaluate memory with Miri was met with errors.Our issues arose for attempting to build with `cargo +nightly miri run --bin transmitter` because the `#[entry]` attribute, used in our `transmitter.rs` and `receiver.rs` to set up the RP2040's specific entry point and memory layout, creates a scenario Miri cannot simulate [66].

```
error: Miri can only run programs that have a main function.
       Alternatively, you can export a `miri_start` function:

       #[cfg(miri)]
       #[no_mangle]
       fn miri_start(argc: isize, argv: *const *const u8) -> isize {
       // Call the actual start function that your project implements, based on your
target's conventions.
       }

error: aborting due to 1 previous error
```

Listing 24: `miri.exe` Error

Execution time measurements relied on the RP2040's built-in hardware timer. In the C benchmarks, the Pico SDK functions `time_us_32()` and `get_absolute_time()` were used before and after the code segment under test; subtracting the start time from the end time provided the duration in microseconds. For operations measured within loops, the approach varied; for GPIO toggling, the total time for a fixed number of `COUNT` (1000) operations was measured, and this was repeated for (100) `TEST_RUNS`. For ADC sampling, (1000) `SAMPLE_COUNT` readings were taken per cycle, over several (100) `TEST_CYCLES`. For PWM configuration and UART transmission, measurements were typically repeated for (100) `TEST_ITERATIONS`. An average time per operation was then calculated to yield a stable metric. Interrupt latency was measured on an event-by-event basis. In the Rust benchmarks, the `rp2040_hal::Timer` abstraction was used. Timestamps were captured using `timer.get_counter()`, and the duration was calculated using the `checked_duration_since()` method on the resulting `Instant` objects, followed by conversion to microseconds using .to_micros(). For operations measured within loops in Rust, a similar approach of measuring total time for a fixed number of iterations was implemented to derive a stable average time per operation. Benchmarks covered ADC read times, GPIO toggle speed, PWM configuration time, UART data transmission throughput, and interrupt response latency.

4.7 Data Logging

The methods for loading the benchmarks onto the Pico and retrieving the measurement data differed slightly due to conventions and tool capabilities within the C and Rust embedded ecosystems.

For the C benchmarks, the compiled executable was loaded onto the Pico using the standard UF2 drag-and-drop. The benchmark results, formatted as strings using `printf`, were transmitted back to our host machine via the UART serial interface. This approach was chosen because it uses the standard I/O mechanisms provided by the Pico SDK and stdio, requiring only a simple serial terminal on the host to capture the output. It is a ubiquitous and straightforward method for debugging and data logging in C-based Pico projects.

For the Rust benchmarks, execution and logging were managed using the `probe-run` utility, executed on our host machine since the conventional `elf2uf2` crate was unable to be used during this project. It interfaces with the Pico Probe debug adapter to flash the Rust executable

onto the target and manage communication. Benchmark results were logged back to the host using the Real-Time Transfer (RTT) protocol, enabled by the `rtt-target` crate. During our Chapter 3 implementation, we relied on the GDB server and logging using semihosting. However, initial experiments indicated that semihosting, which functions by having the target CPU halt and signal the host debugger to handle I/O operations, introduced unacceptable delays and perturbations, potentially skewing the timing measurements. It also made logging over the 100 iterations slow. If we look at how semihosting works, we see why [61]:

1. The semihosting call that runs on the RP2040's CPU has a RISC-V `EBREAK` instruction that halts the CPU when a debugger is connected.
2. Our host PC polls the embedded CPU to check if it is halted.
3. When we halt, our host PC checks if the `EBREAK` instruction is part of a magic sequence.
4. If the sequence is found, the debugger treats the `EBREAK` as a semihosting call.
5. OpenOCD executes the requested semihosting system called on our GDB server.
6. The debugging host stores the return value in a register `a0`.
7. The debugging host unhalts the RP2040's CPU.

Consequently, the project transitioned to using Real-Time Transfer (RTT) for logging benchmark results from the Rust code. Instead of halting the CPU, RTT utilizes shared memory buffers established in the target device's RAM. The target code writes log data into these buffers. This write operation is usually non-blocking or very briefly blocking, allowing the target application to continue running almost entirely undisturbed. The host debugger, connected via our debug probe, then periodically polls these specific memory locations in the target's RAM in the background, without halting the target CPU. When the host detects new data written by the target into the RTT buffers, it reads the data out and displays it in the console. This asynchronous polling and use of dedicated memory buffers allows RTT to achieve very high data throughput with minimal impact on the target's real-time execution behaviour [62].

5 Evaluation

The evaluation delves into the practical differences and implications observed between C and Rust during the development of the project application on the RP2040. It systematically compares language features, non-technical aspects, and the overall experience of transitioning from C to Rust while simultaneously providing data to support the claims made at the beginning of our thesis.

5.1 Language Comparison

This section explores the technical aspects behind Rust's language features compared with C through what's been achieved in the implementation. By this point, enough experience has been gained by working at the lowest level up to the SDK to get a sense of its capabilities and discern the advantages and disadvantages witnessed early on. To reiterate this point– the Rust language itself is large in terms of what it can do. This evaluation only covers features witnessed during the implementations and of which are of importance to constrained embedded systems. This results in us overlooking features such as paradigm comparisons, pattern matching and dynamic memory allocation methods in Rust such as with smart pointers, d global allocators, `alloc::heap` or through `malloc()` and `free()` calls with the C ABI.

5.1.1 Memory Models

Memory models describe the interactions of threads through memory and their shared use of the data. Embedded systems are required to be inherently reliable and since much of Rust development may require unsafe Rust at times, with the challenge being in reducing the amount of code that uses unsafe language features. As of writing, Rust lacks a fully defined memory model distinct from its C11 inheritance [63] [64]. It largely adopts the C11 standard's approach to memory as contiguous bytes with size and alignment properties, where each byte has a unique address and objects generally cannot overlap [65]. This similarity was an initial advantage during the implementation, as the configuration of memory regions in both C and Rust linker files for the RP2040 followed comparable principles, crucial when working with the microcontroller's specific memory layout as depicted in Figure 7.

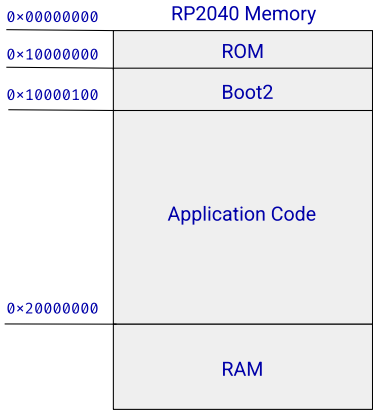

Figure 7: C11 Guarantees Aligned with the RP2040's 32–Bit Architecture

Where the model comparison between C and Rust becomes more evident is when we discuss pointers. At the hardware level, pointers are merely integers, but languages work on the abstract machine. Here, pointers not only consist of the address that it points to in memory, but also its *provenance*. This is termed a *shadow state* that carries along with every pointer information to track which memory regions the pointer can access. More formally, a pointer is a pair of some kind of ID uniquely identifying the allocation, and an offset into the allocation. All of a sudden, the C11 standard fails to apply to Rust when discussing byte semantics [67]. The C standard does not associate any extra metadata with bytes, but in Rust it'd contain fragments of pointers that carry semantic information to satisfy the constraint behind arbitrary casts between pointers and integers. This fundamental difference directly impacted our translation from C to Rust; C code in our bare-metal implementation that relied on implicit pointer behavior, or code that might have invoked Undefined Behavior (UB) through pointer misuse, required careful restructuring in Rust. For instance, accessing our hardware registers structs `sio_hw_t` and `io_bank0_hw_t` in C through raw pointers needed to be encapsulated in unsafe blocks in Rust, and the logic often refactored to comply with Rust's stricter compile-time checks enforced via its ownership and type system. This was a recurring theme in translating direct memory access patterns from `bootStage2.c` to `bootStage2.rs`.

5.1.2 Ownership

The ownership system outlines rules that govern memory management, enforced at compile time rather than runtime without the need for a garbage collector. These fundamental rules state that each value has exactly one owner, only one owner exists at any given time, and when the owner goes out of scope, the value is automatically dropped [71].

Rust complements ownership with borrowing: creating references with `&` for immutable and `&mut` for mutable that allows access to data without transferring ownership. The borrow checker enforces these rules. Multiple immutable references or exactly one mutable reference can exist for a given piece of data at any time, but not both. Furthermore, references cannot outlive the data they point to. These rules eliminate dangling pointers and many data race conditions by construction [72]. Figure 8 visually describes this concept [73].

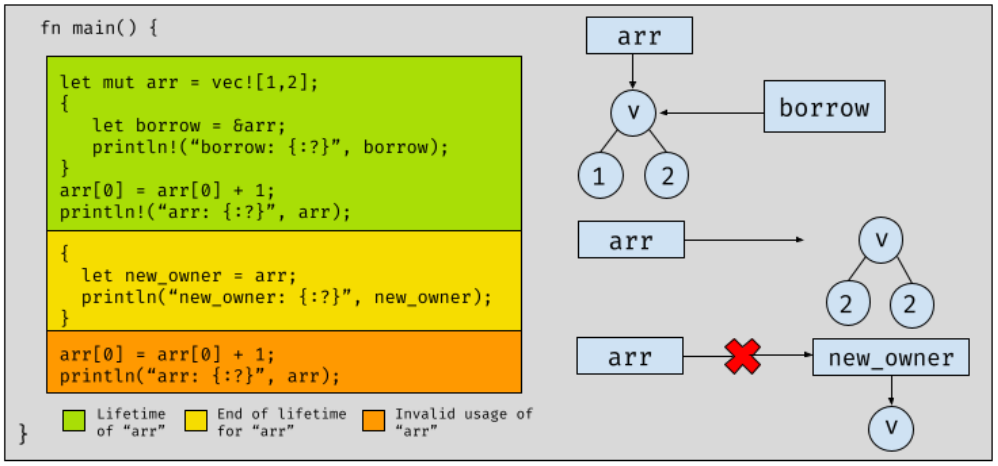

Figure 8: Ownership, Borrowing & Lifetimes

C has no equivalent concept to this system. While C does allow pointers to access and modify data, it provides no compile-time guarantees about their validity or the exclusivity of access. This leads to common memory errors in C programs: use-after-free, double-free, null pointer

dereferences, and memory leaks from manual memory management with `malloc()` and `free()`, errors that are otherwise caught by the ownership and borrow checker system. It removes many data races by preventing concurrent access to resources through multiple threads.

During the migration, Rust's ownership system profoundly influenced implementation. C code patterns involving multiple pointers to mutable data or complex data sharing would have required significant refactoring. The difficulties encountered with the vector table implementation in section 3.2.2, requiring unsafe blocks, exemplified the translation challenge when Rust's ownership rules met low-level hardware expectations that mimic C arrays of function pointers. Similarly, the borrow checker `error[E0502]` code encountered in section 3.4.2, when attempting to simultaneously hold a reference to the UART peripheral for logging and another for receiving, necessitated restructuring the code using splitting borrows by dividing the peripheral's capabilities into separate, borrowable parts to satisfy Rust's exclusivity rules. While ownership demonstrably prevents a large class of C memory errors, it sometimes felt restrictive, especially when dealing with static resources initialized at startup, like our peripherals which were wrapped in `Option` and `RefCell` within a Mutex for safe sharing through dynamic borrowing where needed or requiring shared access in contexts Rust's borrow checker scrutinizes heavily [74].

5.1.3 Types and Traits

The differences in type systems between C and Rust significantly impacted the implementation strategy, particularly when moving from direct register manipulation and SDK usage to Rust's HAL abstractions. C has a relatively simple, weak type system consisting of primitive types, structs, unions, enums, and pointers. While effective for low-level control, it lacks generics that have to be simulated with `void*` pointers and manual casting or through preprocessor macros [75]. It offers no built-in mechanism for enforcing interface contracts or managing the lifetime of references beyond programmer discipline. Our C bare-metal code heavily relied on structs mapped directly to hardware register layouts such as with `sio_hw_t`, `io_bank0_hw_t` and volatile pointers requiring careful manual handling to ensure correctness. The same was with Rust but was made clear within an `unsafe` block. Because of the project's deadline, our intention for this was to not take too long to overhaul the data structures to keep the borrow checker happy. For our sake, we traded safety for refactor time to keep the project moving along. Real business practices, especially within the cybersecurity and defense sectors typically would go oppose this, so the results are clear; in C the chances of tripping into using these unsafe attributes are much more likely, whereas in Rust we are able to actively mitigate this risk and if we are faced in a situation where the compiler complains, its marked clearly where the unsafe code is present for anyone operating on the codebase to see [76].

When we begin to compare at the abstracted level through the HAL, Rust comes out on top. It conversely features a strong, static type system that supported generic programming and traits [59]. While I didn't myself define traits within my code, the HALs implemented them and one benefit they provide was through leveraging generics. It allow writing code that operates over multiple types without through zero-cost abstractions while traits define shared functionality, akin to interfaces. For example with the `rp2040-hal` crate. In our case, the GPIO pins are typed based on their state with `Pin<Gpio16, FunctionSio<SioInput>, PullUp>` preventing misuse like writing to an input pin at compile time. This contrasts sharply with C, where pin configuration relies on setting register bits correctly and leading to runtime errors if done improperly. The compiler errors encountered during the HAL implementation, such as with `error[E0412]` when pin types were incorrect, highlight the strictness and safety benefits of

Rust's system, guiding the developer towards correct usage, albeit sometimes requiring more verbose type annotations or careful structuring compared to C's flexibility.

5.1.4 Arrays

Handling collections of data, specifically arrays, also revealed key differences influencing development. C treats arrays as contiguous blocks of memory, decaying to pointers in most cases when passed to functions making getting the length of arrays difficult without allowing your function to map to some array size since the information is not saved in the layout of memory. In Rust, this doesn't occur as easily as they do in C and allows us to call `.len()`.C provides no inherent bounds checking; accessing an array out of bounds results in undefined behaviour. Our C code used arrays for register blocks, lookup tables and implicitly through string handling. Managing their size and access boundaries was solely our responsibility [77].

Rust distinguishes between fixed-size arrays `[T; N]` and dynamically sized slices `&[T]`. Slices provide a "view" into an array or other contiguous sequence, carrying both a pointer to the data and the length of the sequence. Indexing arrays or slices in safe Rust performs bounds checking by default. In the case an array reference is passed, Rust can't work out at compile time if the indexing is out-of-bounds. Instead, it will do bounds checking at runtime and panic. This almost always causes negligible overhead and prevents buffer overflows. Since we're on a constrained microcontroller it's preferable if we can get rid of this overhead when required, using `get_unchecked()` [77]. Our Rust bare-metal implementation used fixed-size arrays for the vector table and register definitions. In the HAL-based code, the `morse_code` array provided morse mappings, and importantly, the receiver.rs code utilized the heapless crate to create fixed-capacity String types with `String<{MAX_MESSAGE_LENGTH}>` for message buffering, avoiding the need for a dynamic memory allocator within a no_std environment while still providing safe, bounds-checked string manipulation. In such a constrained system, this was notable.

5.1.5 Concurrency

Concurrency was slightly more difficult to investigate. This project primarily involved single-core execution rather than preemptive multithreading due to its inherent complexity, simple nature of our projects and deadlines. Across languages, approaches to handling concurrent access differ. Both languages define a *happens-before* relationship for single-thread execution ordering [78]. However, managing shared mutable state, especially hardware registers accessed by both the main execution path and Interrupt Service Routines (ISRs) can pose challenges. In C, this typically relies on the `volatile` keyword to prevent the compiler from optimizing away reads/writes to shared memory locations.

Rust addresses concurrency safety, particularly data race prevention, through its ownership and borrowing rules enforced by the type system. We saw this with the `Send` and `Sync` marker traits. For more complex sharing, primitives like mutexes `cortex_m::interrupt::Mutex` would be used to encapsulate shared data and provide safe access, even within interrupt contexts. Within our HAL implementation, use of the `nb` crate allowed us to write core I/O APIs for our UART logging [79]. Here we took the busy-waiting approach that polls the hardware until operations complete, it significantly simplifies code that would otherwise require manual state tracking and context management:

```rust
pub fn uart_log(&mut self, message: &str) {
    for byte in message.as_bytes() {
        let _ = block!(self.uart.write(*byte));
    }
    let _ = block!(self.uart.write(b'\r'));
    let _ = block!(self.uart.write(b'\n'));
}
```

Listing 25 `!block` usage in `uart_log()`

This way when `uart.write()` is called:

1. It tries to write a byte to UART.
2. If it returns `WouldBlock` or `InProgress` errors, retries the operation.
3. When successful, returns the actual result.

What we gain in code simplicity, we lose in CPU efficiency with this method since the processor remains occupied during these blocking operations. However, in applications with strict real-time requirements, alternative approaches such as interrupt-driven I/O or more sophisticated task schedulers can be utilised. This nicely introduces us to Rust's async programming, a powerful alternative for concurrency in embedded systems. Async programming allows for cooperative multitasking, where tasks voluntarily yield control when they would otherwise block (e.g., waiting for a peripheral). This can lead to more responsive applications and better CPU utilization than the `block!` macro's busy-waiting. For instance, our receiver.rs logic, which waits for incoming Morse signals via GPIO and then processes them, could have been structured as an async task. Similarly, our UART logging could be an async operation:

Algorithm 1

```
PROCEDURE async_monitor_input(pin)
    LOOP FOREVER
    WAIT FOR pin LOW  // Non-blocking wait
    PRINT "Signal detected!"
    // Process signal start (record time)

    WAIT FOR pin HIGH // Non-blocking wait
    // Process signal end (measure duration)
    END LOOP
END PROCEDURE
```

Listing 26: Concept `async fn monitor_morse_input()`

In this scenario, something like `wait_for_low().await` would yield control if the pin isn't low, allowing other tasks such as blinking a status LED and handling UART commands to run. This would make the system more responsive than a purely polling or `block!` based approach for multiple concurrent activities.

5.1.6 Error Handling

Error handling paradigms diverge significantly. C relies on return codes through integers and null pointers to signal errors, often setting a global errno variable for more details. It's the caller's responsibility to check these return values; forgetting to do so can lead to silent failures or crashes later on. Our C SDK code used these standard patterns, like checking the return value of `i2c_write_blocking`,

Rust employs a more structured approach using e `Result<T, E>` for recoverable errors and the `panic!` for unrecoverable errors. Result will force the caller to explicitly handle potential failure, either by matching on the `Ok(T)` and `Err(E)` variants, or by using combinators like `map`, `and_then`, or methods like `unwrap()` and `expect()` which panic on `Err` or the `?` operator for propagation [80]. This compile-time enforcement prevents accidentally ignoring errors. Panics unwind the stack by default, though configured to abort in our no_std build, terminating the program or current thread. Our Rust HAL code extensively used `Result` and associated methods. Our functions often return Result that are implicitly handled via unwrap() in many examples like `InputPin::is_low().unwrap()`, `OutputPin::set_high().unwrap()`, `init_clocks_and_plls( ... ).ok().unwrap())`. For unrecoverable errors such as with configuration issues, `panic!` provides an explicit exit point, handled by the `panic_halt` crate in our case, which simply enters an infinite loop [81]. The migration strategy involved translating C's return code checks into Rust's Result handling, leading to arguably more robust code as the compiler forces acknowledgment of potential failures. The explicit panic mechanism also makes fatal errors clearer than potential UB or silent failures in C. In systems that are intended to be reliable throughout the duration of their lifetime, Rust poses this clear advantage.

5.1.7 Macros

Both languages utilize macros, but their implementation and capabilities differ. C uses a preprocessor for macro expansion. These are simple text substitutions performed before compilation, useful for our defined constants and simple function-like macros. However, they can lead to variable capture and type safety, and complex macros can be difficult to debug.

Rust offers a more sophisticated and safer macro system that is integrated with the compiler. Declarative macros with `macro_rules!` provide pattern-matching capabilities and are hygienic by default, avoiding accidental name collisions. Procedural macros operate directly on the Abstract Syntax Tree [82], allowing for better code generation, including custom `#[derive]` implementations for traits. Our Rust code used built-in macros extensively, such as `hprintln!` for semihosting output during debugging phases of `transmitter.rs` and `receiver.rs`. This is notably safer than C's `printf` because `hprintln!` and similar formatting macros like `format!` check the format string and arguments at compile time, panicking during compilation if there's a mismatch, thereby preventing runtime errors or potential vulnerabilities associated with C's format string bugs. While we didn't write complex custom macros in this project, the use of `#[entry]` and various `#[derive( ... )]` attributes are examples of procedural macros from the ecosystem that simplified our code and ensured correctness for boilerplate tasks.

5.2 Non-Technicalities

Beyond the direct technical merits of each language, several non-technical factors significantly influence the decision to adopt a language. My experiences during this project, from setting up the development environment for both C and Rust for the RP2040 to debugging and integrating libraries like the Pico SDK and rp2040-hal, provide a practical basis for evaluating these aspects.

5.2.1 Community Support

The availability and quality of community support, alongside the breadth and depth of the ecosystem, are critical factors for developer productivity and efficient problem-solving in any

programming language. This was particularly evident when comparing C and Rust for our RP2040 project.

C, with its decades-long history in embedded systems, possesses an enormous and mature ecosystem. During the C implementation phase of this project, particularly when working with the Pico SDK, this maturity was a clear advantage. I found a wealth of readily available documentation from Raspberry Pi themselves, extensive forum discussions, and a vast array of examples for virtually any issue encountered. Standard libraries are well-established, and vendor support, such as that from Raspberry Pi for the RP2040, evidenced by resources like the official RP2040 Datasheet and detailed C examples in pico-examples repository, is extensive. For instance, resolving issues with the I2C communication for the LCD in the C SDK was significantly facilitated by referencing existing examples like the `lcd_1602_i2c.c` example within the Pico SDK and community threads such as those found on the Arduino forums detailing similar setups with the RP2040. This vast repository of shared knowledge allows many common problems to be resolved quickly. Practically no issues existed with support in C.

Rust, while younger, has cultivated a rapidly growing and notably enthusiastic community, particularly around embedded development. For this project, resources like "The Embedded Rust Book," the official rp2040-hal documentation, and active community forums such as the Rust Embedded WG discussions on Matrix and the official users.rust-lang.org forum were invaluable. The core Rust documentation itself, including "The Rust Programming Language" book covering concepts like ownership, borrowing, generics, and unsafe Rust and "Rust by Example" provides a strong foundation. The only issue is that many of those documents, especially the unsafe documents, are yet to be finished and still underdevelopment. During the Rust HAL implementation, particularly when facing compiler errors related to pin type error `[E0412]` or borrow checker issues with `[E0502]` in `receiver.rs`, the specificity of Rust's error messages often provided a direct path to understanding the problem and provided documentation to resort to in the message. When further clarification was needed, online community support frequently pointed to the solution or a relevant discussion, sometimes in blog posts from experienced community members such as those explaining Derive/Clone mechanics or diving into pointer provenance or forum threads where similar issues were resolved.

However, the Rust embedded ecosystem, while maturing quickly for popular targets like the RP2040, is not yet as universally comprehensive as C's. During the project it was clear to the community that they had taken the right steps to develop the tooling to allow for such a migration; Rust has a much more diverse and apparent ecosystem. It is a lot more common to experience pitfalls however. Despite this, I experienced much of the community being supportive through GitHub and the forums and it helped me tackle issues revolving around the errors I faced in my implementation. While rp2040-hal is a well-maintained and capable crate, with community contributions also visible in related efforts like rp2040-boot2, finding pre-existing, highly specific peripheral driver examples beyond the basics, or troubleshooting niche hardware interactions, sometimes required more investigative work or adapting examples from other HALs than in the C world. This is because C has had time to mature and much of the issues are already thoroughly documented. In Rust, I found myself on forums and reddit posts more than I did in official documentation, and on many forums, it was the same people responding to issues. The actual official documentation for topics, like the Rustonomicon for Send and Sync traits or the Unsafe Code Guidelines for features like union layouts, is thorough but took me some time to wrap my head around . The challenges I encountered with the elf2uf2 crate required a specific version for our project, 3.25 and the probe-rs core lock-up issue, which remained unresolved

through readily accessible community channels during the project timeline. These are often acknowledged and discussed openly within the community through things like Reddit discussions on Rust's memory model implications or internals.rust-lang.org threads on pointer complexities, but solutions may not always be immediate, reflecting a tooling and ecosystem landscape that is more dynamic than C's environment.

5.2.2 Work–Flow

The development workflow and the available tooling significantly impact developer experience and productivity. For C development on the RP2040, the workflow involved setting up the ARM GCC toolchain, CMake for build management with the Pico SDK, and OpenOCD with GDB for debugging. While these tools are powerful and standard, their initial setup and configuration can be somewhat fragmented. Crafting the manual Makefile for the bare-metal C project was a detailed and sometimes tedious process, requiring careful management of compiler flags, linker scripts, and dependencies like CRCpp. Debugging with GDB, while effective, often involves a more manual command-line-driven process, as seen during the step-through debugging of the C bare-metal and SDK implementations.

Rust's workflow, centered around Cargo, presented a more integrated experience and using `build.rs` files that are much easier to formulate and write given that one is comfortable with Rust syntax. By the time I transitioned to the HAL implementation, I was already comfortable with the work-flow presented to Rust. I preferred organising the project into /bin that allowed me to test my files and compile them individually as opposed to C where I was compiling everything at once. I was much more familiar with the style of work-flow that I adopted in C however, Cargo steered me into adopting a more optimal work-flow during development. Cargo handled dependency management, building, and a variety of other development tasks seamlessly. Setting up a new Rust embedded project for the RP2040 was streamlined by templates and the cargo generate command. The Cargo.toml file provided a clear and concise way to manage dependencies like cortex-m, cortex-m-rt, panic-halt, and rp2040-hal. Rustc's compiler's error messages, as encountered with the vector table and type issues during the bare-metal Rust port, were notably more informative and helpful than typical C compiler errors, often suggesting concrete fixes. This significantly aided the debugging process, even if "fighting the borrow checker" was a distinct phase in the learning curve. However, the Rust workflow was not without its challenges. The linker issues encountered with rust-lld overlapping .boot2 and .vector_table sections and the persistent probe-rs errors for the HAL implementation demonstrated that tooling, even with Cargo's integration, can still present complex problems that require significant troubleshooting. The need for two different linker scripts for different Rust build targets also added a layer of complexity not encountered with the more monolithic CMake build process.

5.2.2 Maturity

C is the epitome of maturity in the embedded world. Its language standard evolves slowly, providing a stable foundation. Compilers like GCC have been battle-tested for decades and generate highly optimised code for a vast array of architectures. The Pico SDK used in this project is a mature, vendor-supported library, which contributed to a generally stable development experience for the C implementation. The ready availability of solutions to common problems reflects this maturity.

Rust, having reached its 1.0 release in 2015, is considerably younger. However, it has implemented strong stability guarantees through its edition system, ensuring that code written

for older editions continues to compile. The Rust compiler rustc and Cargo are robust and feature-rich, but the embedded ecosystem, particularly HALs and device drivers, is still in a phase of active development and expansion. As observed in Section 2.2, while support for popular MCUs like the ARM Cortex-M series is strong, it is not yet as universal as C's. My experience with rp2040-hal was largely positive, showcasing a capable HAL crate. Despite this many dependency issues were encountered while using the HAL where the SDK presented no issues regarding this. Fortunately, running `cargo tree` and looking at crate documentation was more than enough to help solve this issue. Regardless, me encountering issues like the probe-rs lock-up in section 3.4.3 and the need to rely on features like `#![feature(linkage)]` suggests a tooling and language feature landscape that is more dynamic than C's. While Rust's core is stable, the cutting edge of its embedded development can involve navigating more recently stabilized features or workarounds for tooling idiosyncrasies. While C has more mature tooling, Rust being "newer" led to me being pointed towards more up-to-date tooling with more features tied into their functionality, such as with probe-rs having the capability of running with RTT. This made debugging much easier on Rust than using ARM semihosting in C as OpenOCD's setup process was tedious to setup–commands didn't save and retyping the commands to the server was daunting. The challenges in implementing the vector table in Rust, requiring multiple unsafe attempts, also point to the learning curve when applying a newer language with different paradigms to well-established low-level patterns typically handled in C.

5.2.4 Standards and Practices

The adoption and adherence to established standards and best practices are paramount in embedded systems development, particularly for ensuring reliability, safety, maintainability, and interoperability. Different languages foster or necessitate different sets of practices.

In the C programming language, its long history in embedded development has led to the establishment of various coding standards aimed at mitigating its inherent risks. Standards like MISRA C (Motor Industry Software Reliability Association) [83] are widely recognized and often mandated in safety-critical sectors such as automotive and aerospace. MISRA C provides a subset of the C language, restricting or disallowing features prone to causing undefined behavior or errors, such as unrestricted use of pointers, dynamic memory allocation, and certain preprocessor directives. While this project did not formally adhere to MISRA C due to its exploratory nature, common C practices were followed, such as careful pointer arithmetic in the bare-metal implementation, explicit volatile qualifiers for hardware register access and adherence to the Pico SDK's API conventions. The C standard itself with C23 being the target for this project defines the language semantics but relies heavily on programmer discipline and external tools such as linters and static analyzers to enforce safer practices. The very existence and necessity of standards like MISRA C underscore the language's capacity for unsafe operations if not carefully managed. My experience in the C bare-metal section, particularly with memory layout in linker.ld and direct pointer-based register access, highlighted this need for meticulous, disciplined coding to ensure correctness, a responsibility that falls squarely on the developer.

Rust, on the other hand, integrates many safety-oriented practices directly into its language design and compiler. The ownership and borrow checking system discussed in 5.1.2 is the most prominent example, effectively enforcing memory safety and data-race freedom for the safe subset of Rust at compile time. This eliminates the need for many of the restrictive rules found in C standards like MISRA C because the prohibited behaviors are often impossible to express in safe Rust. The "standard practice" in Rust involves writing idiomatic code that satisfies the

borrow checker. When low-level control necessitates bypassing these guarantees, Rust requires the explicit unsafe keyword. The best practice, strongly encouraged by the Rust community and essential for maintaining overall system safety, is to minimise the scope of unsafe blocks and, wherever possible, encapsulate them within safe abstractions. My bare-metal Rust implementation required unsafe blocks for direct hardware interaction and FFI-like behavior. The challenges encountered with the vector table, even within an unsafe context, demonstrated that Rust still guides developers towards considering memory layout and aliasing rules carefully, unlike C where such operations might compile silently but hide latent bugs.

Furthermore, the Rust ecosystem promotes standardized practices through tools like Cargo for project structure and dependency management, and rustfmt for code formatting. The emphasis on comprehensive documentation via rustdoc is another strong community practice. While Rust is newer and specific embedded safety certifications for its toolchain are less established than for some C compilers, the language's design principles are well-aligned with the goals of safety-critical software development. Efforts like RustBelt that are mentioned in section 2.4, which formally verifies the soundness of Rust's safety mechanisms and unsafe code abstractions, represent an advanced practice pushing towards provably safe systems. The difficulties encountered in the C-to-Rust migration, such as refactoring C code that relied on implicit pointer behaviors or potential UB, inherently involved adopting Rust's stricter, safer practices. The compiler acted as an enforcer of these practices, often preventing compilation until memory or type safety could be guaranteed, a stark contrast to the C development experience where such issues might only surface during runtime or in obscure failure modes.

5.3 The Switch to Rust

From undertaking this project, I understand why the idea of a switch has been so laid back. Take C syntax for example. When looking at C, it's easy to see what its output to assembly will look like. It does a great job at abstracting the hardware underneath and conveying it to the programmer. Rust’s syntax takes some time to learn and personally, I believe it doesn’t do such a similar job in this regard. The decision to migrate from a deeply entrenched language like C to a newer alternative such as Rust in the embedded domain is multifaceted, involving a trade-off between established practices and potential future gains. This project, through the direct migration of a morse code application on the RP2040 from C to Rust at both bare-metal and SDK/HAL levels, provides first-hand insights into this transition. The primary motivation for considering Rust, as highlighted in the literature review, is its promise of memory safety without a garbage collector, aiming to eliminate entire classes of bugs that plague C programs. My experience confirmed that Rust's compiler, particularly the borrow checker, acts as a stringent gatekeeper, forcing a more disciplined approach to memory and resource management, as seen with the receiver.rs refactoring due to borrow errors (Section 3.4.2). This upfront rigor, while challenging, is designed to yield more robust software.

However, the switch is not without significant hurdles. The most immediate is the learning curve, followed by considerations of ecosystem maturity and the practicalities of integrating Rust into existing C-dominated workflows or porting C idioms that do not map directly to safe Rust. We observed this in Chapter 3.

5.3.1 Learning Curve

Rust's learning curve, especially for developers experienced primarily in C, is undeniably steep, a point also raised as a risk in section 1.3. C is simple and its syntax and memory model are

relatively straightforward to grasp. Rust introduces several novel concepts that require a paradigm shift.

The ownership and borrowing system, Rust's cornerstone for memory safety, was the most significant conceptual hurdle encountered in this project. The compiler's strict enforcement of these rules—one mutable reference or multiple immutable references, but not both—led to initial friction. While it did stall development, we showed that it was all for the right intention; reducing the amount of unsafe code in our work. Not only did it contribute to this, but it highlighted the importance of maintaining a single state of variable and preventing multiple copies being created. Errors encountered during the vector table implementation in bare-metal Rust where attempts to cast function pointers to integers for the table entries were met with compile-time errors related to const evaluation and thread safety, highlighted the compiler's vigilance and the need to understand Rust's deeper semantics, even within unsafe contexts. This contrasts sharply with C, where such conversions, while potentially dangerous, would often compile, deferring error detection to runtime, if at all.

Lifetimes, though not a major point of contention in this specific project due to its relatively simple data-sharing patterns, are another core Rust concept that typically contributes to the learning curve. The Rust type system, with its strong static typing, generics as seen in 5.1.3 and traits, is more expressive and complex than C's. While this enables powerful abstractions and compile-time correctness, it also demands a greater initial investment in learning.

Despite these challenges, Rust's learning resources are a significant asset. "The Rust Programming Language" book and "The Embedded Rust Book" provided excellent, comprehensive, and freely accessible documentation. The compiler's error messages, as frequently noted, were exceptionally helpful and taught me what I should be looking out for when coding in system languages, often providing precise explanations and suggestions, which greatly aided the learning process during this project. This is a marked improvement over the often cryptic error messages from C compilers. My experience aligns with the notion that while "fighting the borrow checker" is a common rite of passage, it ultimately instills better memory management discipline. From using Rust, it has even helped me further my knowledge of programming semantics and memory. The initial time investment in learning Rust, particularly its safety features, can be substantial, but it aims to pay dividends in reduced debugging time and increased reliability later, a contrast to C where rapid initial development can be offset by prolonged bug-fixing phases for memory-related issues.

5.3.2 Portability

Portability in embedded systems refers to the ease with which software can be adapted to run on different microcontroller architectures or product variants with minimal code changes.

C's portability is a nuanced topic. The language itself is highly standardized and C compilers exist for virtually every microcontroller. However, practical portability of C embedded code is often limited by its heavy reliance on vendor-specific Software Development Kits (SDKs), register-level details, and toolchains. My C bare-metal code, with its direct register manipulations, was tightly coupled to the RP2040. The C SDK implementation, while abstracting some hardware details via the Pico SDK, was still fundamentally tied to that SDK and the RP2040 target. Porting this C code to a different microcontroller family would necessitate significant rework, primarily adapting to a new SDK and potentially different peripheral behaviors.

Rust aims to improve portability in the no_std embedded context through the Hardware Abstraction Layer (HAL) and Peripheral Access Crate (PAC) ecosystem. PACs provide low-level register definitions auto-generated from SVD files. HALs, like the rp2040-hal used in this project, build upon PACs to offer higher-level, often trait-based interfaces to peripherals. The embedded-hal crate defines a set of common traits that drivers can implement, theoretically allowing a driver written for an embedded-hal compliant I^2C interface to work with any microcontroller that has an embedded-hal compliant I^2C implementation in its HAL. My Rust HAL implementation for the transmitter and receiver utilized rp2040-hal, making it portable across different boards using the RP2040. To port to a different microcontroller, the primary effort would involve switching to the HAL crate for that specific target, and adapting any direct PAC usage or HAL-specific features. While this provides a better abstraction layer than typical C SDK approaches, the practical portability still depends on the availability and quality of HAL crates for the desired targets. As noted in the literature review, HAL coverage is not yet exhaustive for all MCU families. Furthermore, both C and Rust embedded development require careful management of linker script to define memory layouts, which are inherently target-specific. My experience showed that linker script configuration was a critical and sometimes problematic step in both languages. Thus, while Rust's HAL ecosystem offers a promising path towards better portability of application logic, significant target-specific configuration remains a necessity for both languages at the lowest levels.

5.2.3 Interoperability

C exhibits excellent interoperability with assembly language and is often the lingua franca for low-level interfaces. Most operating systems and hardware drivers expose C APIs. This project, while a migration, implicitly relied on C's interoperability with the hardware itself, as register definitions and memory layouts defined in datasheets are inherently C-like in their structure.

Rust provides strong Foreign Function Interface (FFI) capabilities for interacting with C code. The unsafe keyword is pivotal here, allowing Rust to call C functions and work with raw pointers. Tools like bindgen [84] can automatically generate Rust bindings to C headers, significantly easing the process of using C libraries from Rust. While this project focused on a full rewrite rather than incremental FFI integration, the process of translating C hardware register structs and direct memory manipulation into Rust using unsafe blocks and volatile read/write intrinsics is a form of direct interoperability at the hardware interface level. This experience demonstrated that Rust can map C-style hardware interactions, albeit with the necessary unsafe annotations to acknowledge the bypass of Rust's safety guarantees. Calling Rust code from C is also possible but can be more complex due to Rust's name mangling and different ABI conventions, which required us to explicitly state `#[no_mangle]` annotations and `extern "C"` declarations on the Rust side. The literature review mentioned that many Rust embedded crates use FFI but can face type incompatibilities, necessitating manual wrapper engineering. This project did not delve into complex FFI with large C libraries, but the need for careful unsafe management when mirroring C's direct hardware access underscores the attention required at these boundaries.

5.4 Experiment 1

The static analysis phase of this investigation compared the resource utilisation characteristics of the C and Rust implementations developed for the RP2040 platform, encompassing both bare-metal and SDK/HAL abstraction levels. This analysis provides objective data on the

resource footprint of each language and abstraction approach, offering foundational insights into their suitability for memory-constrained embedded systems, a core aspect of analysing Rust's viability in this domain. We found that the static analysis strongly favours Rust in terms of RAM efficiency due to its architecture discouraging large static allocations. While optimised Rust achieves superior flash efficiency compared to the C SDK implementation, unoptimised Rust binaries can be larger, indicating a greater reliance on compiler optimisations, particularly dead code elimination and monomorphization handling, to achieve minimal size.

5.4.1 Static RAM

Static RAM utilisation revealed the most dramatic differences, particularly when comparing the HAL/SDK implementations to bare-metal code, and between Rust and C when using higher-level abstractions. As Figure 9 illustrates, Rust HAL implementations required minimal static RAM . This is in stark contrast to the C SDK builds, which consumed substantial static RAM. This greater than 99% reduction in Rust's static RAM usage is directly attributable to its memory management philosophy, particularly its ownership system that encourages stack allocation and RAII patterns, thus minimising reliance on global static variables. The C SDK's larger footprint here is evident from Table 3, which shows significant C static allocations for entities like `hw_endpoints` and `_vendord_itf`. In contrast, Rust's largest static variables in the HAL implementation are typically very small, such as cached ROM function pointers or single-byte peripheral flags. When examining the bare-metal implementations, both the optimised Rust and the C versions for the transmitter achieved zero static RAM allocation as confirmed by their respective detailed memory summaries. Both seemingly rely instead on their pre-allocated stack. This comparison highlights that while both languages achieve minimal static RAM footprints with direct hardware programming, Rust maintains its significant advantage even when using its HAL abstractions, unlike the C SDK which introduces substantial static RAM overhead. This characteristic is highly beneficial for embedded systems where RAM is often minimal.

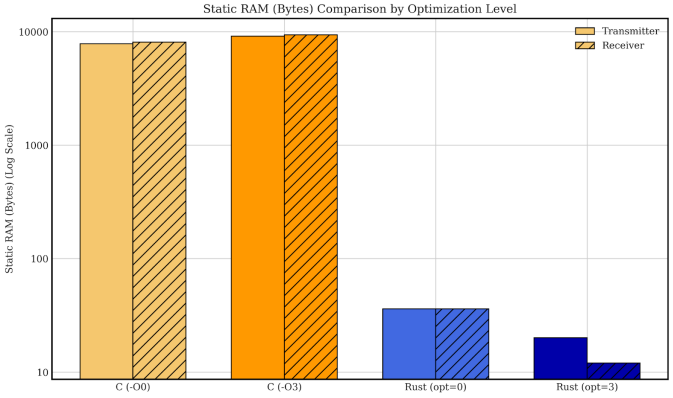

Figure 9: Static RAM Comparisons

5.4.2 Flash Memory

For SDK/HAL implementations, optimised Rust produced significantly smaller binaries than the optimised C SDK. This advantage for optimised Rust likely stems from LLVM's optimisation effectiveness, potentially more aggressive dead code elimination in the Rust HAL, and the considerable overhead included in the linked C SDK libraries. However, unoptimised Rust HAL builds were larger than their unoptimised C SDK counterparts, primarily due to factors like Rust's generics monomorphization and less aggressive code stripping in debug profiles. Turning to the bare-metal implementations, in C we achieved a minimal footprint at 796 bytes. The

optimised Rust bare-metal transmitter was slightly larger at 1200 bytes. Both bare-metal implementations are orders of magnitude smaller than their SDK/HAL counterparts, starkly illustrating the significant flash cost associated with higher-level abstraction libraries in both languages.

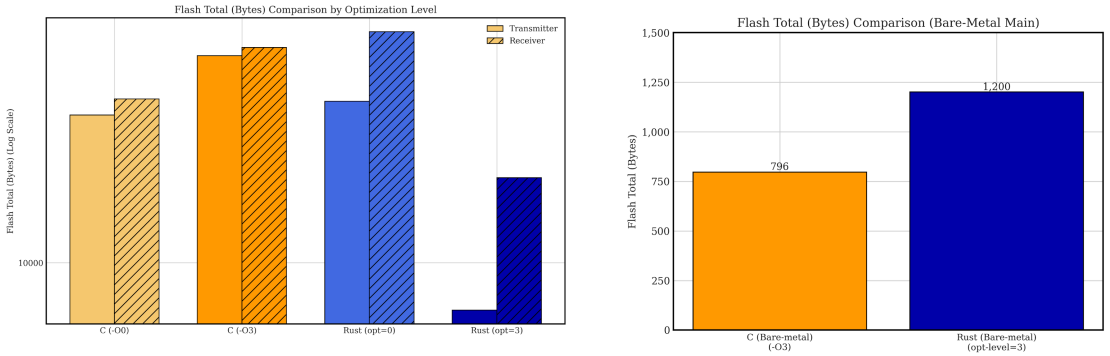

Figure 10: Flash Memory Comparison

5.4.3 Largest Functions and Static Variables

Analyzing the largest functions reveals the composition of the binaries. In C SDK builds, a significant portion of the code size is attributable to SDK components like `_vsnprintf`, USB task handlers (`tud_task_ext, dcd_rp2040_irq`) and standard library functions (`_ftoa, _etoa`). In contrast, optimised Rust binaries show the `__cortex_m_rt_main` function containing the our application logic and compiler built-ins like integer division as major contributors, alongside HAL initialization code. This suggests that while Rust's core abstractions are efficient, the user application code becomes a more dominant factor in the final binary size, whereas the C SDK carries substantial inherent overhead regardless of application complexity. Unoptimised Rust shows more standard library functions contributing significantly, aligning with the larger unoptimised binary sizes observed.

5.5 Experiment 2

The dynamic analysis focused on measuring the runtime performance of key peripheral operations using micro-benchmarks executed directly on the RP2040, comparing Rust HAL (optimised and unoptimised), Rust Raw register access (optimised), and C SDK (optimised and unoptimised) implementations. In summary we find that optimised Rust, particularly using HAL abstractions, can achieve performance highly competitive with, and in some cases like GPIO toggle speed and interrupt latency consistency exceeding the optimised C SDK implementation. Direct register access through unsafe writing (Rust Raw) did not universally yield the best performance, highlighting that HAL/SDK implementations often incorporate efficient low-level techniques like DMA or better interrupt handling that simple polling loops lack. Unoptimised code in both languages suffered significant performance penalties, particularly for CPU-intensive tasks. For real-time systems, the predictability observed in the optimised Rust HAL's interrupt latency is promising, although the C SDK's performance in areas like UART transmission remains strong. A potential area for improvement would be a more in-depth analysis using profiling tools to understand cycle counts for specific code paths and investigating the root cause of the high variability in C SDK interrupt latency measurements. Furthermore, extending benchmarks to include more complex scenarios, longer data transfers such as UART DMA, and concurrent operations would provide a more comprehensive performance picture.

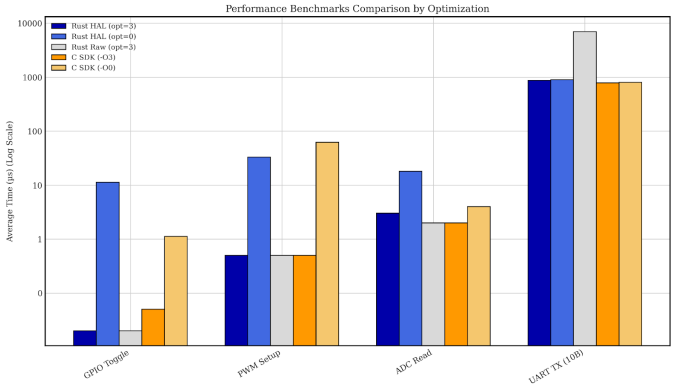

Figure 11: Comparative Performance Benchmarks

5.5.1 GPIO Toggle Speed

GPIO toggle speed benchmarks demonstrate the minimal overhead of Rust's optimised abstractions. Both Rust HAL and our attempt to write unsafe Rust code with optimisations achieved toggle times around 0.02 µs, significantly faster than the optimised C SDK and vastly exceeding the unoptimised C SDK and unoptimised Rust HAL. This indicates that when optimised, Rust's HAL abstractions for simple operations like GPIO toggling compile down to highly efficient machine code, matching or even slightly exceeding direct register access performance in this specific test and significantly outperforming the C SDK equivalent. Both C and Rust show similar performances.

5.5.2 PWM Setup Time

PWM setup time exhibited high variability, especially in optimised builds where initial setup times were higher before settling near zero or sub-microsecond times. Both optimised Rust (HAL/Raw) and optimised C SDK showed average setup times around 0.5 µs after stabilisation, suggesting minimal configuration overhead once initialized. However, the unoptimised versions showed substantial setup times with Rust HAL unoptimised having ~33 µs and C SDK unoptimised with ~62 µs.

5.5.3 ADC Read Time

ADC read performance showed optimised Rust with unsafe features and optimised C SDK performing identically, slightly faster than optimised Rust HAL 3 µs. This shows that if performance is a factor, re-writing Rust to be unsafe can gain us increases. Unoptimised C SDK (4 µs) was faster than unoptimised Rust HAL (18 µs). This suggests a minor overhead in the Rust HAL ADC abstraction compared to direct register access or the optimised C SDK function, which becomes more pronounced without optimisation.

5.5.4 UART TX Rate

UART transmission performance revealed interesting trade-offs. When comparing optimised HAL/SDK approaches for transmitting a 10-byte payload, the C SDK was marginally faster than Rust HAL. Both significantly outperformed the optimised Rust Raw implementation, which likely suffered due to a busy-wait or polling loop implementation compared to potentially interrupt-driven or DMA-based approaches within the HAL/SDK layers. The unoptimised C SDK and unoptimised Rust HAL showed performance comparable to their optimised counterparts, suggesting UART transmission speed in these tests was perhaps less sensitive to

general compiler optimisation flags than CPU-bound tasks like GPIO toggling. The similarity between optimised C SDK, optimised Rust HAL, and their unoptimised counterparts suggests the bottleneck might lie within the UART hardware limitations or the specific implementation within the higher-level libraries rather than pure code execution speed for this short payload.

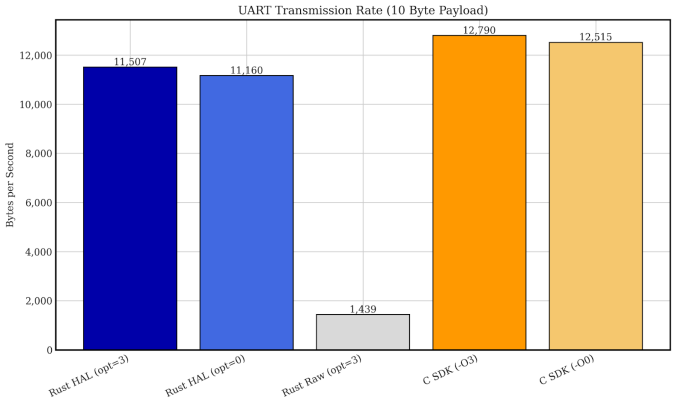

Figure 12: UART Transmission Rate Benchmarks

5.5.5 Interrupt Latency

Interrupt latency showed significant variation, particularly in the unoptimised C SDK which exhibited extremely high maximum latencies (>100,000 µs) at times. optimised C SDK also showed considerable variability and higher median latency compared to both optimised and unoptimised Rust HAL implementations. The optimised Rust HAL generally demonstrated the lowest and most consistent interrupt latencies in this specific test setup, although the unoptimised Rust HAL also performed reasonably well, albeit with slightly more spread than the optimised version. This suggests Rust's mechanisms, potentially related to its interrupt handling abstractions or lower static overhead, might offer advantages in achieving more predictable interrupt response times compared to the C SDK in this context, although the C SDK's high outliers warrant further investigation (possibly related to background tasks or specific SDK behaviour). The inability to get conclusive Rust Raw interrupt results was a limitation as our code did not work–sometimes rewriting for performance reasons leads to drawbacks as faced here.

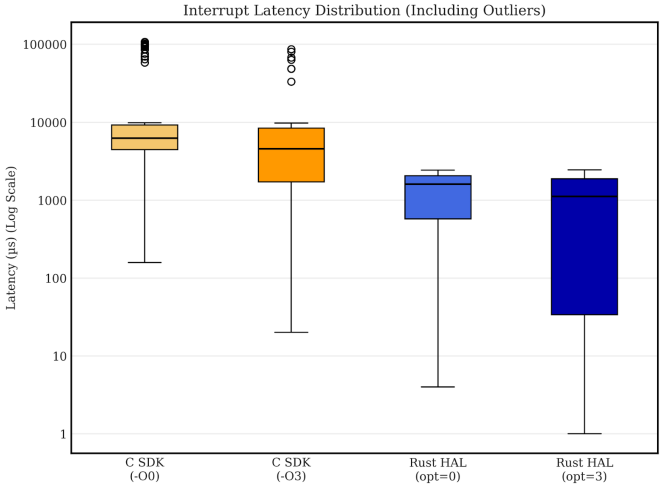

Figure 13: Interrupt Latency Distribution with Outliers

6 Conclusion

This dissertation confronted concerns regarding Rust's viability by investigating its practical application as a successor to C within constrained embedded systems. We finish by providing an executive summary and discussing future work and recommendations to make Rust viable in the future as a potential migration choice for C embedded projects. The investigation affirms that Rust is a viable and increasingly attractive language for embedded systems. It offers a pathway to significantly enhanced software robustness and security through its innovative approach to memory safety, often without compromising performance. While challenges related to the learning curve and ecosystem maturity persist, the progress is undeniable, positioning Rust as a strong candidate to redefine best practices in embedded development.

6.1 Summary

The findings confirm that Rust's core value—compile-time memory safety without sacrificing performance, does indeed hold significant merit in the embedded context. The ownership and borrowing system, a cornerstone of Rust, demonstrably mitigates entire classes of memory-related vulnerabilities endemic to C, such as buffer overflows and use-after-free errors, which was evident during the migration process where the Rust compiler rigorously enforced safer memory management practices.

In terms of performance, this investigation revealed that optimised Rust, particularly when leveraging its HAL abstractions, can achieve and, in certain benchmarks like GPIO toggle speed and interrupt latency consistency, even surpass the performance of optimised C SDK implementations. Static analysis further underscored Rust's advantages in static RAM efficiency, a critical factor in memory-constrained embedded environments, though optimised C maintained a slight edge in flash footprint for the bare-metal scenario. These results affirm that Rust’s "zero-cost abstractions" are largely effective when compiler optimisations are fully engaged.

However, the transition to Rust is not without its challenges. The language presents a steeper learning curve compared to C, primarily due to the paradigm shift required to master concepts like ownership, borrowing, and lifetimes. Furthermore, while Rust's tooling, centered around Cargo, offers a modern and integrated development experience, the broader embedded ecosystem, including HAL coverage and specialized debugging tools for all targets, is still maturing relative to C's decades-established infrastructure. The research questions posed at the outset have been addressed: Rust's safety mechanisms are indeed more robust; its performance is competitive and sometimes superior when optimised; it shows strong potential for meeting real-time requirements; and while its ecosystem is rapidly advancing, it currently trails C in overall maturity for the full spectrum of embedded targets.

6.2 Recommendations

Based on these findings, several suggestions and recommendations can be drawn:

1. **Educational Emphasis**: Educational institutions should consider integrating Rust into systems programming, embedded systems and programming courses. Its compile-time

enforcement of memory safety concepts offers a unique tool to instill safe coding practices from the outset, potentially producing engineers with a deeper understanding of issues that are often subtle and error-prone in C.

2. **Strategic Industry Adoption**: For new embedded projects, particularly those in domains like cybersecurity, automotive, aerospace, and medical devices where software reliability and security are paramount, Rust should be strongly considered as the primary development language. Its inherent safety features can significantly reduce the risk of costly and dangerous memory-related bugs. For organizations with extensive existing C/C++ codebases, a phased adoption strategy is recommended. This could involve using Rust for new, well-contained modules, leveraging Rust's FFI capabilities to integrate these modules with legacy C code, or targeting Rust for specific security-sensitive components.
3. **Continued Ecosystem Development**: The Rust community and commercial entities should continue to invest in maturing the embedded ecosystem. This includes enhancing HAL coverage for a wider range of microcontrollers, improving the robustness and usability of debugging tools like probe-rs, and developing more sophisticated static analysis and formal verification tools for no_std environments.
4. **For Developers Migrating from C**: Approach Rust with an open mind, prepared for a paradigm shift rather than just learning new syntax. Invest time in thoroughly understanding ownership, borrowing, and lifetimes, as these are fundamental. Leverage the official Rust documentation ("The Rust Programming Language," "The Embedded Rust Book," "The Rustonomicon") and community resources. Start with simpler projects or modules to build confidence before tackling complex systems. Embrace the Rust compiler's error messages; they are designed to be helpful guides towards writing correct and safe code. Minimise unsafe code–this is not the philosophy Rust undertakes.

6.3 Future Work

This project could explore several avenues to build upon these findings. A deeper investigation into Rust's `async/await` capabilities for concurrent operations within the morse code application could offer valuable comparisons against traditional interrupt-driven or polling approaches. A more extensive analysis of compilation times under various optimisation levels and project complexities would address a practical concern for development workflows. Profiling dynamic memory usage, particularly stack consumption, would offer a more complete runtime memory footprint comparison. Expanding the dynamic benchmarks to include more complex peripherals and application-level scenarios, potentially incorporating an RTOS for both C and Rust, would provide richer performance data. Furthermore, a rigorous investigation into the probe-rs issues and C SDK interrupt latency variability would be beneficial. Finally, as formal verification tools for Rust mature, applying them to the unsafe portions of the bare-metal code could offer stronger assertions about its correctness and safety.

7 Project Outcomes

This section provides a critical evaluation of how the project met its defined objectives as defined in section 1.2, alongside personal reflections on achievements and learnings.

7.1 Objective 1

Our implementation and migration of a bare-metal and morse transmission SDK/HAL project in C between RP2040 microcontrollers to Rust was met. The bare-metal application involving LED blinking and the passive buzzer activation via a button press was successfully implemented in both C and Rust on the RP2040. This involved writing custom boot sequences, vector tables, linker scripts, and direct peripheral control logic in both languages. Subsequently, a more complex morse code transmission and reception system was developed using the Pico SDK for C and the rp2040-hal for Rust. This system involved UART communication between two Pico devices, button input for morse code generation, LED/buzzer feedback, and I2C LCD output for decoded messages. While the C SDK version achieved full functionality, the Rust HAL version demonstrated core transmitter functionality and receiver initialization, though full end-to-end receiver validation with LCD output was hindered by debugging limitations with the available hardware setup. The migration process itself was a core part of achieving this objective, providing direct experience with the challenges and benefits of moving from C to Rust in an embedded context.

7.3 Objective 2

The project's controlled experiments using the SDK project to evaluate safety and performance was successfully achieved through two main experimental phases. Experiment 1 focused on static analysis of the generated binaries from both C (SDK and bare-metal) and Rust (HAL and bare-metal) implementations. Metrics collected included static RAM usage, flash memory size, and identification of the largest functions and static variables. This provided quantitative data on resource utilization. Experiment 2 involved dynamic analysis through micro-benchmarks executed on the RP2040, comparing the runtime performance of C (SDK) and Rust (HAL and raw register access) for operations such as GPIO toggling, PWM setup, ADC reads, UART transmission rates, and interrupt latency. These controlled experiments provided the empirical data needed to evaluate Rust's claims of zero-cost abstractions and its performance relative to C in specific embedded tasks.

7.3 Objective 3

A comprehensive comparison was conducted and detailed throughout Chapter 5. Technical aspects analyzed included memory models, Rust's ownership and borrowing system versus C's manual memory management, type systems and traits, array handling and bounds checking, concurrency approaches, error handling mechanisms, and macro systems. Non-technical aspects evaluated included community support and ecosystem maturity, development workflow and tooling, language maturity and stability, and adherence to standards and best practices. This comparison drew heavily on the practical experiences gained during both the bare-metal and

SDK/HAL implementations, allowing for a nuanced discussion of each language's strengths and weaknesses in different contexts.

7.4 Objective 4

The results of Experiment 1 showed Rust's advantage in static RAM efficiency, particularly with HAL abstractions consuming significantly less static RAM than C SDK builds. optimised Rust HAL also produced smaller flash binaries than the optimised C SDK. However, unoptimised Rust HAL builds could be larger, and bare-metal C achieved a slightly smaller flash footprint than bare-metal Rust in the specific test case. Experiment 2 indicated that optimised Rust HAL can be highly competitive with, and sometimes outperform, optimised C SDK in performance metrics like GPIO toggle speed and interrupt latency consistency. Challenges for Rust include its steeper learning curve and the fact that direct unsafe register access did not always yield the best performance compared to well-optimised HALs. Unoptimised Rust code showed notable performance drops in some areas like ADC read time compared to unoptimised C. The inherent safety provided by Rust's compiler, preventing many common C errors, stands as a significant advantage, though this often comes with the "challenge" of satisfying the borrow checker during development.

7.5 Objective 5

The concluding chapter synthesizes the findings from the entire project. It reiterates Rust's strong safety guarantees stemming from its ownership model, which inherently addresses many security vulnerabilities common in C related to memory mismanagement. Performance, as shown by the experiments, can be comparable or better than C when Rust code is optimised and leverages its abstraction mechanisms effectively, though it's not a universal guarantee across all scenarios or without optimisation. The project necessitated adopting Rust-idiomatic design patterns, such as using Result for error handling and leveraging traits for hardware abstraction, rather than directly translating C patterns. Lessons learned include the significant upfront time investment required for Rust, the helpfulness of its compiler messages, the current state of its embedded ecosystem (both its strengths and evolving areas), and the critical importance of understanding Rust's core concepts to write effective and safe embedded code. The feasibility of migrating from C to Rust is clear, but it requires a commitment to learning a new paradigm rather than simply a new syntax.

7.6 Objective 6

Recommendations were provided in the main conclusion. The feasibility of adopting Rust for embedded systems at scale is promising but depends on factors like project requirements (safety-criticality), team expertise and willingness to invest in training, and the maturity of Rust's ecosystem for specific hardware targets and required libraries. For new projects, especially those where memory safety is a high priority, Rust offers a compelling alternative to C. For existing large C codebases, a complete rewrite might be prohibitive, but incremental adoption through FFI or focusing Rust on new, critical modules could be a viable strategy. The growing industry support and the language's inherent strengths suggest that Rust's role in embedded systems will continue to expand.

7.7 Personal Objectives and Achievements

This project served as an invaluable learning experience, pushing knowledge into the practical application of two distinct programming languages in the challenging area of embedded systems

development. A primary personal objective was to gain a deep, hands-on understanding of low-level programming concepts, from bootloaders and interrupt handling to direct hardware register manipulation. Implementing these from scratch in both C and Rust was important for me to solidify this understanding. Another key objective was to critically evaluate Rust, a language I was less familiar with initially, against the industry stalwart C, specifically within the embedded context. This involved not only learning Rust's syntax but also grappling with its core philosophies of ownership, borrowing, and safety. Successfully migrating the project, navigating the compiler's demands, and ultimately getting functional code running on the RP2040 in Rust felt like a significant achievement. I also aimed to develop skills in systematic experimentation and data analysis, which were honed through the design and execution of the static and dynamic performance benchmarks. Overcoming the numerous technical hurdles, from linker script configurations and debugging tool issues to understanding complex compiler errors, fostered resilience and problem-solving skills. The process of structuring this dissertation, articulating complex technical details, and drawing evidence-based conclusions has also been a crucial part of my development as a researcher and technical communicator. While not all aspects of the Rust HAL receiver were fully realized due to time and hardware constraints, the depth of learning achieved in working towards that goal was substantial. This project has significantly enhanced my confidence in tackling complex low-level software challenges and provided a solid foundation for future work in embedded systems and systems programming.

8

Statement of Ethics

This project was conducted with a conscious awareness of the legal, social, ethical, and professional (LSEP) responsibilities inherent in computer technology research and development. The investigation centered on comparing programming languages for embedded systems, a field where software reliability and security have profound implications. A Self-Assessment for Governance and Ethics (SAGE) form has been completed and attached separately–our project does not require further ethical review by the University Ethics Committee (UEC).

8.1 BCS Code of Conduct

In line with the British Computer Society (BCS) Code of Conduct, this project endeavors to contribute positively to the IT field. The investigation into Rust as a safer alternative to C for embedded systems directly supports the principle of ensuring IT works for the benefit of society by promoting technologies that can lead to more secure and reliable systems. This research does not discriminate but rather seeks to inform the development community about tools that can enhance software quality for all users. All aspects of this project, from the initial research to the final reporting of results, have been conducted with professional integrity, ensuring that all claims are supported by the findings and that limitations are acknowledged. Full transparency has been maintained with the project supervisor, and all academic requirements of the University of Surrey have been met with due care and diligence. The research, by highlighting potential improvements in software development practices (e.g., memory safety in Rust), aims to bolster the reputation of the IT profession by demonstrating a commitment to addressing known issues and advancing technological capabilities responsibly.

8.2 Do No Harm

The insights derived from comparing Rust and C are intended to equip developers and organizations with knowledge that can lead to the creation of more robust and less error-prone embedded software. While the project itself (a Morse code application) is not safety-critical, the principles and language features analyzed, especially Rust's memory safety guarantees, are directly relevant to preventing software-induced harm in systems where failures could have severe consequences. This research does not involve the creation or dissemination of tools or techniques that could be used to cause harm; on the contrary, it promotes approaches to mitigate software-related risks. All software development and hardware interaction were conducted in a controlled environment on personally managed or institutionally provided equipment, strictly adhering to the Computer Misuse Act (CMA), with no unauthorized access to any systems or data.

8.3 Informed Consent

The nature of this research did not involve human participants in any capacity. The project focused on the technical attributes of programming languages, software performance on microcontrollers, and analysis of publicly available technical documentation. Consequently, the

ethical considerations related to informed consent from human subjects are not applicable to this work.

8.4 Confidentiality of Data

This project did not collect, handle or store personal or confidential data. The information processed consisted entirely of technical specifications from datasheets, open-source code, compiler outputs, performance measurements generated from the experimental setups, and publicly accessible academic and technical literature. Therefore, the detailed provisions of the Data Protection Act (DPA) concerning the safeguarding of personal data, while acknowledged as critical in other contexts, were not directly invoked by the data involved in this specific research.

8.5 Social Responsibilities

The project recognizes its social responsibility by aiming to contribute to the broader goal of enhancing the security and reliability of embedded systems. These systems are increasingly ubiquitous, forming the backbone of critical infrastructure, everyday consumer products, and emerging technologies like the Internet of Things (IoT). Software vulnerabilities or failures in such systems can lead to significant societal disruption, financial loss, or even physical harm. By evaluating and highlighting programming language features that can lead to inherently safer code such as Rust's memory safety, this research seeks to promote development practices that reduce these risks. The project's findings are intended to empower developers to make more informed choices that can lead to more trustworthy technology. While any technical information could theoretically be misinterpreted or misused, the explicit aim here is constructive: to improve software quality and resilience. The comparative performance data, for instance, is presented to inform engineering trade-offs, not to identify or enable exploitation of system weaknesses.

8.6 Intellectual Property

In terms of Intellectual Property (IP), our project adheres to the University of Surrey's Code of Practice in relation to Patenting and Exploitation of Inventions. All original source code developed for the C and Rust and implementations of the morse code system and the benchmarking tools, and the analytical content of this dissertation, constitutes my own intellectual property, created in fulfillment of academic requirements and subject to University regulations. It is not anticipated that any patentable or immediately commercially exploitable IP has been generated through this work. All third-party software, libraries such as thePico SDK, rp2040-hal and various Rust crates and development tools used within this project are either open-source, utilized in accordance with their respective licensing terms, or are standard academic tools. No confidential or proprietary information from external parties has been used or disclosed.

8.2 Plagiarism

Academic integrity has been an important concern throughout the conception, execution, and documentation of this project. All sources of information, including technical manuals, academic papers, online documentation, and existing code that may have informed the development process, have been diligently acknowledged and appropriately cited within this dissertation. The University of Surrey's policies regarding plagiarism have been strictly observed, ensuring that all presented work is original, and that any contributions or ideas from others are given due credit.

References

[1] Jansen, Paul. “TIOBE Index.” TIOBE, https://www.tiobe.com/tiobe-index/. Accessed 12 May 2025.

[2] Catalin Cimpanu. “Microsoft: 70 percent of all security bugs are memory safety issues.” ZDNet, *Microsoft: 70 percent of all security bugs are memory safety issues*, 11 February 2019, https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/ Accessed 20 January 2025. [Online]

[3] Companies Ready Insurance Claims Over CrowdStrike Outage.” *Business Insider*, 22 July 2024,https://www.businessinsider.com/businesses-claiming-losses-crowdstrike-outage-insurance-billions-losses-cyber-policies-2024-7. Accessed 26 January 2025. [Online]

[4] N. Borgsmüller, “The Rust Programming Language for Embedded Software Development,” B.S. thesis, Dept. Comput. Sci., Hochschule für Angewandte Wissenschaften Hamburg, Hamburg, Germany, 2020 (Submitted Jan. 15, 2021). Accessed: March 14, 2025. [Online]. Available: https://opus4.kobv.de/opus4-haw/files/786/I000819827Thesis.pdf

[5] Weston, David. “External Technical Root Cause Analysis — Channel File 291.” CrowdStrike, 6 August 2024, https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf. Accessed 26 January 2025. [Online]

[6] “TRACTOR: Translating All C to Rust.” DARPA, https://www.darpa.mil/program/translating-all-c-to-rust. Accessed 26 January 2025. [Online]

[7] Sharma, Ayushi, et al. “[2311.05063] Rust for Embedded Systems: Current State, Challenges and Open Problems (Extended Report).” arXiv, 8 November 2023, https://arxiv.org/abs/2311.05063. Accessed 26 January 2025. [Online]

[8] Shashank Sharma, Ayushi Sharma, and Aravind Machiry. 2024. Aunor: Converting Rust crates to [no_std] at scale. In Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy (CODASPY '24). Association for Computing Machinery, New York, NY, USA, 163–165. https://doi.org/10.1145/3626232.3658640. Accessed 26 January 2025 [Online]

[9] embedded-hal. “embedded-hal.” embedded-hal, HAL Team, 2025, [Online] https://github.com/rust-embedded/embedded-hal. Accessed 27 January 2025. [Online]

[10] The Rust Project Developers. “How fast is Rust?” How fast is Rust?, 2025, https://doc.rust-lang.org/1.0.0/complement-lang-faq.html#how-fast-is-rust. Accessed 27 January 2025. [Online]

[11] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: toward per-Core 100-Gbps networking. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21). Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3445814.3446724 [Online]

[12] The Rust Developers. “Type Layout.” doc.rust-lang.org, 2015, https://doc.rust-lang.org/reference/type-layout.html. Accessed 28 January 2025. [Online]

[13] Bugden, William, and Ayman Alahmar. “Rust: The Programming Language for Safety and Performance.” arXiv, 11 June 2022, https://arxiv.org/pdf/2206.05503. Accessed 26 January 2025. [Online]

[14] “The Cost of Branching.” Algorithmica, https://en.algorithmica.org/hpc/pipelining/branching/. Accessed 26 January 2025. [Online]

[15] “How much Rust's bounds actually cost | Readyset.” Readyset, https://readyset.io/blog/bounds-checks. Accessed 26 January 2025. [Online]

[16] Balakrishnan, Ashwin Kumar, and Gaurav Nattanmai Ganesh. “Modern C++ and Rust in embedded memory-constrained systems.” odr.chalmers.se, 2022, https://odr.chalmers.se/server/api/core/bitstreams/1568a7d6-8b60-419d-acbc-40abe5858111/content. Accessed 28 January 2025 [Online]

[17] A. Perez, “Rust and C++ performance on the Algorithmic Lovasz Local Lemma,” Course Project, Stanford University, USA, 2017. Accessed: 28 January 2025. [Online]. Available: https://stanford-cs242.github.io/f17/assets/projects/2017/aperez8.pdf

[18] Wandalen, "Size of the executable binary file of an application," The Rust Programming Language Forum, July 10, 2021. Accessed: 28 January 2025. [Online]. Available: https://users.rust-lang.org/t/size-of-the-executable-binary-file-of-an-application/62160
[19] gbip, "Reduce binary size for embedded," The Rust Programming Language Forum, Sept. 27, 2018. Accessed: 29 January 2025. [Online]. Available: https://users.rust-lang.org/t/reduce-binary-size-for-embedded/20804
[20] Hudson Ayers, Evan Laufer, Paul Mure, Jaehyeon Park, Eduardo Rodelo, Thea Rossman, Andrey Pronin, Philip Levis, and Johnathan Van Why. 2022. Tighten rust's belt: shrinking embedded Rust binaries. In Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2022). Association for Computing Machinery, New York, NY, USA, 121–132. Accessed: 29 January 2025. . Available: https://doi.org/10.1145/3519941.3535075 [Online]
[21] johnthagen. min-sized-rust.. GitHub. 2024. Accessed: 29 January 2025. [Online]. Available: https://github.com/johnthagen/min-sized-rust
[22] "Unsafe Rust," The Rust Programming Language Book. Accessed: 29 January 2025. [Online]. Available: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.24
[23] https://arxiv.org/pdf/2404.02230v2
[24] N. R. Foroushaani and B. Jacobs. Modular Formal Verification Of Rust Programs With Unsafe Blocks, Accessed: 29 January 2025. [Online]. Available: https://arxiv.org/pdf/2212.12976
[25] rust-lang/miri: An interpreter for Rust's mid-level ... - GitHub, Accessed: 29 January 2025, https://github.com/rust-lang/miri
[26] A. Reid, L. Church, S. Flur, S. de Haas, M. Johnson, and B. Laurie, "Towards making formal methods normal: meeting developers where they are," in Proc. HATRA 2020: Human Aspects of Types and Reasoning Assistants, Chicago, IL, Nov. 15-20, 2020, pp. 1-10. Accessed: 29 January 2025. [Online]. Available: https://alastairreid.github.io/papers/hatra2020.pdf
[27] K. Hu, L. Wang, C. Mo and B. Jiang, "Work-in-Progress: Unishyper, A Reliable Rust-based Unikernel for Embedded Scenarios," 2023 International Conference on Embedded Software (EMSOFT), Hamburg, Germany, 2023, pp. 1-2. Accessed: 29 January 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10316382
[28] "DWARF," Wikipedia, May 1, 2025. Accessed: 30 January 2025. [Online]. Available: https://en.wikipedia.org/wiki/DWARF
[29] B. Cantrill, "Unikernels are unfit for production," Triton DataCenter Blog, Jan. 8, 2024. Accessed: 30 January 2025. [Online]. Available: https://www.tritondatacenter.com/blog/unikernels-are-unfit-for-production
[30] Raspberry Pi Ltd, "RP2040 Datasheet," Build 3184e62-clean, Feb. 20, 2025. Accessed: 1 February 2025. [Online]. Available: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf
[31] "Introduction to Raspberry Pi Pico guide," Raspberry Pi Projects, Jan. 25, 2021. Accessed: 1 February 2025. [Online]. Available: https://projects.raspberrypi.org/en/projects/introduction-to-the-pico/9.29
[32] "I²C LCD1602," SunFounder Wiki, Nov. 15, 2023. Accessed: 1 February 2025. [Online]. Available: http://wiki.sunfounder.cc/index.php?title=I%C2%B2C_LCD1602
[33] S. Hymel, "Raspberry Pi Pico and RP2040 - C/C++ Part 2 Debugging with VS Code," Maker.io, May 17, 2021. Accessed: 1 February 2025. [Online]. Available: https://www.digikey.co.uk/en/maker/projects/raspberry-pi-pico-and-rp2040-cc-part-2-debugging-with-vs-code/470abc7efb07432b82c95f6f67f184c00
[34] S. Hymel, "Raspberry Pi Pico and RP2040 - C/C++ Part 1: Blink and VS Code," Maker.io, May 10, 2021. Accessed: 1 February 2025. [Online]. Available: https://www.digikey.co.uk/en/maker/projects/raspberry-pi-pico-and-rp2040-cc-part-1-blink-and-vs-code/7102fb8bca95452e9df6150f39ae8422
[35] S. Hymel, "How to Set Up Raspberry Pi Pico C/C++ Toolchain on Windows with VS Code," Shawn Hymel, Apr. 9, 2021. Accessed: 1 February 2025. [Online]. Available: https://shawnhymel.com/2096/how-to-set-up-raspberry-pi-pico-c-c-toolchain-on-windows-with-vs-code/
[36] V. H. Adams, "RP2040 boot sequence," vanhunteradams.com. Accessed: 5 February 2025. [Online]. Available: https://vanhunteradams.com/Pico/Bootloader/Boot_sequence.html.l
[37] vxj9800. bareMetalRP2040/01_bootupBlinky.. GitHub. Accessed: 5 February 2025. [Online]. Available: https://github.com/vxj9800/bareMetalRP2040/tree/main/01_bootupBlinky.33

[38] vxj9800. bareMetalRP2040/tree/main GitHub. Accessed: 9 February 2025. [Online]. Available: https://github.com/vxj9800/bareMetalRP2040/tree/main
[39] “RP2040 Datasheet: A microcontroller by Raspberry Pi.” Raspberry Pi Datasheets, https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf. Accessed 12 May 2025.
[40] ARM Limited, "ARM® v6-M Architecture Reference Manual," ARM DDI 0419E, June 29, 2018. 8 February 2025: Accessed: 12 February, 2025. [Online]. Available: https://cdn.hackaday.io/files/1770827576276288/DDI0419E_armv6m_arm.pdf
[41] vxj9800. bareMetalRP2040/02_xipAndFlash.. GitHub. Accessed: 9 February 2025. [Online]. Available: https://github.com/vxj9800/bareMetalRP2040/tree/main/02_xipAndFlash
[42] vxj9800. bareMetalRP2040/03_vectorTable.. GitHub. Accessed: 9 February 2025. [Online]. Available: https://github.com/vxj9800/bareMetalRP2040/tree/main/03_vectorTable
[43] https://cdn.hackaday.io/files/1770827576276288/DDI0419E_armv6m_arm.pdf#page=209
[44] "RFC 2603: Rust Symbol Name Mangling v0," rust-lang.github.io, Nov. 27, 2018. Accessed: 14 February 2025. [Online]. Available: https://rust-lang.github.io/rfcs/2603-rust-symbol-name-mangling-v0.html
[45] https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf#page=604
[46] Winbond, "W25Q80DV, 3V 8M-BIT SERIAL FLASH MEMORY WITH DUAL AND QUAD SPI," Rev. H, Oct. 2, 2015 / July 21, 2015. Accessed: 19 February 2025. [Online]. Available: http://www.alldatasheet.com/datasheet-pdf/pdf/932085/WINBOND/25Q80DVNIG.html
[47] https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf#page=179
[48] "addr_of in core::ptr - Rust," Rust Lang Documentation, Version 1.88.0-beta.1, May 9, 2025. Accessed: 24 February 2025. [Online]. Available: https://doc.rust-lang.org/beta/core/ptr/macro.addr_of.htm
[49] "IEEE Xplore Digital Library." [Online], "Towards even smaller structs," Rust Internals. Accessed: 27 February 2025. [Online]. Available: https://internals.rust-lang.org/t/towards-even-smaller-structs
[50] P. Saurav, "Why is taking the address of a temporary illegal?," Stack Overflow, Nov. 29, 2010. Accessed: 1 March 2025. [Online]. Available: https://stackoverflow.com/questions/4301179/why-is-taking-the-address-of-a-temporary-illegal
[51] "Send and Sync," The Rustonomicon. Accessed: 5 March 2025. [Online]. Available: https://doc.rust-lang.org/nomicon/send-and-sync.html
[52] https://cdn.hackaday.io/files/1770827576276288/DDI0419E_armv6m_arm.pdf#page=192
[53] G. Owen, "Understanding #[derive(Clone)] in Rust," Stegosaurus Dormant, Aug. 11, 2021. Accessed: 8 March 2025. [Online]. Available: https://stegosaurusdormant.com/understanding-derive-clone/
[54] "Unions - Unsafe Code Guidelines Reference," Rust Lang Unsafe Code Guidelines. Accessed: 11 March 2025. [Online]. Available: https://rust-lang.github.io/unsafe-code-guidelines/layout/unions.html
[55] “Unsafe Rust - The Rust Programming Language.” Rust Documentation, https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html. Accessed 11 March2025.
[56] system. “LCD just blinks.” forum.arduino.cc, 2013, https://forum.arduino.cc/t/lcd-just-blinks/159808/1. Accessed 12 March 2025.
[57] raspberrypi. pico-examples/i2c/lcd_1602_i2c/lcd_1602_i2c.c.. GitHub. Last commit: approx. Sept. 2024 (8 months prior to May 2025). Accessed: 11 March 2025 [Online]. Available: https://github.com/raspberrypi/pico-examples/blob/master/i2c/lcd_1602_i2c/lcd_1602_i2c.c
[58] rp-rs. rp2040-boot2 (Version v0.3.0).. GitHub. May 5, 2023. Accessed: 13 March 2025. [Online]. Available: https://github.com/rp-rs/rp2040-boot2
[59] "Generic Types, Traits, and Lifetimes," The Rust Programming Language Book. Accessed: 11 March, 2025. [Online]. Available: https://doc.rust-lang.org/book/ch10-00-generics.html
[60] https://cdn.hackaday.io/files/1770827576276288/DDI0419E_armv6m_arm.pdf#page=207
[61] T. Verbeure, "Semihosting, your PC as Console of an Embedded RISC-V CPU," Electronics etc…, Dec. 30, 2021. Accessed: 14 March, 2025. [Online]. Available: https://tomverbeure.github.io/2021/12/30/Semihosting-on-RISCV.html
[62] SEGGER Microcontroller GmbH & Co KG, "J-Link RTT – Real Time Transfer," Segger. Accessed: 14 March, 2025. [Online]. Available: https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/

[63] P. E. McKenney (paulmck), "What Memory Model Should the Rust Language Use?," LiveJournal, Nov. 3, 2021. Accessed: March 14, 2025. [Online]. Available: https://paulmck.livejournal.com/66175.html
[64] europa42, "What are the implications of not having a memory model?," Reddit r/rust, approx. 2018. Accessed: March 14, 2025. [Online]. Available: https://www.reddit.com/r/rust/comments/b664x5/what_are_the_implications_of_not_having_a_memory/
[65] "Memory model - The Rust Reference," Rust Documentation. Accessed: March 14, 2025. [Online]. Available: https://doc.rust-lang.org/reference/memory-model.html
[66] surban, "Is read_volatile on uninitialized memory really undefined behavior?," GitHub Issue #2807, rust-lang/miri, Mar. 8, 2023. Accessed: March 14, 2025. [Online]. Available: https://github.com/rust-lang/miri/issues/2807
[67] RalfJung, "Pointers Are Complicated, or: What's in a Byte?," Rust Internals, July 24, 2018. Accessed: March 14, 2025. [Online]. Available: https://internals.rust-lang.org/t/pointers-are-complicated-or-whats-in-a-byte/8045
[68] ralfj.de, "Pointers Are Complicated III, or: Pointer-integer casts exposed," ralfj.de blog, Apr. 11, 2022. Accessed: March 14, 2025. [Online]. Available: https://www.ralfj.de/blog/2022/04/11/provenance-exposed.html
[69] M. J. Batty, "The C11 and C++11 Concurrency Model," Ph.D. dissertation, Computer Laboratory, Univ. of Cambridge, Cambridge, UK, 2014. Accessed: March 18, 2025. [Online]. Available: https://www.cs.kent.ac.uk/people/staff/mjb211/docs/toc.pdf
[70] kornel, "Error: pointers cannot be cast to integers during const eval," The Rust Programming Language Forum, Feb. 9, 2022. Accessed: March 18, 2025. [Online]. Available: https://users.rust-lang.org/t/error-pointers-cannot-be-cast-to-integers-during-const-eval/71589/2
[71] "What is Ownership?," The Rust Programming Language Book. Accessed: March 19, 2025. [Online]. Available: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
[72] "References and Borrowing," The Rust Programming Language Book. Accessed: March 22, 2025. [Online]. Available: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html
[73] https://www.researchgate.net/figure/Ownership-and-borrowing_fig1_335649678
[74] "std::option - Rust," Rust Lang Documentation, Version 1.86.0, Mar. 31, 2025. Accessed: March 23, 2025. [Online]. Available: https://doc.rust-lang.org/std/option/
[75] "Generic Linked List in C," GeeksforGeeks, Aug. 27, 2024. Accessed: March 2312, 2025. [Online]. Available: https://www.geeksforgeeks.org/generic-linked-list-in-c-2/
[76] "Validating References with Lifetimes," The Rust Programming Language Book. Accessed: March 24, 2025. [Online]. Available: https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html
[77] G. Hunter, "Running Rust on Microcontrollers," mbedded.ninja, Nov. 12, 2022. Accessed: May 12, 2025. [Online]. Available: https://blog.mbedded.ninja/programming/languages/rust/running-rust-on-microcontrollers/
[78] "Happened-before," Wikipedia, Feb. 25, 2025. Accessed: May 3, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Happened-before
[79] japaric, and github:rust-embedded:hal. nb (Version 1.1.0).. crates.io. Approx. Oct. 2021. Accessed: May 3, 2025. [Online]. Available: https://docs.rs/nb/latest/nb/index.html
[80] Hunter, Geoffrey. "Running Rust on Microcontrollers." blog.mbedded.ninja, 12 November 2022, https://blog.mbedded.ninja/programming/languages/rust/running-rust-on-microcontrollers/#error-handling. Accessed 4 May 2025. [Online]
[81] "std::result - Rust," Rust Lang Documentation, Version 1.86.0, Mar. 31, 2025. Accessed: May 6, 2025. [Online]. Available: https://doc.rust-lang.org/std/result/
[82] "macro_rules! - Rust By Example," Rust By Example. Accessed: May 8, 2025. [Online]. Available: https://doc.rust-lang.org/rust-by-example/macros.html
[83] "MISRA C," Wikipedia, Jan. 28, 2025. Accessed: May 8, 2025. [Online]. Available: https://en.wikipedia.org/wiki/MISRA_C
[84] The bindgen User Guide," rust-lang.github.io. Accessed: May 10, 2025. [Online]. Available: https://rust-lang.github.io/rust-bindgen/
[85] "rp-rs/rp-hal: A Rust Embedded-HAL for the rp series microcontrollers." GitHub, https://github.com/rp-rs/rp-hal. Accessed 13 May 2025

Appendix A | Measurements

Rust Functions	Size (Bytes)	C Functions	Size (Bytes)
main	156	main	216
resetHandler	80	vector	120
bootStage2	80	resetHandler	92
ioIrqBank0	52	gpio_set_pulls	84
ioIrqBank0Handler	12	bootStage2	80
defaultHandler	8	ioIrqBank0	80

Table 1: Largest Functions in the Bare-Metal Rust and C Binaries after Optimisations

Rust Functions	Size (Bytes)	C Functions	Size (Bytes)
receiver::__cortex_m_rt_main	2576	_vsnprintf	2312
compiler_builtins::int::specialized_div_rem::u64_div_rem	972	alarm_pool_irq_handler	1528
core::str::slice_error_fail_rt	608	main	1004
core::fmt::Formatter::pad_integral	518	process_control_request	1504
core::fmt::write	318	_ftoa	1172
core::str::count::do_count_chars	420	_etoa	1428

Table 2: Largest Functions in the HAL Rust and C SDK Receiver Binaries after Optimisations

Rust Variables	Size (Bytes)	C Variables	Size (Bytes)
rp2040_hal::rom_data::memset4::ptr::CACHED_PTR	2	hw_endpoints	1024
rp2040_hal::rom_data::memcpy::ptr::CACHED_PTR	2	_vendord_itf	588
rp2040_hal::rom_data::clz32::ptr::CACHED_PTR	2	default_alarm_pool_entries	384
DEVICE_PERIPHERALS	1	sf_table	256
cortex_m::peripheral::TAKEN	1	sd_table	256
cortex_m_semihosting::export::HSTDOUT	8	display_message	128

Table 3: Largest Static Variables in the HAL Rust and C SDK Binaries after Optimisations

```
pub static mut VECTOR_TABLE: [u32; 48] = [
                ...
        resetHandler as usize,    // Reset Handler
        NMIHandler as usize,     // NMI Handler
... ];
```

```
error: pointers cannot be cast to integers during const eval
       |
     122       resetHandler as usize,
       |       ^^^^^^^^^^^^^^^^^^^^^
= note: at compile-time, pointers do not have an integer value
= note: avoiding this restriction via `transmute`, `union`, or raw pointers leads to
compile-time undefined behavior
```

```
pub static mut VECTOR_TABLE: [u32; 48] = [
                            ...
        unsafe { core::ptr::addr_of!(resetHandler) as usize },   // Reset Handler
        unsafe { core::ptr::addr_of!(NMIHandler) as usize },    // NMI Handler
... ];
```

```
error[E0745]: cannot take address of a temporary
       |
       123     unsafe { core::ptr::addr_of!(resetHandler) as usize },
       |                                     ^^^^^^^^^^ temporary value
```

```
pub static VECTOR_TABLE: [VectorTableEntry; 48] = unsafe {
        let mut table: [VectorTableEntry; 48] = [VectorTableEntry { reserved: 0 }; 48];
        // Initial Stack Pointer
        table[0] = VectorTableEntry { stack_top: ptr::addr_of!(_sstack) };
```

```
error[E0277]: *const u32 cannot be shared between threads safely
error[E0277]: the trait bound VectorTableEntry: core::marker::Copy is not satisfied
       |
       117   pub static VECTOR_TABLE: [VectorTableEntry; 48] = unsafe { ...
       |                              ^^^^^^^^^^^^^^^^^^^^^^
= help: within [VectorTableEntry; 48], the trait Sync is not implemented for *const u32
= note: required because it appears within the type VectorTableEntry
= note: the Copy trait is required because this value will be copied for each element of
the array
= help: consider annotating VectorTableEntry with #[derive(Copy)]
```

Listing 10: `VECTOR_TABLE` Implementations with Associated Stack Traces

```
(gdb) monitor reset init
[rp2040.core0] halted due to debug-request, current mode: Thread
xPSR: 0x1000000 pc: 0x000000ea msp: 0x20041f00, semihosting fileio
[rp2040.core1] halted due to debug-request, current mode: Thread
xPSR: 0x1000000 pc: 0x000000ea msp: 0x20041f00
(gdb) load
Loading section .boot2, size 0x200 lma 0x10000000
Loading section .vector_table, size 0xc0 lma 0x10000200
Loading section .text, size 0x10f18 lma 0x100002c0
Loading section .rodata, size 0x25e4 lma 0x100111d8
Start address 0x100002c0, load size 79804
```

```
Transfer rate: 26 KB/sec, 9975 bytes/write.
(gdb) info threads
  Id   Target Id  Frame
* 1    Thread 1 "rp2040.core0" (Name: rp2040.core0, state: debug-request)
0×100002c0 in Reset ()
  2    Thread 2 "rp2040.core1" (Name: rp2040.core1, state: debug-request)
0×000000ea in ?? ()
(gdb) continue
Continuing.

Thread 1 "rp2040.core0" received signal SIGINT, Interrupt.
0×10008db4 in rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>::tx_fifo_full<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller> (self=0×20041eb4) at
C:\Users\gokua\.cargo\registry\src\index.crates.io-1949cf8c6b5b557f\rp2040-hal-0
.10.2\src/i2c.rs:350
350          }
(gdb) monitor mdw 0×40034018
0×40034018: 00000187
(gdb) monitor mdw 0×40034024
0×40034024: 00000043
(gdb) info threads
  Id   Target Id                                                          Frame
* 1    Thread 1 "rp2040.core0" (Name: rp2040.core0, state: debug-request)
0×10008db4 in rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>::tx_fifo_full<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller> (self=0×20041eb4)
      at
C:\Users\gokua\.cargo\registry\src\index.crates.io-1949cf8c6b5b557f\rp2040-hal-0
.10.2\src/i2c.rs:350
  2    Thread 2 "rp2040.core1" (Name: rp2040.core1, state: debug-request)
0×00000184 in ?? ()
(gdb) bt
#0  0×10008db4 in rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>::tx_fifo_full<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
```

```
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller> (self=0×20041eb4) at
C:\Users\gokua\.cargo\registry\src\index.crates.io-1949cf8c6b5b557f\rp2040-hal-0
.10.2\src/i2c.rs:350
#1  0×10008bd6 in rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>::write_internal<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
core::iter::adapters::cloned::Cloned<core::slice::iter::Iter<u8>>>
(self=0×20041eb4, first_transaction=true, bytes=... , do_stop=true)
      at
C:\Users\gokua\.cargo\registry\src\index.crates.io-1949cf8c6b5b557f\rp2040-hal-0
.10.2\src\i2c/controller.rs:275
#2  0×10008904 in rp2040_hal::i2c::controller::{impl#5}::write<u8,
rp2040_pac::I2C0, (rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>)>
(self=0×20041eb4, addr=39, tx=... )
      at
C:\Users\gokua\.cargo\registry\src\index.crates.io-1949cf8c6b5b557f\rp2040-hal-0
.10.2\src\i2c/controller.rs:441
--Type <RET> for more, q to quit, c to continue without paging--
#3  0×10008890 in
receiver::Receiver<rp2040_hal::uart::peripheral::UartPeripheral<rp2040_hal::uart
::utils::Enabled, rp2040_pac::UART0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio0,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio1,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>)>,
rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>>::lcd_write_byte<rp2040_hal::uart::peripheral::Uart
Peripheral<rp2040_hal::uart::utils::Enabled, rp2040_pac::UART0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio0,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio1,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>)>,
rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>> (
      self=0×20041e28, byte_value=141) at src\bin/receiver.rs:112
#4  0×10008840 in
receiver::Receiver<rp2040_hal::uart::peripheral::UartPeripheral<rp2040_hal::uart
::utils::Enabled, rp2040_pac::UART0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio0,
```

```
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio1,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>)>,
rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>>::lcd_pulse_enable<rp2040_hal::uart::peripheral::Ua
rtPeripheral<rp2040_hal::uart--Type <RET> for more, q to quit, c to continue
without paging--
::utils::Enabled, rp2040_pac::UART0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio0,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio1,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>)>,
rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>> (
       self=0×20041e28, data=137) at src\bin/receiver.rs:116
#5  0×1000881c in
receiver::Receiver<rp2040_hal::uart::peripheral::UartPeripheral<rp2040_hal::uart
::utils::Enabled, rp2040_pac::UART0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio0,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio1,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>)>,
rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>>::lcd_data<rp2040_hal::uart::peripheral::UartPeriph
eral<rp2040_hal::uart::utils::Enabled, rp2040_pac::UART0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio0,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio1,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>)>,
rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>> (self=0×20041e28,
       data=72) at src\bin/receiver.rs:142
--Type <RET> for more, q to quit, c to continue without paging--
#6  0×100082fe in
receiver::Receiver<rp2040_hal::uart::peripheral::UartPeripheral<rp2040_hal::uart
::utils::Enabled, rp2040_pac::UART0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio0,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio1,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>)>,
rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
```

```
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>>::lcd_print<rp2040_hal::uart::peripheral::UartPerip
heral<rp2040_hal::uart::utils::Enabled, rp2040_pac::UART0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio0,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio1,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>)>,
rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>> (self=0x20041e28,
      text=...) at src\bin/receiver.rs:199
#7  0x10007ab0 in
receiver::Receiver<rp2040_hal::uart::peripheral::UartPeripheral<rp2040_hal::uart
::utils::Enabled, rp2040_pac::UART0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio0,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio1,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>)>,
rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>>::init<rp2040_hal::uart::peripheral::UartPeripheral
<rp2040_hal::uart::utils::Ena--Type <RET> for more, q to quit, c to continue
without paging--
bled, rp2040_pac::UART0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio0,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio1,
rp2040_hal::gpio::func::FunctionUart, rp2040_hal::gpio::pull::PullDown>)>,
rp2040_hal::i2c::I2C<rp2040_pac::I2C0,
(rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio4,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>,
rp2040_hal::gpio::Pin<rp2040_hal::gpio::pin::bank0::Gpio5,
rp2040_hal::gpio::func::FunctionI2c, rp2040_hal::gpio::pull::PullUp>),
rp2040_hal::i2c::Controller>> (self=0x20041e28)
      at src\bin/receiver.rs:103
#8  0x1000792c in receiver::__cortex_m_rt_main () at src\bin/receiver.rs:469
#9  0x10007792 in receiver::__cortex_m_rt_main_trampoline () at
src\bin/receiver.rs:412
```

Listing 22: `info threads` and `bt`