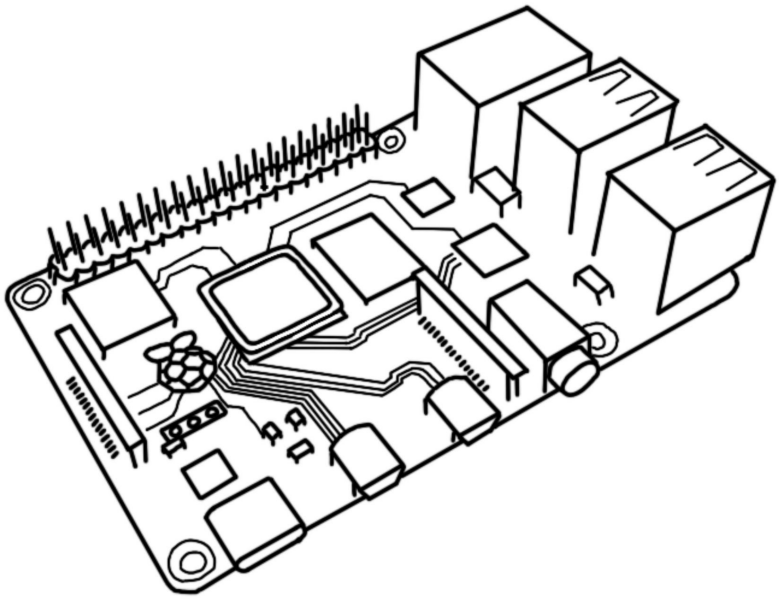
Robotics Research Lab
Department of Computer Science
University of Kaiserslautern-Landau

# Bachelor Thesis



# Closed-Loop Control of the Series Elastic Actuator

Max Erdelmeier

Thursday 22nd August, 2024

Bachelor Thesis

# Closed-Loop Control of the Series Elastic Actuator

Robotics Research Lab
Department of Computer Science
University of Kaiserslautern-Landau

Max Erdelmeier

**Day of issue** : 1. April 2024
**Day of release** : Thursday 22nd August, 2024

**Reviewer** : Prof. Dr. Karsten Berns
**Supervisor** : Oleksandr Sivak

Hereby I declare that I have self-dependently composed the Bachelor Thesis at hand. The sources and additives used have been marked in the text and are exhaustively given in the bibliography.

Thursday 22nd August, 2024 – Kaiserslautern

(Max Erdelmeier)

# Preface

This thesis is a wonderful opportunity to thank the people who've helped me both writing it as well as on my way to it.

First, I want to thank Prof. Karsten Berns for providing me the opportunity to write about a topic that I am deeply passionate about, that is not a given. Second, thanks to my supervisor Oleksandr Sivak, who helped me both with the bachelor thesis and the bachelor seminar.

To my parents, who have always supported me in every way, I just want to say I love with all my heart and I'm proud to be your son.

There are also quite a few people I would consider mentors who influenced in a big way who I am today. While some of them may never read this, I still want to give my thanks to them: Harald Pister, Henrik Roes and Clemens Schroeder

# Abstract


## English Version

This thesis presents a flexible motor controller architecture implmented on a Raspberry Pi, with a focus on achieving both high performance and modularity in control algorithm choice. Two primary approaches are explored: a bare-metal implementation without an operating system, and a Linux kernel-based solution. For the Linux-based approach, two optimization strategies are investigated to minimize execution times and enhance system responsiveness.

All implementations are developed using the Rust programming language, a modern systems language that shows promise in embedded systems due to its safety features and performance capabilities. Special focus is given to the intercompatibility of Rust and C++, which is critical to the architecture of the bare-metal version. The performance of the implementations is compared by measuring the iteration times of the control loop, finding that the bare-metal version is in most cases better suited to this application.

## German Version

Wir stellen eine auf einem Raspberry Pi implementierte Motorsteurung vor. Der Regelungsalgorithmus selbst soll dabei austauschbar sein und wird für alle Beispielprogramme durch einen PID-Regler implementiert. Ziel ist es eine möglichst performante Implementierung zu finden, sowohl was die durchschnittlichen Reaktionszeiten, also auch die Worst-Case Laufzeiten angeht. Um das zu erreichen, vergleichen wir zwei grundlegend verschiedene Ansätze, einmal ein Bare-metal Programm und einmal eines das unter Linux läuft. Diese Implementierung nutzen wir um die Eignung der Programmiersprache Rust für embedded Programme zu untersuchen, insbesondere mit Hinblick auf die interkompatibilität mit C.

# Contents

# 1. Introduction

## 1.1 Closed Loop Control

When addressing motors in a robot there is the challenge of how to control it. The reason why this is a problem is simple; The motor itself is a simple device that only reacts to strength and direction of electrical current. But the demands on the motor by the robot are significantly more complex:
Often the motor is supposed to be in a certain position, move with a certain speed, not exceed a maximum force, reach a certain position without overshooting, etc.. To complicate this even more there are different kinds of electrical motors, that require different input signals. So, to serve the role of translating the desired state into electrical signals to the motor, the role of motor controllers exists. These can be specialized pieces of hardware or software running on general purpose hardware with the necessary IO capabilities. In order to answer the question how this translation works we need to turn to systems theory from electrical engineering.

Whenever you have a stateful system there are two ways of controlling it. In Open Loop Control only the should-be signal influences the output value. In Closed Loop or Feedback Control the output is composed of the should-be value and a feedback signal that measures the actual state of the system. In many real-world applications Closed Loop Control is used for its ability to quickly reach a new target value, while keeping oscillation and overshoot to a minimum.

For these real-world applications it has been implemented in many different ways, from mechanical structures like a centrifugal governor to specialized integrated circuits.

In this paper we will implement closed loop control of a brushless DC motor on a Raspberry Pi 4. This effort is part of the larger goal of building a successor to CARL [Schütz 20] at RRLAB. There are a variety of different ways the motor controllers could be implemented, the old CARL for example used FPGAs. We chose to use a Raspberry Pi, because it provides us with some advantages and exciting opportunities.

## 1.2 Advantages of Closed Loop Control on General Purpose Processors

When compared to the FPGA approach used in CARL Single Board Computers (from now on abbreviated as SBCs) such as a Raspberry Pi are cheaper and easier to program. This comes at a cost, however, typically FPGAs are faster at high speed IO operations, which is an important port of a Closed Loop Controller. Fortunately the Raspberry Pi 4 has special purpose hardware for SPI and PWM IO which should help us alleviate this disadvantage, while outperforming the FPGAs in general compute tasks due to the higher clock frequencies.

When compared to ready built PID ICs a Raspberry Pi is more expensive, consumes more power and must be programmed vs only settings a few variables. But it comes with one major upside for a research project such as ours: It is not bound to a traditional PID controller. If the need arises to use a different control scheme or compute additional tasks in parallel, the Raspberry Pi can!

In addition, using a Raspberry Pi provides an opportunity to use the Rust programming language and validate some of its claims.

## 1.3 A new programming language for embedded systems?

Systems programming and embedded systems are two closely linked categories. In both of them the usage of C++ and especially C have dominated that of any other language. In the last 5-10 years however several new languages have come up, that claim to be able to replace C and C++ for various parts of the systems programming field.

- Go by Google has seen wide adoption in high-performance network related applications, due to its focus on simplicity while maintaining reasonably high performance.[Go 09]
- Zig targets a direct replacement of C for low level systems such as writing operating system kernels or embedded applications. In order to easily integrate into existing C projects the zig compiler can compile both C and Zig files.[Zig 23]
- Carbon directly targets replacing C++ by being able to include C++ files and compiling to the same ABI.[Carbon 22]
- Rust, which was originally developed by Graydon Hoare at Mozilla.[Klabnik 18]

The controller implementation in this thesis will be written in Rust, as that gives us the opportunity to validate some of Rusts claims on its suitability for embedded programs and interoperability with existing SDKs.

To cite the exact quotes, we will be looking at in detail:
"Interoperability - Integrate Rust into your existing C codebase or leverage an existing SDK to write a Rust application."
"Portability - Write a library or driver once and use it with a variety of systems, ranging

from very small microcontrollers to powerful SBCs."
[Embedded Rust 15]

In order to do that, we will implement the controller twice. Once running bare-metal while using the Circle library for interacting with the hardware, and second while running on a standard linux kernel on the Raspberry Pi. This allows us to compare the effort of writing an embedded application in Rust to the effort when writing the same application on top of a general purpose operating system. The Circle library is a C/C++ SDK for all Raspberry Pis which will serve as our test bed for exploring Rust's C interoperability.

Since we will be taking a detailed look at some of Rusts features let us provide some general information on the language.

Rust was designed to be a similarly performant alternative to C and C++ while eliminating some of their most common pitfalls:

- buffer overflows
- out of bounds read
- race conditions
- reading uninitialized memory
- dereferencing null pointers

It does this through a combination of approaches where the most important ones are RAII through an ownership system and a lifetimes for references instead of pointers. We will cover these in more detail in the Background chapter.

# 2. Background

There are several prerequisites necessary for the rest of the thesis. In this chapter we are first going to cover the architectural background of bare-metal, operating systems and scheduling, then, continue with the language prerequisites for Rust and last, provide an overview of the hardware used.

## 2.1 Bare-Metal vs Operating System

Many embedded systems run bare metal, that is, without an operating system kernel that provides task scheduling, heap allocations, and hardware access. The reasons for this are that microcontrollers are often limited in their computational abilities and the operating system always causes some overhead. In addition to the computational overhead, many operating system scheduling strategies are optimized for throughput as opposed to reaction times. With the event-driven nature of many embedded systems, these higher and often irregular reaction times are not desirable. As a result of this, there is a specialized class of operating systems optimized for embedded systems. These so-called real-time operating systems (RTOS) [Marwedel 21] minimize the space and computation overhead and employ special scheduling strategies to guarantee worst-case reaction times.

While bare-metal systems have several advantages, as already mentioned, they are not without their disadvantages. Writing a bare metal program requires the programmer to be closely familiar with several additional aspects of the project:

- the hardware
  - the processor and processor architecture the project is built for
  - hardware interfaces and their protocols, such as SPI and PWM for our project
  - understanding hardware interrupts may be required to write a performant implementation
- the software

  - many embedded platforms require a specific toolchain, that may come with its own caveats
  - debugging is more complex and often requires the use of hardware interfaces such as JTAG
  - hardware specific libraries

All of these take additional time during a project and require more skilled programmers, which can add its own cost.

### 2.1.1 Real time scheduling

Earlier we mentioned real-time operating systems, but what does real-time actually mean? Real-time systems are systems that guarantee the execution of a certain task before a certain time limit [Marwedel 21, p. 208]. These can be divided into two groups, soft real-time systems and hard real-time systems. Hard real-time systems consider any violation of a time limit an unrecoverable failure. Soft real-time systems, on the other hand, just do their best to avoid exceeding time limits, but do not consider them catastrophic. Hard real-time is necessary everywhere where failure would result in damaging equipment or harming humans, such as in industrial machinery and emergency systems. Soft real-time, on the other hand, occurs where exceeding the time limits would degrade the result of the process. Examples of this would be decoding audio and video, where missing the time frame would lead to distortions or visual artifacts, or rendering a video game, where an additional frame of render time may lead to higher input latencies.

Hard real-time by its nature can only consider the worst-case runtime of processes, which leads to a problem with common hardware and software implementations. Many parts of how computers work are optimized to speed up the average case, compromising on worst-case performance. On the hardware side examples of this are caches and branch predictions, where caches do not change the worst-case performance and branch predictions even make them worse. On the software side, the main problem is multitasking, where a scheduler assigns tasks to computing resources. General purpose operating systems often come with schedulers optimized for throughput as opposed to latency.

This is where the already mentioned real-time operating systems come in. By applying different scheduling algorithms, often coupled with additional metadata such as priorities and time limits, RTOS attempt to find a schedule that meets all time limits at runtime.

### 2.1.2 PREEMPT-RT

The Linux kernel itself is not capable of real-time. This has two reasons:
The available scheduling algorithms are not designed for this.
Many kernel functions are not preemptible, making it impossible to guarantee reaction times for high-priority tasks.

Because of Linux's broad compatibility with hardware and familiarity for developers several projects adapt Linux to be real-time capable.

- RTLinux, which is end-of-life since 2011

- RTAI (Real Time Application Interface), which hasn't seen an update for two years at the time of writing [RTAI 22]
- Xenomai [Xenomai 24]
- PREEMPT-RT [PREEMPT-RT 24]

Of these, PREEMPT-RT is the most interesting one for us as we are going to use it later on. PREEMPT-RT is a community-developed patch-set for the Linux kernel. It works by providing preemptible versions of all non-preemptible kernel functions and adding the necessary scheduling algorithms to the Linux kernel. The goal here is to upstream everything at some point into the mainline Linux kernel. This up-streaming effort has seen some notable successes; The SCHED_DEADLINE scheduler is up-streamed since kernel 3.14 and by version 6.0 most of the functions were up-streamed as well.

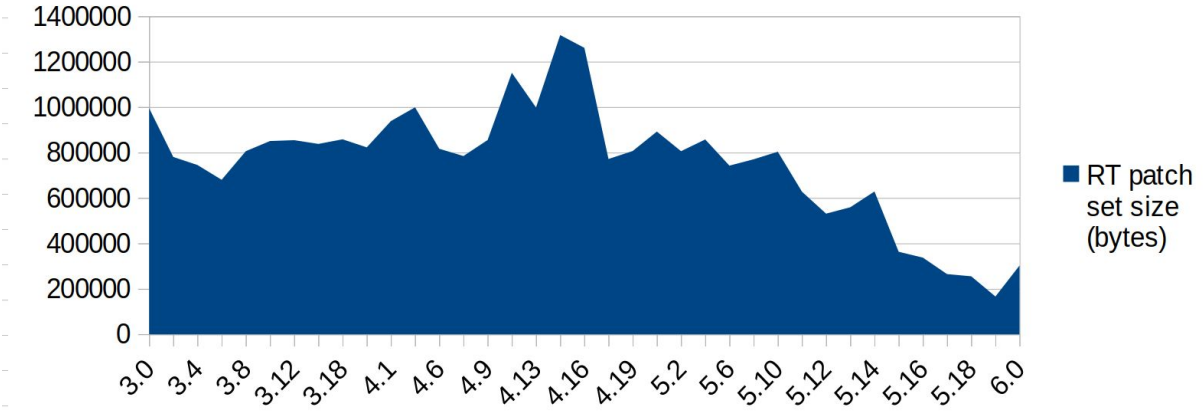



**Figure 2.1:** Size off the PREEMPT-RT patchset over time. A decrease in size means code got up-streamed [Opdenacker 23]

With the current stable version of 6.6-rt still not everything is up-streamed and patching the kernel sources is still required.

## 2.2 Rust Prerequisites

Rust is a modern systems programming language that claims to achieve comparable performance to languages with manual managed memory such as C and C++, while also being "memory safe", a term we will define more clearly shortly. Many modern languages use a garbage collector to manage memory on the heap, which is an additional thread that periodically checks all memory allocations to see if they are still used and frees them if they are not. This introduces an overhead, as the main execution path may be interrupted by the garbage collector and the analysis of the existing allocation takes execution time. Manually memory-managed languages, on the other hand, suffer from the possibility of memory errors, such as double frees and use after frees. Typically manually memory-managed languages also allow for arbitrary memory access leading to data races, reading uninitialized memory, etc. Rust aims to be the best of both worlds, managing memory without a garbage collector, while also avoiding memory management and access errors.

### 2.2.1 Terminology

There are a few terms used in Rust that need only a short introduction.
First, a crate is the equivalent of a package in Rust, it can be either a library or an executable.
Second, traits are the equivalent of interfaces; we will use both terms interchangeably in this thesis.
Third, cargo is the build manager for Rust, comparable to NPM, CMake, Meson, Maven.

### 2.2.2 Memory management by ownership

To achieve this, Rust employs a combination of methods, the most important one being ownership. Ownership in this context means that every resource has exactly one variable assigned as its owner. For our purposes resource usually means "memory on the heap", but can mean anything that needs setup and cleanup when it is used, such as file descriptors, sockets, or an SPI connection. The process of binding resource initialization and destruction to the lifetime of a variable is neither new nor an invention of Rust, but rather the RAII (Resource acquisition is initialization) paradigm from C++. Since in the real world resources need to be used from multiple points, Rust introduces three ways to safely allow that:

- transferring ownership, called moving in Rust
- lending exclusive access, using a mutable reference
- sharing non-exclusive access, using multiple immutable references

Of these references, there can always only be either one mutable reference or an arbitrary number of immutable references per resource. At compile time, a part of the compiler called the borrow checker enforces this by checking the lifetimes of all references. Reference lifetimes are their own complex topic and not necessary for the rest of this thesis, so we will not go into them.

```
1 // create a variable a that owns a heap allocation
2 let a = Box::new([0; 1000]);
3 // move the ownership from a to b
4 let b = a;
5 // this statement would now not compile, because a no longer owns the
     allocation
6 let c = a[0];
```

### 2.2.3 Unsafe Rust

Earlier we said that we would define "memory safety" more closely, and now is the time for that. Operations in Rust are called memory unsafe when they cause undefined behavior (UB), where undefined behavior is defined as the following by the [Rustonomicon 24]:

- Dereferencing dangling or unaligned pointers

- Breaking the pointer aliasing rules of LLVM's noalias memory model
- Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI
- Causing a data race
- Executing code compiled with target features that the current thread of execution does not support
- Producing invalid values:
    1. a bool that isn't 0 or 1
    2. an enum with an invalid discriminant
    3. a null fn pointer
    4. a char outside the ranges [0x0, 0xD7FF] and [0xE000, 0x10FFFF]
    5. a ! (! is a type used to mark unreachable or infallible values)
    6. an integer (i*/u*), floating point value (f*), or raw pointer read from uninitialized memory, or uninitialized memory in a str
    7. a reference/Box that is dangling, unaligned, or points to an invalid value
    8. a wide reference, Box, or raw pointer that has invalid metadata:
        - dyn Trait metadata is invalid if it is not a pointer to a vtable for Trait that matches the actual dynamic trait the pointer or reference points to
        - slice metadata is invalid if the length is not a valid usize
    9. a type with custom invalid values that is one of those values, such as a NonNull that is null.

This comes with a problem though, in order to prevent UB from occurring, some operations such as reading from an arbitrary memory address are not possible because the compiler cannot check if they would cause UB. But in reality there are cases where it is both necessary and safe to read from specific memory addresses, most notably when talking to memory-mapped IO.

This is where the unsafe keyword in Rust comes in. Unsafe marks sections of code where the programmer has additional tools at his disposal but is also responsible for upholding the aforementioned guarantees. Unsafe code that successfully upholds all the invariants is called sound, and unsafe code that can produce UB is called unsound. The additional things unsafe allows us to do are:

- Dereference raw pointers
- Call unsafe functions (including C functions, compiler intrinsics, and the raw allocator)
- Implement unsafe traits
- Mutate statics

- Access fields of unions

Most of these are not interesting for us, so we will focus only on dereferencing raw pointers and calling unsafe functions. Dereferencing raw pointers is self-explanatory in its meaning; the interesting part here is when is it sound to do so? There are two requirements for a raw pointer access to be sound:
The pointer must be correctly aligned and not aliased. If the access is a write we need to make sure that we don't cause data races through an additional read or write.

Calling other unsafe functions may not seem very important at a first glance, but this changes when we consider that there are a few sources of unsafe functions that are not written by us. The most important for us are C functions. Since C code is inherently memory unsafe, it is up to the programmer to check its soundness. So, calling any C code from Rust needs to be done in an unsafe block. The second important source of unsafe functions is the standard library. Here are compiler intrinsics, platform specific functions, and raw access to the heap allocator, which allows for a C style manual memory management.

### 2.2.4 Rust in the embedded world

Since we will look at Rust through the lenses of an embedded program, we should discuss the ecosystem around embedded programming in Rust. Compared to C and C++ there is a notable amount of standardization in the Rust ecosystem. The Rust Embedded SIG (Special Interest Group) provides several key libraries that provide common interfaces for all Rust embedded software to use. This allows for a clear distinction between hardware drivers that implement these interfaces and those who use a generic version of the interface. The most important of these libraries both in general and for our uses are:

- embedded-hal (interfaces for SPI, PWM, I2C, GPIO, and delays)
- embedded-io (interfaces for UART, USB)
- critical-section (interface for uninterrupted code execution)

There are also libraries for the CAN-bus, DMA controllers, and async versions of embedded-hal and embedded-io, but, since these are not relevant for our project and function in similar ways to embedded-hal, we will not discuss them further.

Implementations of these traits are called HALs (Hardware Abstraction Layers). Typical implementations for a microcontroller consist of three parts:

1. A Peripheral Access Crate (PAC) that contains all the memory addresses for the memory mapped IO. This can be generated using svd2rust from an svd file.
2. A HAL crate that implements the embedded-hal traits, embedded-io traits for UART, DMA, PCIe and a critical section implementation for the bcm2711 processor.
3. A Board Support Crate/Package (BSP) that contains board but not processor-specific implementations, such as the boot process, USB and Ethernet.

The svd2rust crate used for creating PACs is also an official crate maintained by the Embedded SIG.

## 2.3 Hardware Details

With all the software details out of the way, we can take a closer look at the hardware that we will be using. First, we will give an overview over the Raspberry Pi and afterwards a detailed explanation of the parts we're actually using. For communication with our position encoder the iC-MU, we use the SPI bus, and for sending the output signal to the motor driver, we use a PWM signal.

### 2.3.1 Raspberry Pi 4

The Raspberry Pi 4 is a Single Board Computer (SBC) based on the Broadcom bcm2711 processor [Raspberry Pi Ltd 22]. It has four 64bit arm cores and current models run at a frequency of 1.8GHz, while earlier production units ran at 1.5GHz. The interesting features for us are mainly its IO capabilities. The onboard GPIO pins support simple on/off input and output, as well as six SPI buses, four I2C buses, a UART port and two PWM outputs.

### 2.3.2 SPI

The SPI bus is a serial communication interface between one master and several slaves. The name itself stands for "Serial Peripheral Interface". It uses 4 lanes for communication[Meroth 21, p. 220]:

1. CLK or SCLK - the clock generated by the master
2. SS or CS or CE - of these lines there exists one per slave, it is used to select which slaves are currently enabled for communication. The names stand for "Slave Select", "Chip Select", "Chip Enable"
3. MOSI or PICO - the data line which the master writes and the slaves read. The names stand for "Master out Slave in" and "Peripheral in Controller out"
4. MISO or POCI - the data line which the slaves write and the master reads. The names stand for "Slave out Master in" and "Perihperal out Controller in"

Depending on the polarity and phase of SCLK a SPI bus can be operated in four different modes. The clock polarity is called CPOL and the clock phase is called CPHA. A CPOL of 0 means that SCLK idles at logical low, and a CPOL of 1 means that SCLK idles at logical high. CPHA influences when data is sent. When $CPHA = 0$ data is outputted when SCLK transitions to its idle level. For $CPHA = 1$ data is outputted when SCLK transitions from its idle level. The relation of CPOL and CPHA to the 4 modes can be seen in table 2.1.

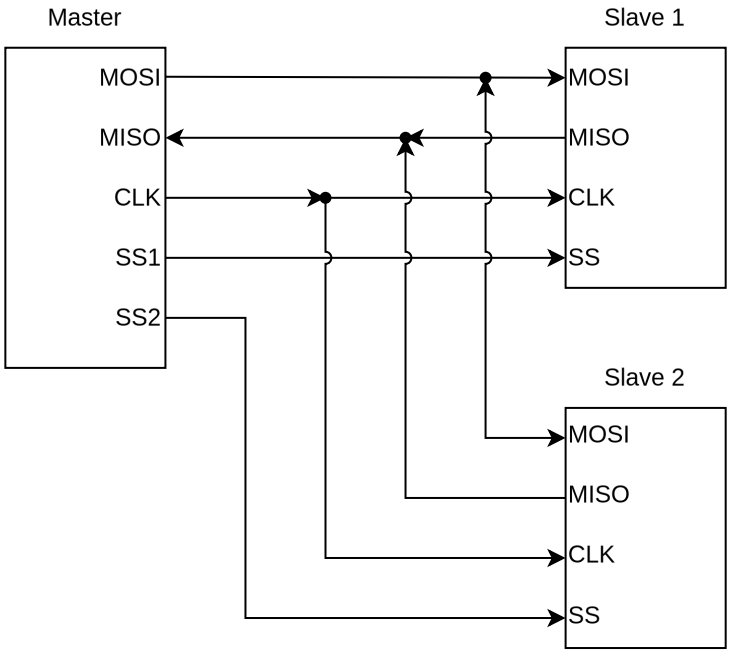



**Figure 2.2:** Wiring of SPI with one master and two slaves

| | CPOL = 0 | CPOL = 1 |
|---|---|---|
| CPHA = 0 | Mode 0 | Mode 2 |
| CPHA = 1 | Mode 1 | Mode 3 |

**Table 2.1:** SPI Modes

### 2.3.3 PWM

Since we are going to use PWM to control the motor driver, PWM should also receive an introduction. Pulse Width Modulation (PWM) [Marwedel 21, p. 187] is the technique of generating a digital signal where the ratio of on to off time is variable. This allows driving electric loads at only a percentage of their maximum power, for example, when dimming LEDs or running a motor at slower speeds. The signal can be fully described by its frequency or period, and duty cycle or on-time. The period is the time a full on-off cycle takes, the frequency simply the inverse. The duty cycle and on-time describe what happens inside of a cycle, with the duty cycle being the percentage of time that the signal is on and the on-time being the raw time value for which the signal is on.

### 2.3.4 iC-MU

The iC-MU position encoder is a chip that is mounted on a motor in combination with a magnetic band, to measure the current position of the motor, as well as the number of previous rotations. It is able to communicate this information through SPI, BiSS, SSI and ExtSSI [iC-Haus GmbH 21]. Since the only common protocol with our Raspberry Pi is SPI, we use that. The communication itself works by sending the chip an Opcode,

optionally with some arguments such as a register address. The chip replies by echoing the Opcode and any arguments and, if applicable, attaching the return values. The different opcodes are one byte long and can be seen in table 2.2

| Code | Description |
|---|---|
| 0xB0 | ACTIVATE |
| 0xA6 | SDAD-transmission (sensor data) |
| 0xF5 | SDAD Status |
| 0x97 | Read Register |
| 0xD2 | Write Register |
| 0xAD | Register Status |

**Table 2.2:** Opcodes for the iC-MU

The communication protocol the iC-MU expects is read from its configuration EEPROM and falls back to SPI. Since all the default configuration values of the iC-MU are fine for us, we pulled the EEPROM data pin to ground.

The only opcode we will use is the SDAD-transmission one, as it fetches the current position of the motor. The transmission process can be seen in figure 2.3

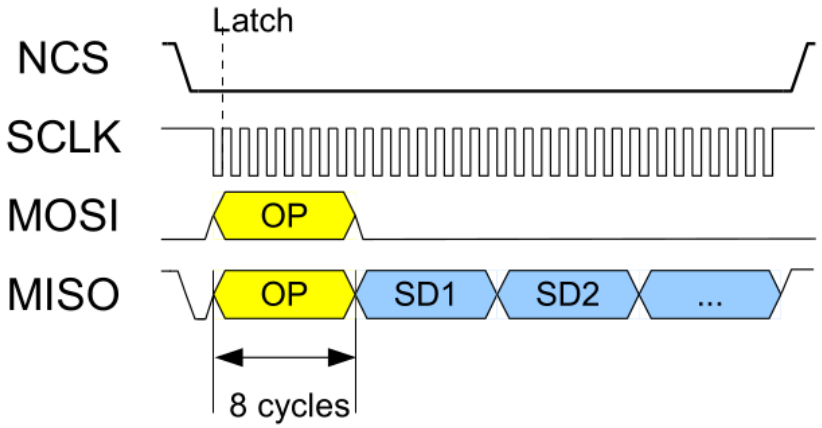



**Figure 2.3:** The SDAD-transmission process, from a signal and byte point of view [iC-Haus GmbH 21, p. 37]

# 3. Related Work

The idea of using SBCs for embedded systems tasks is not new. The advantages of short test cycles, flexible implementations and a for the programmers familiar environment make general purpose processors seem very enticing for embedded systems. This in combination with the spread of Raspberry Pis has led to similar research to our paper.

## 3.1 Rasbperry Pi Research

In *Raspberry Pi performance analysis in real-time applications with the RT-Preempt patch* [Carvalho 19] the timing behavior of a Raspberry Pi 3 with the PREEMPT-RT patch is investigated. This includes a test polling a GPIO input, a test for reaction times to hard- and software interrupts, and a test running a ping-pong loop between two Rasbperry Pis with a one millisecond delay.

Although these tests are not directly comparable to our tests, as we mainly output things, while they tested input latencies, there is still very valuable information for us in this paper. When we look at the test results for the timed loop test in Figure 3.1, we can see that they observed maximum latencies of three to eight times the minimum latencies. If this translates into the worst-case latencies for our testing later, it would render the Linux version unsuitable for our goal.

Almost identical research has been done on the Raspberry Pi 3 in *Real-Time Performance and Response Latency Measurements of Linux Kernels on Single-Board Computers* [Adam 21]. This paper also used the ping-pong loop between two Raspberry Pis for the reaction time measurement. There are some small differences to the former paper:

1. For comparison the paper includes the Beagle Bone Black SBC
2. The kernels tested are the far older 4.14 and 4.19 kernels
3. The paper also considers multiple Linux distributions (Ubuntu, Arch Linux, and Debian), which can affects the tools and flags to compile the kernel.

| | Latency ($\mu$s) | Load | |
|---|---|---|---|
| | | Minimum | Maximum |
| Without RT-preempt | Average | 98,13 | 94,09 |
| | Minimum | 34,83 | 10,83 |
| | Maximum | 243,06 | 628,13 |
| | Standard deviation | 17,28 | 27,86 |
| | Jitter | 208,23 | 617,29 |
| With RT-preempt Round-Robin | Average | 20,41 | 23,60 |
| | Minimum | 12,52 | 3,91 |
| | Maximum | 57,44 | 149,35 |
| | Standard deviation | 3,61 | 16,16 |
| | Jitter | 44,92 | 145,44 |
| With RT-preempt FIFO | Average | 19,33 | 23,47 |
| | Minimum | 10,14 | 1,02 |
| | Maximum | 59,53 | 171,05 |
| | Standard deviation | 3,35 | 16,24 |
| | Jitter | 49,38 | 170,03 |

**Figure 3.1:** Timed loop test results from [Carvalho 19]

Interesting here are the results in comparison with the former paper, for the Debian version, which had the newest kernel, the minimum response time was 40$\mu$s and the maximum 90 $\mu$s. One possible explanation for the much higher latencies than in the former paper could be the comparatively old kernel version.

Similarly, *A Performance Evaluation of Embedded Multi-core Mixed-criticality System Based on PREEMPT RT Linux* [Li 23], tests latency and throughput characteristics with PREEMPT-RT on the Raspberry Pi 4. Although this is the same hardware as we use, there are some important differences:
They propose an approach that divides real-time and best-effort tasks by running the best-effort tasks on a virtualized unmodified Linux kernel.
They measure task wake-up times using cyclitest, a tool developed together with PREEMPT-RT, while we test for our specific use case and compare to a bare-metal version.

During the writing of this thesis the new Raspberry Pi 5 was released, with a faster processor and more IO capabilities. Does this mean that our research is already outdated? Not necessarily, similar to the Raspberry Pi 4, the Raspberry Pi 5 suffers from supply bottlenecks, as demand outpaces the production capacity of the Raspberry Pi foundation, making Raspberry Pi 4s cheaper and more readily available. Additionally many improvements of the Raspberry Pi 5 are more benefitial for server style tasks as opposed to embedded systems:

- The additional IO capabilities of the Pi 5 are just more and faster PCIe lanes, which has very little relevance for embedded systems and especially our use case.
- The addition of a real-time clock to have persistent time across reboots is of little importance for our use case.

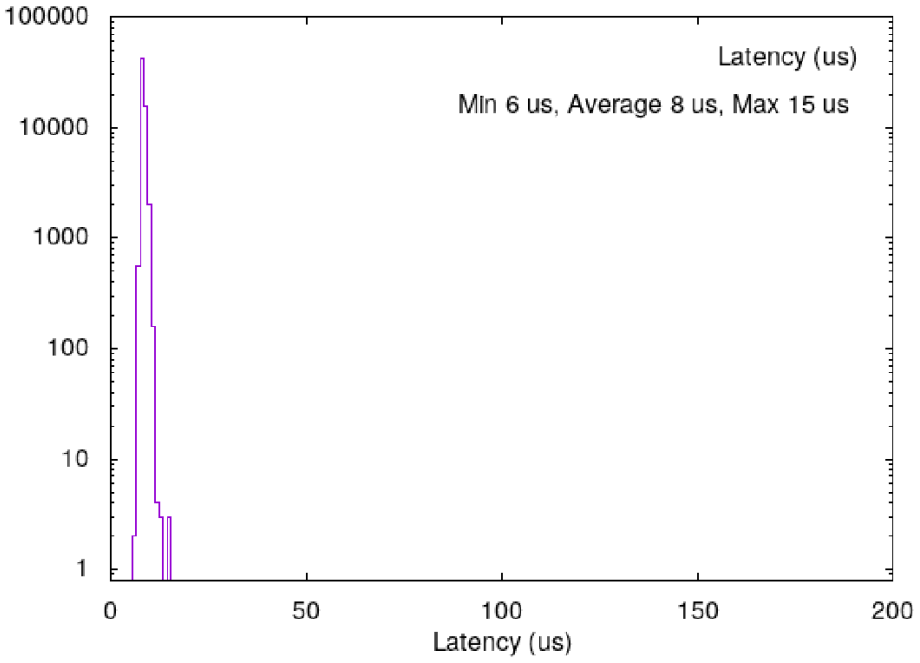



**Figure 3.2:** Kernel scheduling latency on the Raspberry Pi 4 measured with cyclitest [Li 23]

- The faster processor may only benefit our system marginally, as the IO times on the Pi 4 are already significantly slower than the computations.

Still, there will be some benefits to the Pi 5 and there is already some preliminary research on its real-time capabilities. *A Preliminary Assessment of the real-time capabilities of Real-Time Linux on Raspberry Pi 5* [Dewit 24] examines the real-time capabilities of the Raspberry Pi 5 using cyclitest, similar to *A Performance Evaluation of Embedded Multi-core Mixed-criticality System Based on PREEMPT RT Linux* [Li 23]. In Figure 3.2 and 3.3, we compare the results of the two papers. This comparison is interesting because according to these results, the Pi 4 is better at real-time operations at the moment. This may be because Linux is not yet fully optimized for the architecture of the new bcm2712 processor.

## 3.2 Rust for embedded systems

Rust's usage for embedded systems is a very active research area. The research can be roughly split into two categories:

1. Evaluating new approaches that are made possible by Rust's modern features, like the borrow checker and zero sized types.

2. Evaluating how Rust solves old problems with already known solutions.

In the first category, research into embedded operating systems is a very common target. A very prominent example of this is the Tock RTOS [Levy 17a], a Rust-written OS kernel

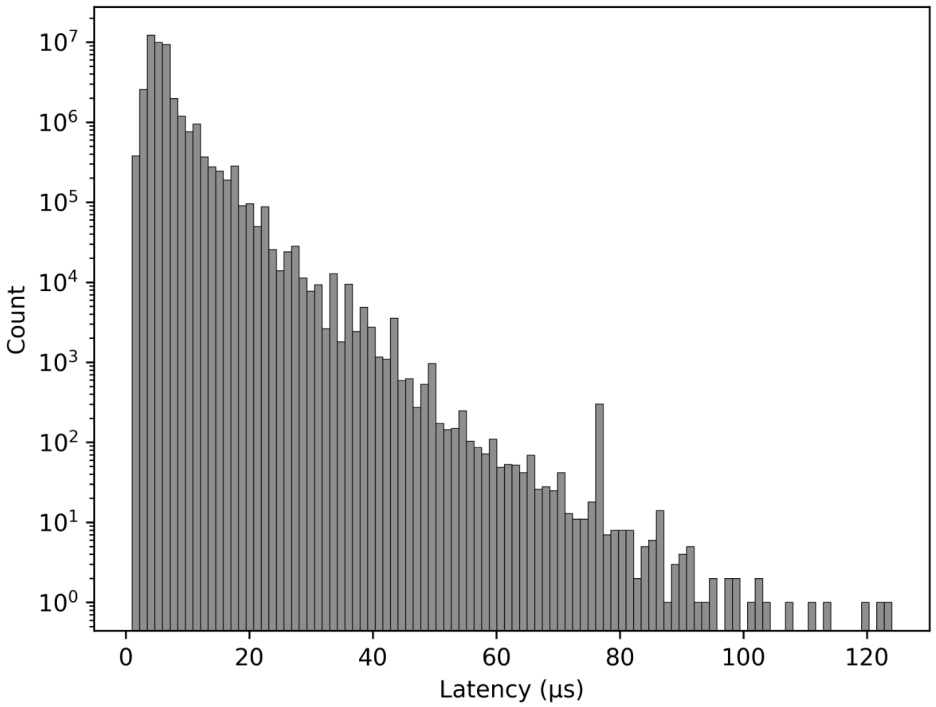



**Figure 3.3:** Kernel scheduling latency on the Raspberry Pi 5 measured with cyclitest [Dewit 24]

with a strong focus on security. For this RTOS alone, there are already papers investigating the general architecture [Culic 22], the limitations [Levy 15], possible solutions [Levy 17b], and the security model [Ayers 22].

But since our research clearly falls into the second category, we will focus on that. For an up-to-date overview of the general state of Rust, *Rust for Embedded Systems: Current State, Challenges and Open Problems* [Sharma 23] is a recent paper that focuses on a general overview. The important part for us here is their second research question: Interoperability of RUST. How well can RUST interoperate with existing C codebases and what are its challenges?

Because this is closely related to our own Rust research, this opens up a few questions. How did they investigate the question? What did they find, and how does it relate to our research?

For this research, they developed a simple Rust application on top of FreeRTOS, a common RTOS written in C, and the same application in C built on top of a Rust port of FreeRTOS. We will later only look at the direction Rust on top of C. The results of their research are two findings:

> Significant development effort is required to engineer a C embedded application on top of RUST RTOSes.

> It is easy to convert a self-contained, embedded system component of C RTOS into RUST.

There is one big difference in our research, however, that could lead to a different conclusion than they have; Our project will be built upon the Circle library, which is a C++ library, that provides a mostly C-style API, but has some functions that make use of inheritance.

# 4. Concept and Implementation

In this chapter, we will first consider the general control flow of our application, both for the bare-metal and OS based versions. For the Linux version specifically, we will then compare multiple approaches to optimize the OS both in terms of average performance and worst-case latencies. These different versions will be compared later to the bare metal version in the Experiments chapter. The bare-metal version demands a more detailed look, as it is the focus of this thesis. For the bare-metal version, we will first look at the boot process on the Raspberry Pi and how to produce a Rust binary that successfully boots. From that we will continue with how the bindings to Circle were created as well as how and why an abstraction on top of it was written to make the underlying C code adhere to Rust's safety guarantees.

## 4.1 The Control Loop

The general control flow follows the principle of a superloop. On startup we initialize the communication hardware and from there on it is just a fetch, compute, update loop. In the fetch step, we get the current position of the actuator, the control value, and the time since the last cycle. The position is fetched through SPI from the IC-MU and the duration since the last cycle from a CPU timer. For increased reproducibility, we will mock an external control value input by replacing it with a fixed signal based on timers on the Raspberry Pi. In the compute step, we apply those values to our PID-controller formula to get the new output value. Finally, in the update step, we update the duty cycle of the output PWM signal based on the just computed output value.

## 4.2 Linux based version

Interacting with the SPI and PWM hardware on Linux works through the spidev character device in /dev/spidev0.0 and the pwm interface in /sys/class/pwm. For Rust there exists a library called rppal that provides a simple interface for interacting with these. Because of this, it will be used for the Linux versions.

### 4.2.1 Approaches to minimize jitter

In order to give the Linux version the best chance to compete with the bare-metal version, we try two optimization approaches. First, we tell the Linux scheduler to reserve an entire core for our program, so that it is never moved between cores and never interrupted by core local interrupts. By this we aim to improve the worst-case iteration times without changing the average case. In an attempt to go even further, we will also try a Linux kernel with the PREEMPT_RT patchset, which replaces the scheduler by a real-time capable one and makes every section of the kernel preemptible. The idea behind this is that even the parts of our program that run in kernel space, by making syscalls to the spi and pwm kernel drivers, run at a higher priority than other kernel tasks. To verify if our approaches made any significant difference, we will also be running an unmodified stock kernel with our program at default priority.

### 4.2.2 Setup with rppal

Setting up Spi and PWM through rppal isn't very complex, but it comes with a few small caveats that are noteworthy and is a good primer on how linking to external code works in Rust which will be useful when we turn to the bare-metal version.

Rust projects are built using the Cargo build system. This allows us to add rppal into the compilation process by editing our projects Cargo.toml file.

```
1  [package]
2  name = "os-based"
3  version = "0.1.0"
4  edition = "2021"
5
6  [dependencies]
7  ctrlc = "3.4.4"
8  embedded-hal = "1.0.0"
9  pid-ctrl = "0.1.4"
10 rppal = { version = "0.18.0", features = ["hal"] }
11
12 [profile.release]
13 lto = true
```

Without going into too much detail, the Cargo.toml first specifies some metadata about our project, such as name, version, and edition. The edition is a language standard specifier similar to C99 vs. C11, and 2021 is the latest at the time of writing. Rust editions typically get released every three years and are backward compatible. Afterwards, the dependencies are specified with versions, and optionally feature flags and where to pull them from. By default, this pulls the rppal source code from crates.io and builds it together with our program and links the two afterwards. Alternatively, we could have specified a link or path to a custom version of rppal if that were needed. Then, setting up the devices is done by including the relevant types and creating instances of them.

```
1  use embedded_hal::spi::SpiBus;
2  use pid_ctrl::PidCtrl;
```

```
3  use rppal::pwm::{Channel, Polarity, Pwm};
4  use rppal::spi::{Bus, Mode, SlaveSelect, Spi};
5  pub fn setup() -> (Spi, Pwm, PidCtrl<f64>) {
6      (
7          Spi::new(Bus::Spi0, SlaveSelect::Ss0, 20_000_000, Mode::Mode0).
       unwrap(),
8          Pwm::with_frequency(Channel::Pwm0, 1_000., 0.5, Polarity::Normal,
        false).unwrap(),
9          PidCtrl::new_with_pid(10., 1., 5.),
10     )
11 }
```

On a stock Raspberry Pi 4 with Raspberry Pi OS this would still fail, because by default the spi and pwm device tree overlays are not enabled. In order to enable them, we need to set dtparam=spi=on in /boot/firmware/config.txt for spi and dtoverlay=pwm for pwm.

### 4.2.3 Getting the current position

In the SPI (2.3.2) and iC-MU (2.3.4) sections we have already seen how SPI and the position encoder that we use work. This leaves us with the question of how to transmit the actual position data. The position transmission is initiated by the SPI master by sending the SDAD_TRANSMISSION opcode. The IC-MU chip responds by echoing the opcode and appending the position. The format and length of the position are dependent on several configuration values on the IC-MU, but for our purposes the default configuration is appropriate. In the default configuration, this is 19 bits of actual position data with five zeros added as padding. In Rust, the transmission looks like this:

```
1  fn get_position(spi: &mut Spi) -> f64 {
2      const SDAD_TRANSMISSION: u8 = 0xa6;
3      let mut buf = [SDAD_TRANSMISSION, 0, 0, 0, 0];
4
5      spi.transfer_in_place(&mut buf).unwrap();
6      let position = u32::from_be_bytes(buf[1..].try_into().unwrap()) >>
       13;
7
8      position as f64
9  }
```

Here we transmit by writing in the same buffer that we are reading from for sending, which means that we need four bytes of space, one for the opcode and three for the data. To make converting to an integer easier, we add a fifth byte, so we can just interpret the last 32 bits of the buffer as our position. Because the IC-MU sends in big endian, we use u32::from_be_bytes() for the conversion.

### 4.2.4 Providing a PID controller

Although the aim of this thesis is to enable the flexible implementation of different control schemes, implementing these is not within our scope. For now, we will implement a simple PID controller that stays the same between the Linux and bare-metal versions.

For this we decided on the pid-ctrl library to be used, as it is easy to use, works on bare metal, and supports differing time deltas.

The process of including it is analogous to rppal. In fact in the Setup with rppal (4.2.2) chapter, we already included it in our build process and created an instance of the Pid type. So, all we need to do is to use it inside our main loop.

```
pub fn iteration<F: Fn() -> f64>(
    last_iteration_start: Instant,
    get_setpoint: F,
    spi: &mut Spi,
    pid: &mut PidCtrl<f64>,
    pwm: &Pwm,
) -> (Instant, Duration) {
    // fetch step: calculate elapsed time, get new position and setpoint
    let iteration_time = last_iteration_start.elapsed();
    let iteration_start = Instant::now();

    let position = get_position(spi);
    let setpoint = get_setpoint();

    // compute step: calculate new output value
    pid.setpoint = setpoint;
    let output = pid
        .step(pid_ctrl::PidIn::new(position, iteration_time.as_secs_f64()
   ))
        .out;

    // update step: output the new value over PWM
    pwm.set_duty_cycle(output).unwrap();

    (iteration_start, iteration_time)
}
```

## 4.3 Bare-Metal

Bare metal programming is traditionally something reserved for microcontrollers or OS kernels. Although the Raspberry Pi 4 is not a microcontroller but a fully fledged PC capable of running desktop-class operating systems, its simple boot process and access to low-level IO such as $I^2C$, SPI and GPIO pins capable of analog input and output still allow it to be used, as if it were a microcontroller.

### 4.3.1 HAL vs Circle

As we have seen in the Rust in the embedded world (2.2.4) section the traditional way to program for a microcontroller in Rust is to create a library for it that implements all the traits from the embedded-hal crate and a second library for the boot process. Doing that

from scratch is a significant task that would exceed the scope of a bachelor thesis by quite a bit. However, in order to enable a better evaluation of Rust's suitability for embedded systems, we will demonstrate how such an implementation would look for the Gpio pins in Rust native HAL (4.4). For the actually functional bare-metal program, we will be using a different route. The Circle library is a C++ framework that provides bare-metal access to pretty much all of the Rpi's available hardware. That means not only Gpio, Spi, and I2c, but also the more complex subsystems, such as PCIe, USB, display outputs, and ethernet. So we are going to use this library through Rust's Foreign Functions Interface (FFI from now on).

### 4.3.2 Cross compiling for bare-metal aarch64

On Linux, there are two main ways to obtain a Rust compiler. Through your distributions package manager or through rustup, a tool for managing, if necessary, multiple versions of rustc, as well as any connected tooling. For cross compilers, the first option is nearly non-existent, so we need to use rustup. Rustup is easily installed on Linux by running the following command.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

To be able to cross compile to aarch64-unknown-none we run

```
rustup target add aarch64-unknown-none
```

For most targets the rustup target add command does two things. First, it downloads the compiler backend for the new target architecture. Second, it downloads a precompiled version of the rust standard library for the new target if available. Since we are adding a bare-metal target, there is no complete standard library, but Rust's standard library is split into three parts, of which we can use two. The lowest and most constraint part is the core library. It contains primitive types and operations on them. Because this part is built purely on llvm primitives it is available for every target and does not need to be compiled as it lives inside the compiler. The second part available to bare metal systems is the alloc library, which adds types, operations, and traits that depend on the existence of a memory allocator. If we want to use this part, we need to provide a memory allocator. Fortunately for us, Circle provides a malloc and free implementation, which we can use as our allocator. The third part of the standard library is called std and contains all types that rely on syscalls to the underlying operationg system. This means networking, file system access, and timing.

If we were only cross-compiling rust code, this would already be enough. But since we also want to compile Circle and then link the final binaries together, we need a C cross-compiler as well as a linker. Arm provides precompiled toolchains for C, so we will use the latest one provided by them at the time of writing.

There is one final thing to do, when linking to external binaries Rust uses ${target}-gcc as the linker, so in our case aarch64-unknown-none-gcc. To use aarch64-none-elf-ld instead which we got from Arm in our project directory we add a .cargo/config.toml file with the following content:

```
[target.aarch64-unknown-none]
linker = "aarch64-none-elf-ld"

[build]
target = "aarch64-unknown-none"
```

The linker line sets the linker to the correct binary and the target lines changes the default target when using cargo.

### 4.3.3 Bootup process of the Raspberry Pi 4

The first part of creating a bare metal controller is creating a binary that even boots on the Raspberry Pi.

1. The on-chip first stage bootloader is loaded from address 0x60000000
2. It checks if a recovery.bin is present on the first FAT32 partition of the SD-card. If it is found, it is flashed to the onboard EEPROM chip and the Rpi is rebooted.
3. Else it loads the second stage bootloader and its configuration from the EEPROM
4. The second stage bootloader loads the third stage bootloader either from the SD-card, a USB storage device or the network, depending on the configuration. The third stage bootloader is split into the binary start4.elf, the relocation table fixup4.dat, and config.txt for the configuration.
5. The third stage bootloader loads the kernel image and jumps to address 0x80000. By default, this is kernel8.img, but the kernel file name can be set in config.txt.
6. In some cases, this kernel.img may be another bootloader such as u-boot, which could then load grub. This additional boot part is used in many Linux distributions that support UEFI boot on aarch64 and not the Raspberry Pi specific chain.

To produce a bootable rust binary we start from a clean project folder generated with cargo new. This clean project contains a Cargo.toml for the build system configuration and a src/main.rs with the actual code.

First, we will add a .cargo/config.toml similar to what we saw in Cross compiling for bare-metal aarch64 (4.3.2).

```
[target.aarch64-unknown-none]
linker = "aarch64-none-elf-ld"
rustflags = ["-C", "link-arg=Tlink.ld"]

[build]
target = "aarch64-unknown-none"
```

The added rustflags line allows us to control which logical addressess code is placed at in the linking step. To control this, we create link.ld:

```
SECTIONS
{
    . = 0x80000;
    .text :
    {
        *(.text._start)
        *(.text*)
    }
}
```

The .text section of a binary is where the actual instructions lie as opposed to .data and .bss. What we are doing here is keeping all the segments in .text, placing .text._start at the beginning of .text and letting .text begin at 0x80000. As we already discussed, 0x80000 is the address to which the Raspberry Pi jumps after loading the kernel. Now for the actual Rust code. Since there is no standard library on bare metal, we need to tell Rust not to link to the standard library. This is done by prepending #![no_std] to main.rs. The second thing to prepend is #![no_main] because otherwise Rust expects a main function that can be called with the runtime setup required for command-line arguments. On bare-metal we neither have command-line arguments nor a runtime. One thing that the standard library provided for us was a panic_handler implementation. As the name suggests, the panic handler controls panic exceptions. On an operating system that typically includes logging the error message to stderr, but for bare-metal Rust expects us to provide an implementation. That leaves one final task, writing the function that will actually be run and ensuring that it is linked to the .text._start section.

```
#![no_std]
#![no_main]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_: &PanicInfo) -> ! {
    loop {}
}

#[no_mangle]
#[link_section = ".text._start"]
pub extern "C" fn _start() -> ! {
        loop {}
}
```

The #[no_mangle] attribute before the start function tells the Rust compiler not to mangle the function name, which it would do by default. The #[link_section] attribute ensures that our start function is linked to the section we forced in the linker script to be included at address 0x80000. The pub attribute makes the function visible to the linker instead of only to the compiler. The extern C attribute tells the compiler to adhere to the C ABI

(Application Binary Interface) for the start function. This is necessary because the Rust ABI is unstable and may be more complicated than jumping to the start address.

At this point, we can obtain a binary by running cargo build. But this binary would be an elf executable that cannot be executed without a program loader. ELF in this context stands for executable-linkable-format and is the file format that Linux expects for executable binaries, with a header that contains all the relevant information for the dynamic linker. However, the bootloader just copies the kernel image to RAM and sets the instruction pointer to 0x80000. This means that we have to rip out the code from our elf binary and create a flat binary without an elf header. The program that does this is called objcopy and is a standard part of the gnu toolchain. This means that we already have the correct version aarch64-none-elf-objcopy available by installing the C cross-compiler.

```
cargo build --release
aarch64-none-elf-objcopy -O binary \
    target/aarch64-unknown-none/release/<project-name> \
    kernel8.img
```

When building a Rust-only bare-metal project, there is an easier way to obtain objcopy than to install the complete Gnu toolchain.

```
rustup component add llvm-tools
cargo install cargo-binutils
```

Which allows to build and extract the binary in one command with any target available that was added through rustup.

```
cargo objcopy --release -- -O binary kernel8.img
```

### 4.3.4 Compiling Circle and linking it to Rust

Many languages provide a way to link to C ABI functions; examples of this are C++ with its extern C, Haskell with its FFI module, and Python. The reasons for this are very simple; for one, the C ABI for each target platform is stable and allows thus to be used as a common language to bridge other languages. Second, C is still a widely used language with many useful libraries in any field of programming, which can then be used by languages with a C interface.

As Rust is a systems programming language, the prevalence of C libraries in areas interesting to Rust is even greater. In its most basic form, the Rust FFI system works through declaring functions as extern C, when calling Rust functions from C and declaring extern C functions in Rust without a definition, when calling C code from Rust.

```
1 use std::ffi::c_int;
2
3 extern "C" {
4     fn c_function(number1: c_int) -> c_int;
5 }
6
```

```rust
#[no_mangle]
pub extern "C" fn rust_function(number2: c_int) -> c_int {
    number2
}

fn calling_c_from_rust() {
    let _ = unsafe { c_function(42) };
}
```

```c
int rust_function(int number2);

int c_function(int number1) {
    return number1;
}

void calling_rust_from_c(void) {
    rust_function(42);
}
```

Writing these bindings manually is fine for a handful of functions, but for anything larger in scope, some automation would be useful. Fortunately, there are a few well-supported libraries which aim to automate this under different circumstances. Bindgen takes in C header files and generates a Rust file with the appropriate extern C declarations from it. It works both as an executable for manual use and as a library that can be integrated into the build process. Since intercompatibility with C code has been important for Rust from the beginning, bindgen is an official project maintained under the same umbrella as the compiler and cargo. One company that makes heavy use of Rust is Mozilla for their Firefox browser. Because firefox is mostly written in C, the opposite case of bindgen, calling Rust from C code is very common in the firefox codebase. As a result, Mozilla created cbindgen, which takes in Rust files and creates a C header with declarations for all public functions. For interacting with C++ code, there are two more libraries, cxx and autocxx. Because C++ functions do not necessarily adhere to the C ABI calling them from Rust and vice versa can be difficult with bindgen, as we will see later. Instead, cxx takes a declaration for a common interface between C++ and Rust and generates a bridge from pure C functions that both languages can access. Autocxx by Google is similar to bindgen for C++, but instead of directly generating a Rust interface, it generates a cxx interface, which in turn generates the Rust interface.

To come back to our original problem, which one should we use to access the Circle library from our Rust code? Since Circle is written in C++ and we just want to call C++ functions from Rust, not the other way around, autocxx would be the ideal choice. Unfortunately, at the time of writing autocxx does not work with bare-metal programs. There is ongoing work on this, and autocxx itself can actually be compiled for bare metal, but one of its dependencies cannot. This leaves us with two options: use cxx and manually write the interface, or use bindgen and work with a more convoluted interface. Since bindgen has been stable for a long time and cxx is still partially a work in progress, we chose to use bindgen for a more meaningful insight into Rust.

As mentioned earlier, bindgen can be used either as a command-line utility to generate the bindings once, or as a build dependency to automatically generate the bindings at compile time. We are going to use the second approach. To add bindgen as a build dependency we add

```
[build-dependencies]
bindgen = "0.69.4"
```

to our Cargo.toml.

Cargo provides the option of creating a build.rs file that will be compiled and executed before the actual compile. In order to create the bindings and link the final binaries correctly we create a build.rs.

```
1 use bindgen::{Builder, MacroTypeVariation};
2 use std::{env, path::PathBuf};
3
4 fn main() {
5     ...
6 }
```

Inside the main function, we do three different things. First, setting up the linker search path, second, linking to the correct binaries, and third, generating the bindings for Rust. For all this, we have placed the Circle source code in the circle directory in our project folder as well as configured and built the Circle library. We will take a more detailed look at configuring and compiling Circle soon, but for now the focus is on the build script.

The linker search path is expanded as follows:

```
1  use bindgen::{Builder, MacroTypeVariation};
2  use std::{env, path::PathBuf};
3
4  fn main() {
5      let manifest_dir = PathBuf::from(env::var("CARGO_MANIFEST_DIR").
       unwrap());
6      println!(
7          "cargo:rustc-link-search={}",
8          manifest_dir.join("circle").display()
9      ); // for circle.ld
10     println!(
11         "cargo:rustc-link-search={}",
12         manifest_dir.join("circle/lib/usb").display()
13     ); // for libusb.a
14     println!(
15         "cargo:rustc-link-search={}",
16         manifest_dir.join("circle/lib/input").display()
17     ); // for libinput.a
18     println!(
19         "cargo:rustc-link-search={}",
```

```
        manifest_dir.join("circle/lib/fs/fat").display()
    ); // for libfatfs.a
    println!(
        "cargo:rustc-link-search={}",
        manifest_dir.join("circle/lib/fs").display()
    ); // for libfs.a
    println!(
        "cargo:rustc-link-search={}",
        manifest_dir.join("circle/lib").display()
    ); // for libcircle.a
```

Then we link to the binaries mentioned in the above code:

```
    println!("cargo:rustc-link-lib=usb");
    println!("cargo:rustc-link-lib=input");
    println!("cargo:rustc-link-lib=fatfs");
    println!("cargo:rustc-link-lib=fs");
    println!("cargo:rustc-link-lib=circle");
```

Generating the bindings is simple thanks to bindgen, we generate bindings for all symbols in wrapper.hpp, a file that we will write where we include all the headers interesting to us. Notable are the use_core() and vtable_generation(true) methods. Use_core() tells bindgen to use only types from the core library instead of std in the bindings, which is necessary for us as we are targeting a bare metal environment. Vtable_generation(true) enables generating types for the vtables of C++ classes. This allows us to make calls to inherited functions of classes.

```
    let bindings = Builder::default()
        .header("wrapper.hpp")
        .use_core()
        .clang_arg(format!(
            "-I{}",
            manifest_dir.join("circle/include").display()
        ))
        .vtable_generation(true)
        .default_macro_constant_type(MacroTypeVariation::Signed)
        .parse_callbacks(Box::new(bindgen::CargoCallbacks::new()))
        .generate()
        .expect("Unable to generate bindings");

    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
    bindings
        .write_to_file(out_path.join("bindings.rs"))
        .expect("Couldn't write bindings!");
}
```

The build script places the generated bindings in a file called bindings.rs in the cargo build directory. To include this file in our main Rust code, we create a new source file that includes the bindings:

```
#![allow(non_upper_case_globals)]
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]
#![allow(improper_ctypes)]
#![allow(unused)]

include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
```

Because the bindings will use the C++ type and function names, we would receive a lot of warnings for not following Rust naming conventions. To ignore the warnings, we add some allow directives to the file.

Compiling and configuring Circle works like this: Obtain the source code for Circle either from a tarball or the repository, we used the Step46 tag, as it was the latest at the time of writing. Inside of the Circle source directory execute the following commands:

```
./configure --raspberrypi 4 --prefix aarch64-none-elf-
./makeall
```

To create a bootable binary from this, we need to create a suitable main.rs and .cargo/config.toml. Both are very similar to what we already saw in the Bootup process of the Raspberry Pi 4 (4.3.3) chapter, so we will show only the differences here. In the .cargo/config.toml we only change the compile flags to use Circle's linker script and .init section as the boot function. Circle's linker script is not as simplistic as ours, because it provides the necessary sections for static variables, constants, and a heap. The .init section is defined in the main Circle library and provides a bootup function that initializes the .bss and .data sections correctly by zeroing them and copying any constants to RAM.

```
...
rustflags = ["-C", "link-args=--section-start=.init=0x80000 -Tcircle.ld"]
...
```

In the main.rs instead of the _start function we create a function called main with the type void → int, because Circle links to such a function in order to call it after the bootup initialization is done.

```
...
#[no_mangle]
pub unsafe extern "C" fn main() -> c_int {
    ffi::reboot()
}
...
```

Building the final binary works the same as in Bootup process of the Raspberry Pi 4 (4.3.3). This leaves only one final step to make the Raspberry Pi boot our Rust binary. As mentioned in the Rpi boot process we need the start4.elf, fixup4.dat, bcm2711-rpi-4-b.dtb and config.txt files on the final SD card or USB stick. These can be obtained from the official Raspberry Pi repository, but Circle can download them for us. To download them through Circle, call make in the circle/boot directory.

### 4.3.5 The final implementation

With all the pieces in place, we can write the actual application. Similarly to the Linux version, we start by setting up Spi and PWM and afterwards start the infinite control loop. To setup Spi and PWM, we use the C++ constructors, which are translated by bindgen into the ClassName::new() functions. In addition, we need to set pin 18 in alternate function mode 5, so that it is wired up to the PWM signal from the PWM controller. The alternate function mode we need can be found in the bcm2711 data sheet or the Circle documentation.

```
1    const SPI_FREQ: c_uint = 20_000_000;
2    const CPOL: c_uint = 0;
3    const CPHA: c_uint = 0;
4    const SPI_DEVICE: c_uint = 0;
5    let mut spi = ffi::CSPIMaster::new(SPI_FREQ, CPOL, CPHA, SPI_DEVICE);
6    spi.Initialize();
7
8    let _pwm_pin = ffi::CGPIOPin::new1(
9        18,
10       ffi::TGPIOMode_GPIOModeAlternateFunction5,
11       core::ptr::null_mut(),
12   );
13   const PWM_RANGE: u32 = 1024;
14   let mut pwm = ffi::CPWMOutput::new(
15       ffi::TGPIOClockSource_GPIOClockSourceOscillator,
16       2,
17       PWM_RANGE,
18       true,
19   );
```

The actual control loop is almost identical to the Linux version, with the only differences being the way to get the time and the different syntax of setting the PWM duty cycle. To measure the iteration-time deltas, we use Circle's CTimer class. The accuracy of this clock will be discussed in more detail in the Methodology (5.1) section. Setting the PWM duty cycle works a bit differently in Circle than in rppal, in rppal we express the target value through a float in the range of 0 to 1, which aligns nicely with the output of the pid library. Circle instead expects us to set a granularity at device initialization and then an integer value of 0 to granularity. We used a granularity of 1024 steps and clamp it because Circle will just segfault on a too high value, which could not happen in pure Rust.

```
1    pwm.Start();
2
3    let mut iteration_start = ffi::CTimer::GetClockTicks64();
4    loop {
5        let position = get_position(&mut spi);
6        let setpoint = get_setpoint();
7
8        pid.setpoint = setpoint;
9        let time = ffi::CTimer::GetClockTicks64() - iteration_start;
```

```
        let output = pid
            .step(pid_ctrl::PidIn::new(
                position,
                f64::from(time as u32) / 1_000_000.,
            ))
            .out;
        iteration_start = ffi::CTimer::GetClockTicks64();

        pwm.Write(
            PWM_CHANNEL1 as c_uint,
            ((output * PWM_RANGE as f64).clamp(0., PWM_RANGE as f64)) as c_uint,
        );
    }
```

As Circle also does not provide an inplace SPI send-receive function, we also need to change the get_position() function a bit to use two buffers instead of one.

```
unsafe fn get_position(spi: &mut ffi::CSPIMaster) -> f64 {
    const SDAD_TRANSMISSION: u8 = 0xa6;
    let buf_write = [SDAD_TRANSMISSION, 0, 0, 0, 0];
    let mut buf_read = [0; 5];

    const CS: u32 = 0;

    spi.WriteRead(
        CS,
        buf_write.as_ptr() as *const c_void,
        buf_read.as_mut_ptr() as *mut c_void,
        5,
    );

    let position = u32::from_be_bytes(buf_read[1..].try_into().unwrap());

    position as f64
}
```

## 4.4 Rust native HAL

Instead of using a foreign language library like Circle for this project, we could also have written our own Rust native HAL. This would provide several key benefits:

- Vastly simpler build process, completely eliminating the need for the C toolchain
- Upholding Rust's memory safety guarantees, which we largely lost/ignored when using Circle

- Possibly higher performance because we could use link time optimization on the entire program
- Access to any Rust crate that depends on an embedded-hal implementation.

In order to understand why we chose against this approach, let us look at the complexity it already takes to implement a HAL just for the GPIO pins of the Raspberry Pi. A pure native Rust stack for the Raspberry Pi 4 would consist out of the following components:

1. A Peripheral Access Crate (PAC) that contains all the memory addresses for the memory mapped IO. This can be generated using svd2rust from an svd file.
2. A HAL crate that implements the embedded-hal traits, embedded-io traits for UART, DMA, PCIe and a critical section implementation for the bcm2711 processor.
3. A Board Support Crate/Package (BSP) that contains board but not processor-specific implementations, such as the boot process, usb and ethernet.

Fortunately Broadcom provides the svd files for their processors, and someone already has generated a PAC from them and published it on crates.io. From the HAL we implemented the GPIO pins as well as single core critical sections, and from the BSP we implemented an entry macro to mark the boot function and a linker script that supports the .data and .bss sections.

For the HAL gpio implementation we need to take the GPIO struct from the PAC that contains all addresses of the IO registers and use it to create a new pin type that implements the embedded-hal traits. GPIO pins on the Raspberry Pi can be in input, output, and alternate function mode. These modes can be switched through the fsel (function select) registers. In C, one would have to check at run-time if a pin is in the correct mode before using it. By modeling the modes as type parameters in Rust, we can check at compile time that functions only use pins in the correct mode, with obvious benefits for performance and binary size.

```
1  use core::marker::PhantomData;
2
3  pub trait GpioExt {
4      type Parts;
5
6      fn split(self) -> Self::Parts;
7  }
8
9  pub struct Unknown;
10
11 pub struct Input<MODE> {
12     _mode: PhantomData<MODE>,
13 }
14
15 pub struct Floating;
16 pub struct PullDown;
```

```
17 pub struct PullUp;
18
19 pub struct Output;
20
21 use bcm2711_lpa::GPIO;
22 use embedded_hal as hal;
23 use core::convert::Infallible;
24
25 pub struct Parts {
26     pub pin0: Pin0<Unknown>,
27 }
28
29 impl GpioExt for GPIO {
30     type Parts = Parts;
31
32     fn split(self) -> Parts {
33         Parts {
34             pin0: Pin0 { _mode: PhantomData },
35         }
36     }
37 }
38
39 pub struct Pin0<MODE> {
40     _mode: PhantomData<MODE>,
41 }
```

The Pin0 type can be now differentiated into Pin0<Output>, Pin0<Unknown>, Pin0<Input<Floating», Pin0<Input<PullDown» and Pin0<Input<PullUp». For these types, we can now implement constructors to switch between the types:

```
1  impl<MODE> Pin0<MODE> {
2      pub fn into_input(self) -> Pin0<Input<Floating>> {
3          unsafe { (*GPIO::PTR).gpio_pup_pdn_cntrl_reg0().set_bits(|w| {w.
       gpio_pup_pdn_cntrl0().none()})}
4
5          unsafe { (*GPIO::PTR).gpfsel0().set_bits(|w| {w.fsel0().input()})
       };
6
7          Pin0 { _mode: PhantomData }
8      }
9
10     pub fn into_input_pulldown(self) -> Pin0<Input<PullDown>> {
11         unsafe { (*GPIO::PTR).gpio_pup_pdn_cntrl_reg0().set_bits(|w| {w.
       gpio_pup_pdn_cntrl0().down()})}
12
13         unsafe { (*GPIO::PTR).gpfsel0().set_bits(|w| {w.fsel0().input()})
       };
14
```

```
15          Pin0 { _mode: PhantomData }
16      }
17
18      pub fn into_input_pullup(self) -> Pin0<Input<PullUp>> {
19          unsafe { (*GPIO::PTR).gpio_pup_pdn_cntrl_reg0().set_bits(|w| {w.
    gpio_pup_pdn_cntrl0().up()})}
20
21          unsafe { (*GPIO::PTR).gpfsel0().set_bits(|w| {w.fsel0().input()})
     };
22
23          Pin0 { _mode: PhantomData }
24      }
25
26      pub fn into_output(self) -> Pin0<Output> {
27          unsafe { (*GPIO::PTR).gpfsel0().set_bits(|w| {w.fsel0().output()
    })}
28
29          P0i { _mode: PhantomData }
30      }
31
32      pub fn into_output_low(self) -> Pin0<Output> {
33          unsafe { (*GPIO::PTR).gpfsel0().set_bits(|w| {w.fsel0().output()
    })}
34
35          P0i { _mode: PhantomData }
36      }
37
38      pub fn into_output_high(self) -> Pin0<Output> {
39          unsafe { (*GPIO::PTR).gpfsel0().set_bits(|w| {w.fsel0().output()
    })}
40
41          P0i { _mode: PhantomData }
42      }
43  }
```

Note the actual register write operations: the raw address, the bit offset as well as the written value have all been transformed into constants and methods when generating the PAC. Because these are writes to arbitrary memory addresses, they are unsafe by Rust's memory model. But, in contrast to the Circle implementation, we are not using unsafe here because we don't care about memory safety, but rather for its intended purpose to tell the compiler that we checked we are upholding all the safety guarantees. In this case, the register writes are safe for two reasons. For one, we know that there will never be unrelated data at this address, as it is hardwired to the MMIO. Second, set_bits() is atomic so we don't get race conditions from multiple pins using the same fsel register switching their modes. This leaves implementing the embedded-hal traits for the pins. For a normal gpio pin, these are InputPin with is_high() and is_low() as well as OutputPin with set_high() and set_low().

```
impl<MODE> hal::digital::InputPin for Pin0<Input<MODE>> {
    fn is_high(&mut self) -> Result<bool, Self::Error>{
        Ok(unsafe{ (*GPIO::PTR).gplev0().read().lev0().bit_is_set() })
    }

    fn is_low(&mut self) -> Result<bool, Self::Error> {
        Ok(unsafe{ (*GPIO::PTR).gplev0().read().lev0().bit_is_clear() })
    }
}

impl<MODE> hal::digital::ErrorType for Pin0<MODE> {
    type Error = Infallible;
}

impl hal::digital::OutputPin for Pin0<Output> {
    fn set_low(&mut self) -> Result<(), Self::Error> {
        Ok(unsafe{ (*GPIO::PTR).gpset0().write_with_zero(|w| {w.set0().
   set_bit()}) })
    }

    fn set_high(&mut self) -> Result<(), Self::Error> {
        Ok(unsafe{ (*GPIO::PTR).gpclr0().write_with_zero(|w| {w.clr0().
   clear_bit_by_one()}) })
    }
}
```

In order to actually use this code, the control flow would look like this:

```
let peripherals = hal::pac::Peripherals::take().unwrap();
let gpio = peripherals.GPIO.split()
let pin0 = gpio.pin0;
let output_pin = pin0.into_output();
pin0.set_high();
```

The Peripherals::take() method is generated by the PAC and is used to ensure that only one execution path can gain control over the peripherals initially. However, for the internal check if peripherals have already been taken, to work, svd2rust requires an implementation of the CriticalSection trait. CriticalSection, as the name already implies, is supposed to provide an environment in which execution cannot be interrupted. Implementing CriticalSection consists of the two methods acquire() and release(). Since the bcm2711 is a multicore CPU implementing these would usually mean writing a spinlock, but for simpilicity, we will force the user to only use a single core, which means that we only have to disable all interrupts, which is possible with a aarch64 specific assembly instruction.

```
use aarch64_cpu::registers::Writeable;
use aarch64_cpu::registers::DAIF;
use critical_section::RawRestoreState;
```

```
4
5  struct SingleCoreCS;
6  critical_section::set_impl!(SingleCoreCS);
7
8  unsafe impl critical_section::Impl for SingleCoreCS {
9      unsafe fn acquire() -> RawRestoreState {
10         DAIF.write(DAIF::A::Unmasked + DAIF::I::Unmasked + DAIF::F::
       Unmasked);
11     }
12
13     unsafe fn release(_: RawRestoreState) {
14         DAIF.write(DAIF::A::Masked + DAIF::I::Masked + DAIF::F::Masked);
15     }
16 }
```

By default, the bootup function from address 0x80000 is executed on all four cores, so if we want to allow only single core execution, we must halt the execution flow on the other cores.

To do this, we run a check on the MPIDR_EL1 register, which is an aarch64 specific register that contains the id of the current virtual core. Only if it is zero and we are on the first core, we continue with the user-provided ___start_rust function. If we are on another core, we halt execution by waiting for an interrupt.

```
1  use core::arch::asm;
2
3  use aarch64_cpu::{
4      asm::wfe,
5      registers::{Readable, MPIDR_EL1, MPIDR_EL1::Aff0},
6  };
7
8  #[no_mangle]
9  pub unsafe extern "C" fn __start() -> ! {
10     if MPIDR_EL1.read(Aff0) == 0 {
11         // safe because MPIDR_EL1.read() already checks if we're on
       aarch64
12         asm!("bl␣__start_rust");
13     }
14
15     loop {
16         wfe()
17     }
18 }
```

In total, this Rust native approach would allow an application to be written with minimal effort, including the BSP as a dependency in cargo would take care of all the linking intricacies, and the actual code would be almost the same as in the Linux based version. Because Spi and PWM are abstracted behind the embedded-hal traits, one could even use

a shared code base with a Linux version for rapid development and an embedded version for the final version. But implementing this fully would require a further and extremely detailed understanding of the inner working of the bcm2711, how to operate the DMAs to communicate with the SPI controller, the SPI controller itself, and the PWM controller.

## 4.5 Retrospect

Before we continue with the performance experiments in the next chapter, we should take a moment to look back at the pros and cons of using Rust that we found. The main question we wanted to answer is whether Rusts intercompatibility with C works for embedded systems and allows Rust to replace C in environments where only C libraries exist. While the short answer is yes, it works, we also need to mention the drawbacks.

For this, let us consider three scenarios:

- a Rust HAL already exists and we build the program on top in Rust
- a C library exists and we build the program on top in Rust
- a C library exists and we build the program on top in C

Programming the first version would be analogous to the Linux version that we presented earlier. The main difference would be that instead of rppal a HAL for the respective microcontroller would be used. As a result, in this ideal scenario, the advantages and disadvantages would be very similar to what we have seen with the Linux version. This includes on the advantages side the relatively low development effort and required know-how, Rust's memory safety guarantees, and ecosystem of easy to include dependencies.

The second version is what we did for the bare metal version earlier. The biggest disadvantage here is the development effort; as we saw earlier, this approach requires both additional time and knowledge to create the bindings between the languages, adapt the build systems to compiling and linking multiple languages, and translate complex types like strings safely between the languages. In addition to this disadvantage, we lose the advantage of Rust's memory safety because the underlying C code is unsafe, and in order to check the unsafe invariants, one would have to check the entire C codebase. While it is technically feasible to check the C code, at the point someone has a good enough understanding of both the hardware and the C code to thoroughly check it, the effort level is very similar to just writing a proper HAL and using a whole Rust stack. Another interesting insight we get from this comparison is where the added effort actually comes from. If Rust's approaches to memory safety would have made the language itself more complex, then the Linux based version, which as we established is comparable to a native Rust bare-metal version, would likely be closer to the bare-metal version in terms of complexity. Instead, most of the additional time was spent bridging the two languages and setting up the tooling around them.

The third version is the antithesis to what we have presented in this thesis so far and is only here for comparison. Writing everything in C brings us to a sort of middle point between the first two versions, where we get none of the advantages of Rust but also not the disadvantage of either writing a custom HAL or adapting a C library in Rust.

This leaves the question, when does it make sense to use our approach of adapting C to Rust? There are two scenarios where this is feasible:

1. There is some utility library in Rust that doesn't exist in C and is highly desireable for the current project.

2. The project should transition to Rust step by step instead of everything at once.

Due to the widespread usage of C, the first reason is applicable only in niche cases. The second reason, however, may apply for a lot more teams that want to evaluate Rust on their own or work in safety critical applications.

# 5. Experiments

In order to compare our different approaches, we measured the time it takes for an iteration of our control loop. This allows us to compare not only the speed of each version of the program but also the consistency in the iteration times.

## 5.1 Methodology

In order to make the benchmarks as comparable as possible, we set a few invariants between the runs:

- Each of the benchmarks is run on the exact same RaspberryPi 4B.
- The Pi runs at the stock core frequency of 1800MHz.
- The SPI Bus runs at 20MHz, which is the maximum the iC-MU supports.
- The linux versions run RaspberryPi OS, a derivative of Debian Linux with the stock configuration.
- The kernel version is 6.1.73 for the stock and isolated core versions.
- The realtime version runs on 6.1.73-rt which was compiled prior on the RaspberryPi.
- Our control loop target value is set to a constant for these tests and the IC-MU position encoder is fixed in place in order to return almost the same value each iteration.
- Rustc 1.79.0 was used for all rust builds, both native and cross compiled to bare-metal
- GCC 13.2.1 was used as the cross compiler as well as linker for compiling the Circle library and linking it to the Rust code.
- GCC 12.2.0 was used as the native linker for the Rust code.

### 5.1.1 Testing the linux based versions

The linux versions all run a common iteration function:

```
pub fn iteration(...) -> Instant {
    // fetch step: calculate elapsed time, get new position and setpoint
    let iteration_time = last_iteration_start.elapsed();
    let iteration_start = Instant::now();

    let position = get_position(spi);
    let setpoint = get_setpoint();

    // compute step: calculate new output value
    pid.setpoint = setpoint;
    let output = pid
        .step(pid_ctrl::PidIn::new(position, iteration_time.as_secs_f64()))
        .out;

    // update step: output the new value over PWM
    pwm.set_duty_cycle(output).unwrap();

    iteration_start
}
```

Time measurement is done here via Rusts stdlib Instant and Duration types. On Linux these compile to the clock_gettime syscall in order to get microsecond accurate time. Because the RaspberryPi 4 does not yet have a real-time clock like the Raspberry Pi 5, Linux uses the clock interrupts of a 1MHz oscillator to measure the time. The benchmarks are first run for 10000 iterations without measuring, in order to avoid any latency spikes because of cache misses in the real benchmark. The real benchmark is then run for 1000000 iterations while saving the measured time deltas to an array. To execute the benchmarks, the criterion framework for Rust was used as it allows easy configuration of the amount of samples, runs an unmeasured three-second warmup loop of the benchmark beforehand, and automatically generates plots and statistics data from the results.

For the isolated core version, we use the cset python program to easily manipulate the Linux kernels cpuset subsystem. This allows us with few commands to isolate one core from all running processes and run our program on it, without being disturbed by core local interrupts. Global interrupts such as spinlocks when running a syscall can, however, still halt our execution flow.

We can run cset from inside of our program with the current process id this way:

```
Command::new("sudo")
    .args([
        "cset",
        "shield",
        "--cpu=3",
```

```
6           "--kthread=on",
7           &format!("--pid={}", std::process::id()),
8       ])
9       .spawn()
10      .expect("Could not start cset binary")
11      .wait()
12      .expect("cset did not exit successfully");
```

For the real-time version we needed a kernel that can boot with the Raspberry Pi bootloader, have the out-of-tree kernel modules for spi and pwm and support fully preemtive scheduling. In order to obtain that combination, the official sources for the Rpi kernel were patched with the fitting version of the PREEMP_RT patchset and compiled.

To achieve this, Step 1 is to get all the required build dependencies.

```
1   sudo apt update && sudo apt install build-essential flex
       bison libssl-dev bc
```

Step 2 is to get the kernel sources and patch them with the correct PREEMPT-RT patchset.

```
1   wget https://github.com/raspberrypi/linux/archive/refs/tags
       /stable_20240124.tar.gz
2   wget https://cdn.kernel.org/pub/linux/kernel/projects/rt
       /6.1/older/patch-6.1.73-rt22.patch.xz
3   tar -xf stable_20240124.tar.gz
4   cd linux-stable_20240124
5   xzcat ../patch-6.1.73-rt22.patch.xz | patch -p1
```

Step 3 is to generate the kernel config, we use the provided default config for the Raspberry Pi 4 and only set the PREEMPT_RT config value to enable full kernel preemption.

```
1   make bcm2711_defconfig
2   ./scripts/config -e PREEMPT_RT
3   make olddefconfig
```

Step 4 is to actually compile the kernel; this takes about 3 hours on all 4 cores of the Raspberry Pi 4.

```
1   make -j4 Image.gz modules dtbs
```

Step 5 is to actually install the files.

```
1   sudo make modules_install
2   sudo cp arch/arm64/boot/dts/broadcom/*.dtb /boot/firmware/
3   sudo cp arch/arm64/boot/dts/overlays/*.dtb* /boot/firmware/
       overlays/
4   sudo cp arch/arm64/boot/dts/overlays/README /boot/firmware/
       overlays/
5   sudo cp arch/arm64/boot/Image.gz /boot/firmware/kernel8.img
```

In the end, reboot to load the new kernel.

### 5.1.2 Testing the bare-metal version

Benchmarking the bare metal version is a bit more involved than the linux-based versions. For measuring times, Circle's CTimer::GetClockTicks64() is used. This in turn returns the number of ticks of the 1MHz oscillator on the Rpi. Analogously to the Linux versions, the loop is run for 10000 iterations before we start measuring.

The second difficulty with the bare metal version is getting the results out of the RaspberryPi. We need to either transmit the data over a connection such as ethernet, spi, i2c to another PC or save it to a filesystem on removable storage. Since we are running bare-metal that means that we need a driver for one of these options. Because we are already using the Circle library for Spi and it provides simple access to a USB mass storage device with a FAT32 file system, we will be using it to save the results.

The process for this is equivalent to how we already used the SPI and PWM drivers in the Concept and Implementation (4) chapter, so we will only skim over the most important parts.

In our wrapper.hpp we need to include "circle/fs/fat/fatfs.h" and "circle/usb/usbhcidevice.h"

```
#include "circle/fs/fat/fatfs.h"
#include "circle/usb/usbhcidevice.h"
```

In our main.rs we initialize these devices and save the measured times as a csv file.

```
let mut usb_hci =
    ffi::CXHCIDevice::new(&mut interrupt_system, &mut timer,
       false, 0, null_mut());
let mut filesystem = ffi::CFATFileSystem::new();


((*usb_hci._base._base.vtable_).CUSBController_Initialize)(&
   mut usb_hci._base._base, true);

for _ in 0..10000 {
    ...
}
const N: usize = 200;
let mut times = [0; N];
for time in times.iter_mut() {
    ...
    *time = ffi::CTimer::GetClockTicks64() - iteration_start;
    ...
    iteration_start = ffi::CTimer::GetClockTicks64();
    ...
}

let partition = device_name_service.GetDevice(c"umsd1-1".
   as_ptr(), true);
```

```
22 filesystem.Mount(partition);
23
24 let file = filesystem.FileCreate(c"times.csv".as_ptr());
25 let mut buffer = String::from("iteration,elapsed_time_us\n");
26 for (n, time) in times.iter().enumerate() {
27     buffer.push_str(&format!("{},{}\n", n, time));
28 }
29 let buffer = alloc::ffi::CString::new(buffer).unwrap();
30
31 filesystem.FileWrite(
32     file,
33     buffer.as_ptr() as *const c_void,
34     buffer.count_bytes() as u32,
35 );
36 filesystem.FileClose(file);
37 filesystem.UnMount();
```

## 5.2 Results

Now for the actual measurements. In our simplest case, we look at some statistical data from the measurements. Most important for our purposes are the average and worst-case times.

| | Mean | Min | Max | Std Dev |
|---|---|---|---|---|
| bare-metal | 4.4us | 4us | 7us | 0.49us |
| linux-default | 40.38us | 39us | 876us | 11.53us |
| linux-isolated | 40.11us | 38us | 2200us | 12.29us |
| linux-rt | 53.35us | 51us | 2507us | 25.59us |

To better visualize this and better understand the behavior of any outliers, we have depicted the iteration times both in a histogram and in the raw data points. The iteration-time graphs allow for a better understanding of the relative difference between a normal iteration and the outlier ones, but due to the amount of datapoints, conceal how often or rare these outliers actually occur. In order to overcome this downside, we have the histogram with a logarithmic scale.
The histograms can be seen in Figure 5.1 for the bare metal version, Figure 5.3 for the unmodified Linux version, Figure 5.5 for the isolated core version and Figure 5.7 for the PREEMPT-RT version.
The iteration time graphs can be seen in Figure 5.2 for the bare metal version, Figure 5.4 for the unmodified Linux version, Figure 5.4 for the isolated core version, and Figure 5.8 for the PREEMPT-RT version.

The bare-metal version is the first one we will look at in more detail. The first observation from the graphs 5.1 and 5.2 is that the iteration time values are integer values. The reason for this is the earlier discussed 1MHz oscillator on the Raspberry Pi, which only allows us to count full microseconds. With the bare-metal version we are close to the limit of making useful time measurements on the Pi. The second observation is the consistency

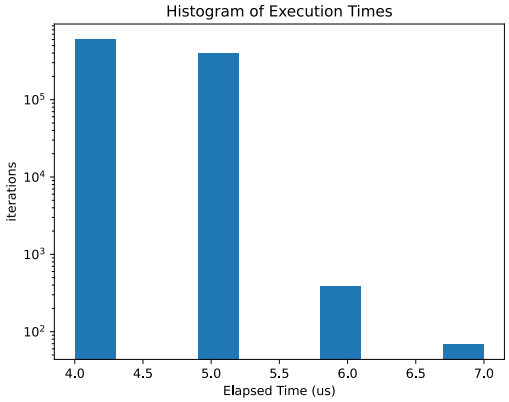



**Figure 5.1:** Histogram of the iteration times for one million iterations of the bare-metal version

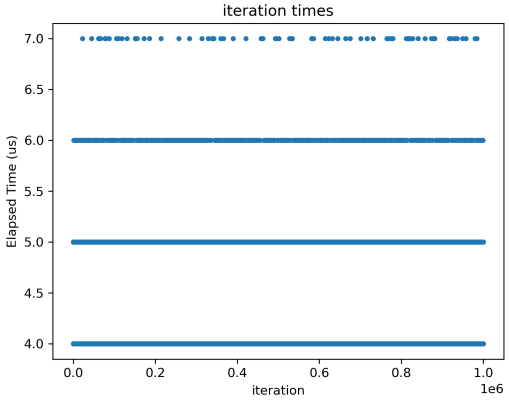



**Figure 5.2:** The iteration time of every iteration in chronologic order for the bare-metal version

of this version. As can be seen in the histogram, in 99% of the cases the iteration time is four or five microseconds, 99.9% is under seven microseconds and the rest is exactly 7 microseconds. This is also reflected in the standard deviation of the bare-metal version, which is the lowest of all even when adjusted for the mean.

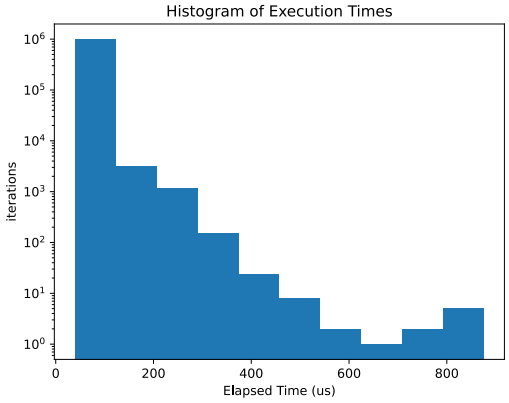



**Figure 5.3:** Histogram of the iteration times for one million iterations of the linux version without any modifications

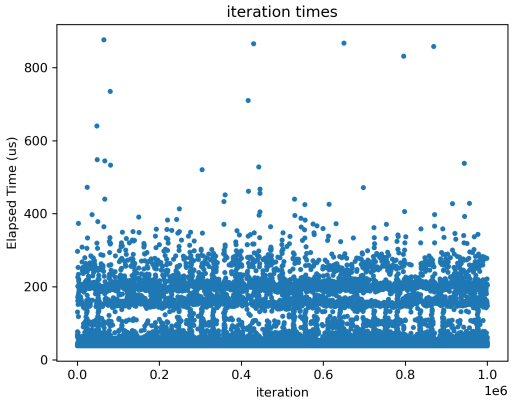



**Figure 5.4:** The iteration time of every iteration in chronologic order for the linux version without any modifications

With the first linux version we also see drastic differences compared to the bare-metal version. The quantization problem from the bare-metal version is gone, which can be nicely seen in Figure 5.4, but this is due to the generally higher runtimes. The average iteration times are about 40.38us, which is about ten times higher than in the bare-metal version, with its 4.4us. Similar to the bare-metal version, iteration the average iteration times again make up about 99% or the measured times. The worst-case latencies got disproportionately

bigger, in the bare-metal version they were roughly double the average iteration time with seven vs four microseconds, here they are over 20 times higher with 876 vs 40 microseconds. Both higher average and worst-case times we're expected when comparing a bare-metal and linux version, the disproportionately higher increase in worst-case latencies however is undesireable.

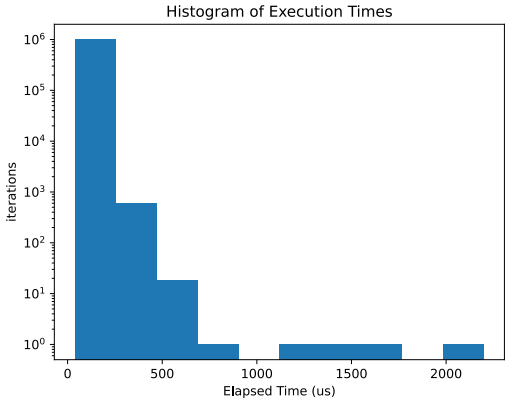



**Figure 5.5:** Histogram of the iteration times for one million iterations of the linux version on a reserved core

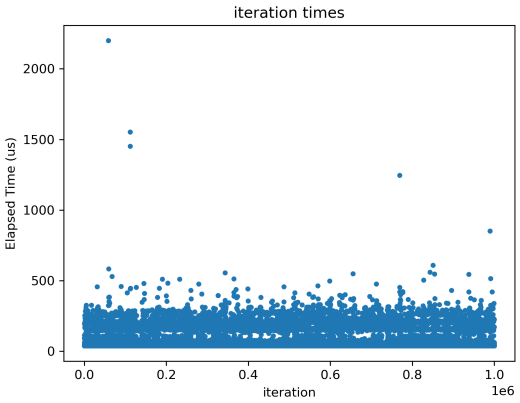



**Figure 5.6:** The iteration time of every iteration in chronologic order for the linux version on a reserved core

The goal of the isolated core and realtime linux versions was to reduce the expected impact linux had on the worst-case times. In light of that goal, the most important metric are the worst-case latencies. When looking at the raw numbers, the isolated core with 2.2ms and the realtime version with 2.5ms are significantly worse than the standard linux version with its 876us. There is however an important detail that gets lost in looking only at the worst value. When comparing the iteration time graphs in figure 5.4, figure 5.6 and figure 5.8, we can see that these extreme latencies only occur a handful of times, five times for the isolated core version and two times for the realtime version. This causes both graphs to be more compressed, hiding the fact that the isolated-core version slightly outperforms the standard linux version. For the realtime version this does the exact opposite, hiding that even the sub 1ms spikes occur roughly 100 times more often than in both other linux versions.

## 5.3 Interpretation

There are two different things to analyze here. First, the difference between the bare-metal and linux versions. All of the linux-based versions have average execution times of an order of magnitude more than the bare-metal version, which leaves the question: Is this speed difference adequately explained by the Linux version having to switch the execution context to the kernel and back for the PWM and SPI kernel drivers? To answer this properly, we profiled a run of the default Linux version, the result can be seen in Figure 5.9. From this we can see that most of the time (84.12%) is spent setting the duty cycle, which in turn means that the context switching overhead is not the issue. If it were, we

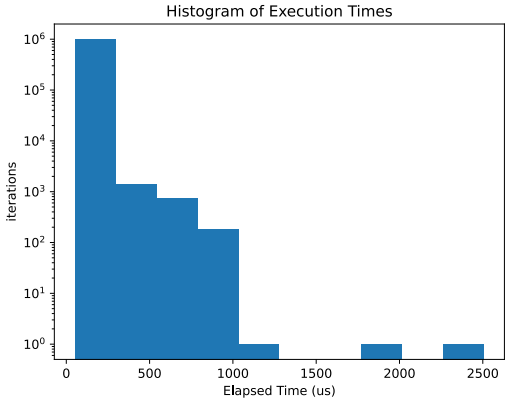



**Figure 5.7:** Histogram of the iteration times for one million iterations of the linux version with a realtime kernel

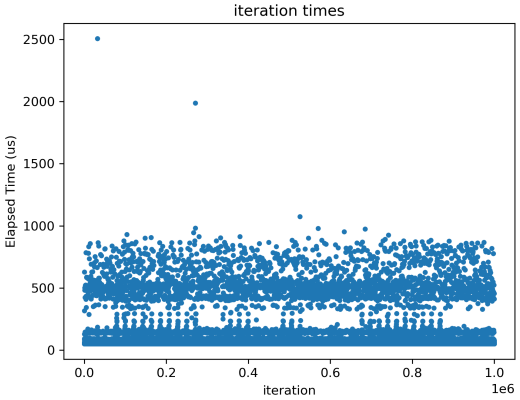



**Figure 5.8:** The iteration time of every iteration in chronologic order for the linux version with a realtime kernel

would see close to the same duration for SPI and obtain the system time, as we see for PWM. This can have one of two causes:
The Linux kernel PWM driver is very slow. Or, the rppal implementation of set_duty_cycle() is not optimal, either because it does extra steps not needed for our case, or talks to the kernel in an inefficient way. Of these two the second one is more likely for two reasons, code in the Linux kernel gets significantly more exposure and testing than rppal and the Circle PWM source code states that it is heavily inspired by the Linux implementation. If the reason is indeed that rppal could have been programmed differently for this task, a significant speedup could be achieved. Assuming the set_duty_cycle() was as fast as the SPI transfer, which would be reasonable since the spi transfer has to write and read to memory and the duty cycle function only has to do one write, the linux version times would come down to roughly 15us, putting it in a very comparable ballpark as the bare-metal version.

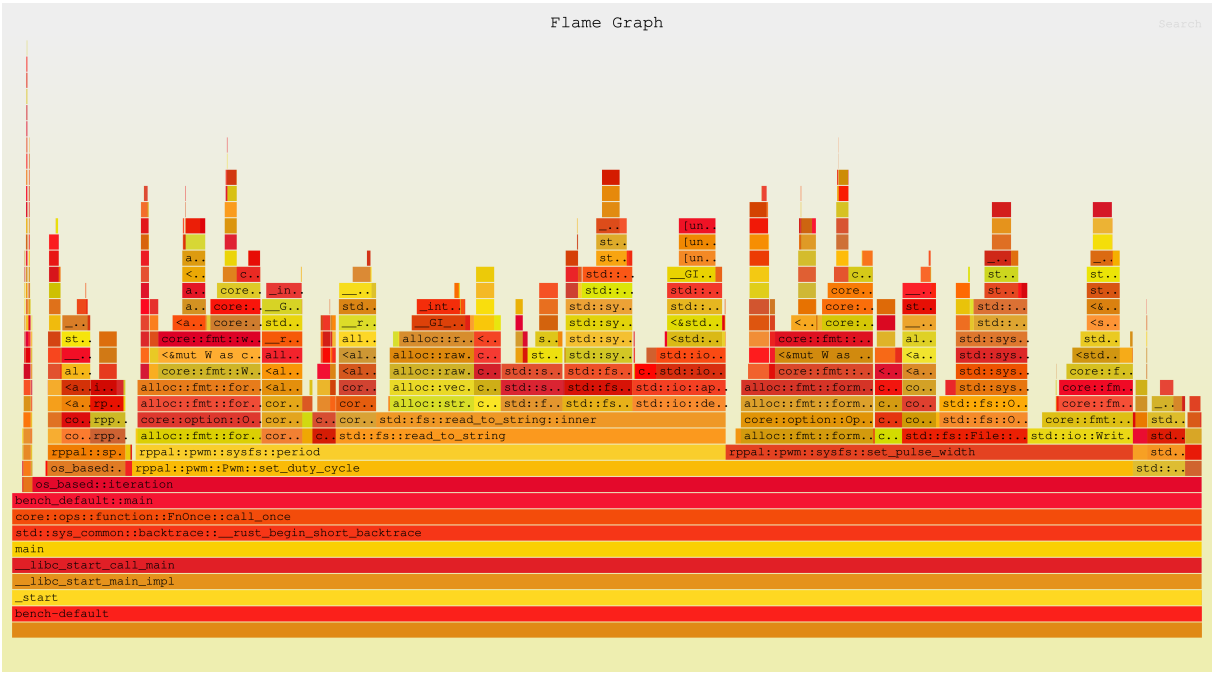



**Figure 5.9:** A flamegraph for the default linux version, with times spent inside of each function

The second part of the results worth analyzing is the comparison of the Linux versions with each other. The goal of reducing the worst-case latencies on Linux has failed. Every one of the Linux versions has regular lag spikes to more than an order of magnitude higher than the average iteration time. The isolated core and real-time versions that we hoped would reduce this behavior show very similar behavior as the stock version for iteration times under 1ms, but additionally have rare spikes up in the 2-2.5ms region. These spikes are a major problem, because when implementing a motor controller on top, the possible high reaction times result in several problems: From a safety perspective, when hitting an obstacle, such as possibly a human, we want the motor to slow down, but with these spikes the motor could be using its full force on an obstacle for up to 2.5ms. The second problem is accuracy, in the Introduction chapter we discussed that part of the job of a a controller is to prevent large overshoot and oscillation, both of which become more difficult with higher reaction times. This is especially important for the intended target application of our motor controller in the CARL [Schütz 20, ] successor, as balancing a bipedal robot is especially demanding on accuracy.

With both of these results in mind, it is safe to say that the bare metal version is the one more suited for the target application, as regular reaction time spikes of almost a microsecond are just not acceptable and the lower average iteration times are preferable.

# 6. Conclusion

We initially set out to create a flexible motor controller on a Raspberry Pi with the idea that different control schemes could easily be implemented in software and tested. To reach this goal, we looked at several different approaches, both from a detailed look at the implementation as well as the resulting performance characteristics. For all of the approaches, we used the programming language Rust, whose role in the embedded world is still an area of active research. While a lot of research focuses on how Rust's powerful static analysis can be applied to embedded systems, we examined the interaction of Rust with C/C++ libraries.

The different versions we developed were:

1. A bare-metal version using the Circle C++ library to access peripherals
2. A version running on the standard Linux kernel using the Rust native rppal library for peripheral access
3. A variant of the Linux version that had one core reserved for it to avoid being interrupted or having to wait for a core to be available.
4. A second variant of the Linux version that exchanges the default Linux kernel with one patched with a real-time patchset.

When testing these for performance, we found two main results. For one, the Linux-based versions all exhibited regular iteration time spikes, of up to 2.5 milliseconds, while the regular run time was more around 40 microseconds. In this behavior, the Linux-based versions did not differ significantly from each other, so for this workload, our approaches at optimizing the thread allocation were unsuccessful. Second, the bare metal version performed about 10 times faster than the Linux based versions, all while not having the latency spike problem.

Regarding the usage of Rust, we found that most of the added complexity of adding another language is not in the language interaction itself, but rather in the surrounding tooling such as managing multiple compilers, linkers, and build systems to all work together. This

can be seen both as positive and as a negative. The positive side is that the language tools such as bindgen for generating the bindings to the C code are in good shape both from a usability and stability perspective, while the negative side is that the added build system complexity might deter people who are just trying to code the actual program.

This leaves several directions in which this research could be expanded in future work. Writing the bare-metal program in C for a performance comparison between Rust with a C++ library and pure C++ would be a logical follow-up to the embedded Rust work. And, on a different route, an in-depth analysis of why and how the latency spikes for the Linux version occur could yield results on how to achieve comparable performance to the bare-metal version.

# Bibliography


[Adam 21] G. K. Adam, “Real-Time Performance and Response Latency Measurements of Linux Kernels on Single-Board Computers”, *Computers*, vol. 10, no. 5, 2021.

[Ayers 22] H. Ayers, P. Dutta, P. Levis, A. Levy, P. Pannuto, J. Van Why, J.-L. Watson, “Tiered Trust for Useful Embedded Systems Security”, in *Proceedings of the 15th European Workshop on Systems Security*, ser. EuroSec ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 15–21.

[Carbon 22] “Carbon Language: An experimental successor to C++”, https://github.com/carbon-language/carbon-lang, 2022. Accessed on 27.03.2024.

[Carvalho 19] A. Carvalho, C. Machado, F. Moraes, “Raspberry Pi Performance Analysis in Real-Time Applications with the RT-Preempt Patch”, in *2019 Latin American Robotics Symposium (LARS), 2019 Brazilian Symposium on Robotics (SBR) and 2019 Workshop on Robotics in Education (WRE)*. 2019, pp. 162–167.

[Culic 22] I. Culic, A. Vochescu, A. Radovici, “A Low-Latency Optimization of a Rust-Based Secure Operating System for Embedded Devices”, *Sensors*, vol. 22, no. 22, 2022.

[Dewit 24] W. Dewit, A. Paolillo, J. Gossens, “A Preliminary Assessment of the real-time capabilities of Real-Time Linux on Raspberry Pi 5”, in *OSPERT 2024 18th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*. 2024, pp. 7–12.

[Embedded Rust 15] “The Rust Programming Language - Embedded Devices”, https://www.rust-lang.org/what/embedded, 2015. Accessed on 27.03.2024.

[Go 09] “Build simple, secure, scalable systems with Go”, https://go.dev/, 2009. Accessed on 27.03.2024.

[iC-Haus GmbH 21] iC-Haus GmbH. *iC-MU MAGNETIC OFF-AXIS POSITION ENCODER - POLE WIDTH 1.28MM*. 2021.

[Klabnik 18] S. Klabnik, C. Nichols, *The Rust Programming Language*, San Francisco: No Starch Press, 2018.

[Levy 15] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, P. Pannuto, “Ownership is theft: experiences building an embedded OS in rust”, in *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*,

ser. PLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 21–26.

[Levy 17a] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, P. Levis, "Multiprogramming a 64kB Computer Safely and Efficiently", in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP'17. New York, NY, USA: ACM, 10 2017, pp. 234–251.

[Levy 17b] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, P. Levis, "The Case for Writing a Kernel in Rust", in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ser. APSys '17. New York, NY, USA: ACM, 9 2017, pp. 1:1–1:7.

[Li 23] Y. Li, Y. Matsubara, H. Takada, K. Suzuki, H. Murata, "A Performance Evaluation of Embedded Multi-core Mixed-criticality System Based on PREEMPT_RT Linux", *Journal of Information Processing*, vol. 31, pp. 78–87, 2023.

[Marwedel 21] P. Marwedel, "System Software", in *Embedded System Design*, Switzerland: Springer International Publishing AG, 2021.

[Meroth 21] A. Meroth, P. Sora, *Sensornetzwerke in Theorie und Praxis: Embedded Systems-Projekte erfolgreich realisieren*, Springer Fachmedien Wiesbaden, 2021.

[Opdenacker 23] M. Opdenacker, "RT patch set size", https://www.linkedin.com/posts/michaelopdenacker_here-is-an-update-of-our-diagram-following-activity-6983362119823994881-Z8V5, 2023. Accessed on 1.8.2024.

[PREEMPT-RT 24] "Real-Time Linux", https://wiki.linuxfoundation.org/realtime/start, 2024. Accessed on 1.8.2024.

[Raspberry Pi Ltd 22] Raspberry Pi Ltd. *BCM2711 ARM Peripherals*. 2022.

[RTAI 22] "RTAI - Real Time Application Interface", https://www.rtai.org, 2022. Accessed on 1.8.2024.

[Rustonomicon 24] "The Rustonomicon - The Dark Arts of Unsafe Rust", https://doc.rust-lang.org/nomicon/, 2024. Accessed on 8.01.2024.

[Schütz 20] S. Schütz, "CARL–A Compliant Robotic Leg Designed for Human-Like Bipedal Locomotion", Dissertation, Technische Universität Kaiserslautern, Kaiserslautern, 2020. https://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/5975.

[Sharma 23] A. Sharma, S. Sharma, S. Torres-Arias, A. Machiry, "Rust for Embedded Systems: Current State, Challenges and Open Problems", 2023.

[Xenomai 24] "Xenomai Real-time Linux", https://xenomai.org, 2024. Accessed on 1.8.2024.

[Zig 23] "Zig is a general-purpose programming language and toolchain for maintaining robust, optimal and reusable software.", https://ziglang.org/, 2023. Accessed on 27.03.2024.

# A. Appendix

## A.1 Unabridged code

Here is the full context for all code samples provided in the thesis

### A.1.1 bare-metal .cargo/config.toml

```
[target.aarch64-unknown-none]
linker = "aarch64-none-elf-ld"
rustflags = ["-C", "link-args=--section-start=.init=0x80000␣-Tcircle.ld"]

[build]
target = "aarch64-unknown-none"
```

### A.1.2 bare-metal build.rs

```
use bindgen::{Builder, MacroTypeVariation};
use std::{env, path::PathBuf};

fn main() {
    let manifest_dir = PathBuf::from(env::var("CARGO_MANIFEST_DIR").unwrap());
    println!(
        "cargo:rustc-link-search={}",
        manifest_dir.join("circle").display()
    ); // for circle.ld
    println!(
        "cargo:rustc-link-search={}",
        manifest_dir.join("circle/lib/usb").display()
    ); // for libusb.a
```

```
    println!(
        "cargo:rustc-link-search={}",
        manifest_dir.join("circle/lib/input").display()
    ); // for libinput.a
    println!(
        "cargo:rustc-link-search={}",
        manifest_dir.join("circle/lib/fs/fat").display()
    ); // for libfatfs.a
    println!(
        "cargo:rustc-link-search={}",
        manifest_dir.join("circle/lib/fs").display()
    ); // for libfs.a
    println!(
        "cargo:rustc-link-search={}",
        manifest_dir.join("circle/lib").display()
    ); // for libcircle.a

    println!("cargo:rustc-link-lib=usb");
    println!("cargo:rustc-link-lib=input");
    println!("cargo:rustc-link-lib=fatfs");
    println!("cargo:rustc-link-lib=fs");
    println!("cargo:rustc-link-lib=circle");

    let bindings = Builder::default()
        .header("wrapper.hpp")
        .use_core()
        .clang_arg(format!(
            "-I{}",
            manifest_dir.join("circle/include").display()
        ))
        .vtable_generation(true)
        .default_macro_constant_type(MacroTypeVariation::Signed)
        .parse_callbacks(Box::new(bindgen::CargoCallbacks::new()))
        .generate()
        .expect("Unable to generate bindings");

    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
    bindings
        .write_to_file(out_path.join("bindings.rs"))
        .expect("Couldn't write bindings!");
}
```

### A.1.3 bare-metal Cargo.toml

```
[package]
name = "bare-metal"
```

```
version = "0.1.0"
edition = "2021"

[dependencies]
pid-ctrl = "0.1.4"

[build-dependencies]
bindgen = "0.69.4"

[profile.release]
lto = true
debug = true

[features]
default = [ "measure" ]
measure = []
```

### A.1.4 bare-metal src/ffi.rs

```
#![allow(non_upper_case_globals)]
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]
#![allow(improper_ctypes)]
#![allow(unused)]

include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
```

### A.1.5 bare-metal src/main.rs

```
#![no_std]
#![no_main]

extern crate alloc;
use alloc::{format, string::String, vec};
use core::{
    alloc::GlobalAlloc,
    ffi::{c_int, c_uint, c_void},
    panic::PanicInfo,
    ptr::null_mut,
};
use pid_ctrl::PidCtrl;

mod ffi;

// just reboot on rust panic
#[panic_handler]
```

```
18 unsafe fn panic_handler(_panic_info: &PanicInfo) -> ! {
19     ffi::reboot()
20 }
21
22 // use circle's allocator implementation for rust
23 struct CircleAllocator;
24 unsafe impl GlobalAlloc for CircleAllocator {
25     unsafe fn alloc(&self, layout: core::alloc::Layout) -> *mut u8 {
26         ffi::malloc(layout.size()) as *mut u8
27     }
28
29     unsafe fn dealloc(&self, ptr: *mut u8, _layout: core::alloc::Layout)
       {
30         ffi::free(ptr as *mut c_void)
31     }
32 }
33
34 #[global_allocator]
35 static ALLOCATOR: CircleAllocator = CircleAllocator;
36
37 #[no_mangle]
38 pub unsafe extern "C" fn main() -> c_int {
39     let mut act_led = ffi::CActLED::new(false);
40     let options = ffi::CKernelOptions::new();
41     let mut device_name_service = ffi::CDeviceNameService::new();
42     let mut screen = ffi::CScreenDevice::new(options.GetWidth(), options.
       GetHeight(), false, 0);
43     let mut interrupt_system = ffi::CInterruptSystem::new();
44     let mut serial = ffi::CSerialDevice::new(&mut interrupt_system, false,
       0);
45     let _exception_handler = ffi::CExceptionHandler::new();
46     let mut timer = ffi::CTimer::new(&mut interrupt_system);
47     let mut logger = ffi::CLogger::new(options.GetLogLevel(), &mut timer,
       true);
48     let mut usb_hci =
49         ffi::CXHCIDevice::new(&mut interrupt_system, &mut timer, false, 0,
       null_mut());
50     let mut filesystem = ffi::CFATFileSystem::new();
51
52     screen.Initialize();
53     serial.Initialize(115200, 8, 1, ffi::CSerialDevice_TParity_ParityNone
       );
54     let mut log_device = device_name_service.GetDevice(options.
       GetLogDevice(), false);
55     if log_device.is_null() {
56         log_device = &mut serial._base;
57     }
```

```
58      logger.Initialize(log_device);
59      interrupt_system.Initialize();
60      timer.Initialize();
61      ((*usb_hci._base._base.vtable_).CUSBController_Initialize)(&mut
        usb_hci._base._base, true);
62
63      const SPI_FREQ: c_uint = 20_000_000;
64      const CPOL: c_uint = 0;
65      const CPHA: c_uint = 0;
66      const SPI_DEVICE: c_uint = 0;
67      let mut spi = ffi::CSPIMaster::new(SPI_FREQ, CPOL, CPHA, SPI_DEVICE);
68      spi.Initialize();
69
70      let _pwm_pin = ffi::CGPIOPin::new1(
71          18,
72          ffi::TGPIOMode_GPIOModeAlternateFunction5,
73          core::ptr::null_mut(),
74      );
75      let _pwm_pin_19 = ffi::CGPIOPin::new1(
76          19,
77          ffi::TGPIOMode_GPIOModeAlternateFunction5,
78          core::ptr::null_mut(),
79      );
80      const PWM_RANGE: u32 = 1024;
81      let mut pwm = ffi::CPWMOutput::new(
82          ffi::TGPIOClockSource_GPIOClockSourceOscillator,
83          2,
84          PWM_RANGE,
85          true,
86      );
87
88      let mut pid = PidCtrl::new_with_pid(10., 1., 5.);
89
90      act_led.Blink(5, 500, 300);
91
92      pwm.Start();
93
94      let mut iteration_start = ffi::CTimer::GetClockTicks64();
95
96      #[cfg(feature = "measure")]
97      {
98          for _ in 0..10_000 {
99              let position = get_position(&mut spi);
100             let setpoint = get_setpoint();
101
102             pid.setpoint = setpoint;
103             let time = ffi::CTimer::GetClockTicks64() - iteration_start;
```

```
104             let output = pid
105                 .step(pid_ctrl::PidIn::new(
106                     position,
107                     f64::from(time as u32) / 1_000_000.,
108                 ))
109                 .out;
110             iteration_start = ffi::CTimer::GetClockTicks64();
111 
112             pwm.Write(
113                 ffi::PWM_CHANNEL1 as c_uint,
114                 ((output * PWM_RANGE as f64).clamp(0., PWM_RANGE as f64))
    as c_uint,
115             );
116         }
117         const N: usize = 1_000_000;
118         let mut times = vec![0; N];
119         for time in times.iter_mut() {
120             let position = get_position(&mut spi);
121             let setpoint = get_setpoint();
122 
123             pid.setpoint = setpoint;
124             *time = ffi::CTimer::GetClockTicks64() - iteration_start;
125             let output = pid
126                 .step(pid_ctrl::PidIn::new(
127                     position,
128                     f64::from(*time as u32) / 1_000_000.,
129                 ))
130                 .out;
131             iteration_start = ffi::CTimer::GetClockTicks64();
132 
133             pwm.Write(
134                 ffi::PWM_CHANNEL1 as c_uint,
135                 ((output * PWM_RANGE as f64).clamp(0., PWM_RANGE as f64))
    as c_uint,
136             );
137         }
138 
139         let partition = device_name_service.GetDevice(c"umsd1-1".as_ptr(),
    true);
140         filesystem.Mount(partition);
141 
142         let file = filesystem.FileCreate(c"times.csv".as_ptr());
143         let mut buffer = String::from("iteration,elapsed_time_us\n");
144         for (n, time) in times[1..].iter().enumerate() {
145             buffer.push_str(&format!("{},{}\n", n, time));
146         }
147         let buffer = alloc::ffi::CString::new(buffer).unwrap();
```

```
148
149         filesystem.FileWrite(
150             file,
151             buffer.as_ptr() as *const c_void,
152             buffer.count_bytes() as u32,
153         );
154         filesystem.FileClose(file);
155         filesystem.UnMount();
156
157         ffi::halt()
158     }
159
160     #[cfg(not(feature = "measure"))]
161     loop {
162         let position = get_position(&mut spi);
163         let setpoint = get_setpoint();
164
165         pid.setpoint = setpoint;
166         let time = ffi::CTimer::GetClockTicks64() - iteration_start;
167         let output = pid
168             .step(pid_ctrl::PidIn::new(
169                 position,
170                 f64::from(time as u32) / 1_000_000.,
171             ))
172             .out;
173         iteration_start = ffi::CTimer::GetClockTicks64();
174
175         pwm.Write(
176             PWM_CHANNEL1 as c_uint,
177             ((output * PWM_RANGE as f64).clamp(0., PWM_RANGE as f64)) as
    c_uint,
178         );
179     }
180 }
181
182 unsafe fn get_position(spi: &mut ffi::CSPIMaster) -> f64 {
183     const SDAD_TRANSMISSION: u8 = 0xa6;
184     let buf_write = [SDAD_TRANSMISSION, 0, 0, 0, 0];
185     let mut buf_read = [0; 5];
186
187     const CS: u32 = 0;
188
189     spi.WriteRead(
190         CS,
191         buf_write.as_ptr() as *const c_void,
192         buf_read.as_mut_ptr() as *mut c_void,
193         5,
```

```
194     );
195
196     let position = u32::from_be_bytes(buf_read[1..].try_into().unwrap());
197
198     position as f64
199 }
200
201 pub fn get_setpoint() -> f64 {
202     0.
203 }
```

### A.1.6 bare-metal wrapper.hpp

```
1  #define PREFIX64 aarch64-none-elf-
2  #define AARCH 64
3  #define RASPPI 4
4
5  #include "circle/actled.h"
6  #include "circle/alloc.h"
7  #include "circle/devicenameservice.h"
8  #include "circle/exceptionhandler.h"
9  #include "circle/fs/fat/fatfs.h"
10 #include "circle/gpiopin.h"
11 #include "circle/interrupt.h"
12 #include "circle/koptions.h"
13 #include "circle/logger.h"
14 #include "circle/pwmoutput.h"
15 #include "circle/screen.h"
16 #include "circle/serial.h"
17 #include "circle/spimaster.h"
18 #include "circle/startup.h"
19 #include "circle/timer.h"
20 #include "circle/usb/usbhcidevice.h"
```

### A.1.7 Rust native HAL build.rs

```
1 use std::{env, fs, path::PathBuf};
2
3 fn main() {
4     let out_dir = PathBuf::from(env::var("OUT_DIR").unwrap());
5     println!("cargo:rustc-link-search={}", out_dir.display());
6
7     fs::copy("link.x", out_dir.join("link.x")).unwrap();
8     println!("cargo:rerun-if-changed=link.x");
9 }
```

### A.1.8 Rust native HAL Cargo.toml

```
[package]
name = "bcm2711-hal"
version = "0.1.0"
edition = "2021"

[dependencies]
aarch64-cpu = "9.4.0"
bcm2711-lpa = { version = "0.4.0", features = ["critical-section"] }
critical-section = "1.1.2"
embedded-hal = "1.0.0"
```

### A.1.9 Rust native HAL link.x

```
ENTRY(__start);

SECTIONS {
    .text 0x80000: {
        *(.text.__start)
    }

    .rodata : {
        *(.rodata .rodata.*)
    }

    .bss : {
        *(.bss .bss.*)
    }

    .data : {
        *(.data .data.*)
    }
}
```

### A.1.10 Rust native HAL src/critical_section.rs

```
use aarch64_cpu::registers::Writeable;
use aarch64_cpu::registers::DAIF;
use critical_section::RawRestoreState;

struct SingleCoreCS;
critical_section::set_impl!(SingleCoreCS);

unsafe impl critical_section::Impl for SingleCoreCS {
```

```
9       unsafe fn acquire() -> RawRestoreState {
10          DAIF.write(DAIF::A::Unmasked + DAIF::I::Unmasked + DAIF::F::
        Unmasked);
11      }
12
13      unsafe fn release(_: RawRestoreState) {
14          DAIF.write(DAIF::A::Masked + DAIF::I::Masked + DAIF::F::Masked);
15      }
16  }
```

### A.1.11 Rust native HAL src/entry.rs

```
1   use core::arch::asm;
2
3   use aarch64_cpu::{
4       asm::wfe,
5       registers::{Readable, MPIDR_EL1, MPIDR_EL1::Aff0},
6   };
7
8   #[no_mangle]
9   pub unsafe extern "C" fn __start() -> ! {
10      if MPIDR_EL1.read(Aff0) == 0 {
11          // safe because MPIDR_EL1.read() already checks if we're on
        aarch64
12          asm!("bl␣__start_rust");
13      }
14
15      loop {
16          wfe()
17      }
18  }
19
20  #[macro_export]
21  macro_rules! entry {
22      ($path:path) => {
23          #[no_mangle]
24          pub unsafe extern "C" fn __start_rust() -> ! {
25              let f: fn() -> ! = $path;
26
27              f()
28          }
29      };
30  }
```

### A.1.12 Rust native HAL src/gpio.rs

```
use core::marker::PhantomData;

pub trait GpioExt {
    type Parts;

    fn split(self) -> Self::Parts;
}

pub struct Unknown;

pub struct Input<MODE> {
    _mode: PhantomData<MODE>,
}

pub struct Floating;
pub struct PullDown;
pub struct PullUp;

pub struct Output;

// GPFselx, gpsetx, gpclrx
macro_rules! gpio {
    ([$($PXi:ident: (
        $pxi:ident,
        $gpfselx:ident,
        $fselx:ident,
        $gplevx:ident,
        $levx:ident,
        $gpio_pup_pdn_cntrl_regx:ident,
        $gpio_pup_pdn_cntrlx:ident,
        $gpsetx:ident,
        $setx:ident,
        $gpclrx:ident,
        $clrx:ident,
        $MODE:ty
    ),)+]) => {
        use bcm2711_lpa::GPIO;
        use embedded_hal as hal;
        use core::convert::Infallible;

        pub struct Parts {
            $(
                pub $pxi: $PXi<$MODE>,
            )+
        }

```

```
47          impl GpioExt for GPIO {
48              type Parts = Parts;
49
50              fn split(self) -> Parts {
51                  Parts {
52                      $(
53                          $pxi: $PXi { _mode: PhantomData },
54                      )+
55                  }
56              }
57          }
58
59          $(
60              pub struct $PXi<MODE> {
61                  _mode: PhantomData<MODE>,
62              }
63
64              impl<MODE> $PXi<MODE> {
65                  pub fn into_input(self) -> $PXi<Input<Floating>> {
66                      unsafe { (*GPIO::PTR).$gpio_pup_pdn_cntrl_regx().
    set_bits(|w| {w.$gpio_pup_pdn_cntrlx().none()})}
67
68                      unsafe { (*GPIO::PTR).$gpfselx().set_bits(|w| {w.
    $fselx().input()}) };
69
70                      $PXi { _mode: PhantomData }
71                  }
72
73                  pub fn into_input_pulldown(self) -> $PXi<Input<PullDown>>
     {
74                      unsafe { (*GPIO::PTR).$gpio_pup_pdn_cntrl_regx().
    set_bits(|w| {w.$gpio_pup_pdn_cntrlx().down()})}
75
76                      unsafe { (*GPIO::PTR).$gpfselx().set_bits(|w| {w.
    $fselx().input()}) };
77
78                      $PXi { _mode: PhantomData }
79                  }
80
81                  pub fn into_input_pullup(self) -> $PXi<Input<PullUp>> {
82                      unsafe { (*GPIO::PTR).$gpio_pup_pdn_cntrl_regx().
    set_bits(|w| {w.$gpio_pup_pdn_cntrlx().up()})}
83
84                      unsafe { (*GPIO::PTR).$gpfselx().set_bits(|w| {w.
    $fselx().input()}) };
85
86                      $PXi { _mode: PhantomData }
```

```
87                  }
88
89                  pub fn into_output(self) -> $PXi<Output> {
90                      unsafe { (*GPIO::PTR).$gpfselx().set_bits(|w| {w.
    $fselx().output()})}
91
92                      $PXi { _mode: PhantomData }
93                  }
94
95                  pub fn into_output_low(self) -> $PXi<Output> {
96                      unsafe { (*GPIO::PTR).$gpfselx().set_bits(|w| {w.
    $fselx().output()})}
97
98                      $PXi { _mode: PhantomData }
99                  }
100
101                 pub fn into_output_high(self) -> $PXi<Output> {
102                     unsafe { (*GPIO::PTR).$gpfselx().set_bits(|w| {w.
    $fselx().output()})}
103
104                     $PXi { _mode: PhantomData }
105                 }
106             }
107
108             impl<MODE> hal::digital::InputPin for $PXi<Input<MODE>> {
109                 fn is_high(&mut self) -> Result<bool, Self::Error>{
110                     Ok(unsafe{ (*GPIO::PTR).$gplevx().read().$levx().
    bit_is_set() })
111                 }
112
113                 fn is_low(&mut self) -> Result<bool, Self::Error> {
114                     Ok(unsafe{ (*GPIO::PTR).$gplevx().read().$levx().
    bit_is_clear() })
115                 }
116             }
117
118             impl<MODE> hal::digital::ErrorType for $PXi<MODE> {
119                 type Error = Infallible;
120             }
121
122             impl hal::digital::OutputPin for $PXi<Output> {
123                 fn set_low(&mut self) -> Result<(), Self::Error> {
124                     Ok(unsafe{ (*GPIO::PTR).$gpsetx().write_with_zero(|w|
     {w.$setx().set_bit()}) })
125                 }
126
127                 fn set_high(&mut self) -> Result<(), Self::Error> {
```

```
128                     Ok(unsafe{ (*GPIO::PTR).$gpclrx().write_with_zero(|w|
    {w.$clrx().clear_bit_by_one()}) })
129                 }
130             }
131         )+
132     }
133 }
134
135 gpio!([
136     Pin0: (pin0, gpfsel0, fsel0, gplev0, lev0, gpio_pup_pdn_cntrl_reg0,
    gpio_pup_pdn_cntrl0, gpset0, set0, gpclr0, clr0, Unknown),
137     Pin1: (pin1, gpfsel0, fsel1, gplev0, lev1, gpio_pup_pdn_cntrl_reg0,
    gpio_pup_pdn_cntrl1, gpset0, set1, gpclr0, clr1, Unknown),
138     Pin2: (pin2, gpfsel0, fsel2, gplev0, lev2, gpio_pup_pdn_cntrl_reg0,
    gpio_pup_pdn_cntrl2, gpset0, set2, gpclr0, clr2, Unknown),
139     Pin3: (pin3, gpfsel0, fsel3, gplev0, lev3, gpio_pup_pdn_cntrl_reg0,
    gpio_pup_pdn_cntrl3, gpset0, set3, gpclr0, clr3, Unknown),
140     Pin4: (pin4, gpfsel0, fsel4, gplev0, lev4, gpio_pup_pdn_cntrl_reg0,
    gpio_pup_pdn_cntrl4, gpset0, set4, gpclr0, clr4, Unknown),
141     Pin5: (pin5, gpfsel0, fsel5, gplev0, lev5, gpio_pup_pdn_cntrl_reg0,
    gpio_pup_pdn_cntrl5, gpset0, set5, gpclr0, clr5, Unknown),
142     Pin6: (pin6, gpfsel0, fsel6, gplev0, lev6, gpio_pup_pdn_cntrl_reg0,
    gpio_pup_pdn_cntrl6, gpset0, set6, gpclr0, clr6, Unknown),
143     Pin7: (pin7, gpfsel0, fsel7, gplev0, lev7, gpio_pup_pdn_cntrl_reg0,
    gpio_pup_pdn_cntrl7, gpset0, set7, gpclr0, clr7, Unknown),
144     Pin8: (pin8, gpfsel0, fsel8, gplev0, lev8, gpio_pup_pdn_cntrl_reg0,
    gpio_pup_pdn_cntrl8, gpset0, set8, gpclr0, clr8, Unknown),
145     Pin9: (pin9, gpfsel0, fsel9, gplev0, lev9, gpio_pup_pdn_cntrl_reg0,
    gpio_pup_pdn_cntrl9, gpset0, set9, gpclr0, clr9, Unknown),
146     Pin10: (pin10, gpfsel1, fsel10, gplev0, lev10,
    gpio_pup_pdn_cntrl_reg0, gpio_pup_pdn_cntrl10, gpset0, set10, gpclr0,
    clr10, Unknown),
147     Pin11: (pin11, gpfsel1, fsel11, gplev0, lev11,
    gpio_pup_pdn_cntrl_reg0, gpio_pup_pdn_cntrl11, gpset0, set11, gpclr0,
    clr11, Unknown),
148     Pin12: (pin12, gpfsel1, fsel12, gplev0, lev12,
    gpio_pup_pdn_cntrl_reg0, gpio_pup_pdn_cntrl12, gpset0, set12, gpclr0,
    clr12, Unknown),
149     Pin13: (pin13, gpfsel1, fsel13, gplev0, lev13,
    gpio_pup_pdn_cntrl_reg0, gpio_pup_pdn_cntrl13, gpset0, set13, gpclr0,
    clr13, Unknown),
150     Pin14: (pin14, gpfsel1, fsel14, gplev0, lev14,
    gpio_pup_pdn_cntrl_reg0, gpio_pup_pdn_cntrl14, gpset0, set14, gpclr0,
    clr14, Unknown),
151     Pin15: (pin15, gpfsel1, fsel15, gplev0, lev15,
    gpio_pup_pdn_cntrl_reg0, gpio_pup_pdn_cntrl15, gpset0, set15, gpclr0,
    clr15, Unknown),
```

```
152     Pin16: (pin16, gpfsel1, fsel16, gplev0, lev16,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl16, gpset0, set16, gpclr0,
        clr16, Unknown),
153     Pin17: (pin17, gpfsel1, fsel17, gplev0, lev17,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl17, gpset0, set17, gpclr0,
        clr17, Unknown),
154     Pin18: (pin18, gpfsel1, fsel18, gplev0, lev18,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl18, gpset0, set18, gpclr0,
        clr18, Unknown),
155     Pin19: (pin19, gpfsel1, fsel19, gplev0, lev19,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl19, gpset0, set19, gpclr0,
        clr19, Unknown),
156     Pin20: (pin20, gpfsel2, fsel20, gplev0, lev20,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl20, gpset0, set20, gpclr0,
        clr20, Unknown),
157     Pin21: (pin21, gpfsel2, fsel21, gplev0, lev21,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl21, gpset0, set21, gpclr0,
        clr21, Unknown),
158     Pin22: (pin22, gpfsel2, fsel22, gplev0, lev22,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl22, gpset0, set22, gpclr0,
        clr22, Unknown),
159     Pin23: (pin23, gpfsel2, fsel23, gplev0, lev23,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl23, gpset0, set23, gpclr0,
        clr23, Unknown),
160     Pin24: (pin24, gpfsel2, fsel24, gplev0, lev24,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl24, gpset0, set24, gpclr0,
        clr24, Unknown),
161     Pin25: (pin25, gpfsel2, fsel25, gplev0, lev25,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl25, gpset0, set25, gpclr0,
        clr25, Unknown),
162     Pin26: (pin26, gpfsel2, fsel26, gplev0, lev26,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl26, gpset0, set26, gpclr0,
        clr26, Unknown),
163     Pin27: (pin27, gpfsel2, fsel27, gplev0, lev27,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl27, gpset0, set27, gpclr0,
        clr27, Unknown),
164     Pin28: (pin28, gpfsel2, fsel28, gplev0, lev28,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl28, gpset0, set28, gpclr0,
        clr28, Unknown),
165     Pin29: (pin29, gpfsel2, fsel29, gplev0, lev29,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl29, gpset0, set29, gpclr0,
        clr29, Unknown),
166     Pin30: (pin30, gpfsel3, fsel30, gplev0, lev30,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl30, gpset0, set30, gpclr0,
        clr30, Unknown),
167     Pin31: (pin31, gpfsel3, fsel31, gplev0, lev31,
        gpio_pup_pdn_cntrl_reg1, gpio_pup_pdn_cntrl31, gpset0, set31, gpclr0,
```

```
    clr31, Unknown),
168  Pin32: (pin32, gpfsel3, fsel32, gplev1, lev32,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl32, gpset1, set32, gpclr1,
    clr32, Unknown),
169  Pin33: (pin33, gpfsel3, fsel33, gplev1, lev33,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl33, gpset1, set33, gpclr1,
    clr33, Unknown),
170  Pin34: (pin34, gpfsel3, fsel34, gplev1, lev34,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl34, gpset1, set34, gpclr1,
    clr34, Unknown),
171  Pin35: (pin35, gpfsel3, fsel35, gplev1, lev35,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl35, gpset1, set35, gpclr1,
    clr35, Unknown),
172  Pin36: (pin36, gpfsel3, fsel36, gplev1, lev36,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl36, gpset1, set36, gpclr1,
    clr36, Unknown),
173  Pin37: (pin37, gpfsel3, fsel37, gplev1, lev37,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl37, gpset1, set37, gpclr1,
    clr37, Unknown),
174  Pin38: (pin38, gpfsel3, fsel38, gplev1, lev38,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl38, gpset1, set38, gpclr1,
    clr38, Unknown),
175  Pin39: (pin39, gpfsel3, fsel39, gplev1, lev39,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl39, gpset1, set39, gpclr1,
    clr39, Unknown),
176  Pin40: (pin40, gpfsel4, fsel40, gplev1, lev40,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl40, gpset1, set40, gpclr1,
    clr40, Unknown),
177  Pin41: (pin41, gpfsel4, fsel41, gplev1, lev41,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl41, gpset1, set41, gpclr1,
    clr41, Unknown),
178  Pin42: (pin42, gpfsel4, fsel42, gplev1, lev42,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl42, gpset1, set42, gpclr1,
    clr42, Unknown),
179  Pin43: (pin43, gpfsel4, fsel43, gplev1, lev43,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl43, gpset1, set43, gpclr1,
    clr43, Unknown),
180  Pin44: (pin44, gpfsel4, fsel44, gplev1, lev44,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl44, gpset1, set44, gpclr1,
    clr44, Unknown),
181  Pin45: (pin45, gpfsel4, fsel45, gplev1, lev45,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl45, gpset1, set45, gpclr1,
    clr45, Unknown),
182  Pin46: (pin46, gpfsel4, fsel46, gplev1, lev46,
    gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl46, gpset1, set46, gpclr1,
    clr46, Unknown),
```

```
183     Pin47: (pin47, gpfsel4, fsel47, gplev1, lev47,
        gpio_pup_pdn_cntrl_reg2, gpio_pup_pdn_cntrl47, gpset1, set47, gpclr1,
        clr47, Unknown),
184     Pin48: (pin48, gpfsel4, fsel48, gplev1, lev48,
        gpio_pup_pdn_cntrl_reg3, gpio_pup_pdn_cntrl48, gpset1, set48, gpclr1,
        clr48, Unknown),
185     Pin49: (pin49, gpfsel4, fsel49, gplev1, lev49,
        gpio_pup_pdn_cntrl_reg3, gpio_pup_pdn_cntrl49, gpset1, set49, gpclr1,
        clr49, Unknown),
186     Pin50: (pin50, gpfsel5, fsel50, gplev1, lev50,
        gpio_pup_pdn_cntrl_reg3, gpio_pup_pdn_cntrl50, gpset1, set50, gpclr1,
        clr50, Unknown),
187     Pin51: (pin51, gpfsel5, fsel51, gplev1, lev51,
        gpio_pup_pdn_cntrl_reg3, gpio_pup_pdn_cntrl51, gpset1, set51, gpclr1,
        clr51, Unknown),
188     Pin52: (pin52, gpfsel5, fsel52, gplev1, lev52,
        gpio_pup_pdn_cntrl_reg3, gpio_pup_pdn_cntrl52, gpset1, set52, gpclr1,
        clr52, Unknown),
189     Pin53: (pin53, gpfsel5, fsel53, gplev1, lev53,
        gpio_pup_pdn_cntrl_reg3, gpio_pup_pdn_cntrl53, gpset1, set53, gpclr1,
        clr53, Unknown),
190     Pin54: (pin54, gpfsel5, fsel54, gplev1, lev54,
        gpio_pup_pdn_cntrl_reg3, gpio_pup_pdn_cntrl54, gpset1, set54, gpclr1,
        clr54, Unknown),
191     Pin55: (pin55, gpfsel5, fsel55, gplev1, lev55,
        gpio_pup_pdn_cntrl_reg3, gpio_pup_pdn_cntrl55, gpset1, set55, gpclr1,
        clr55, Unknown),
192     Pin56: (pin56, gpfsel5, fsel56, gplev1, lev56,
        gpio_pup_pdn_cntrl_reg3, gpio_pup_pdn_cntrl56, gpset1, set56, gpclr1,
        clr56, Unknown),
193     Pin57: (pin57, gpfsel5, fsel57, gplev1, lev57,
        gpio_pup_pdn_cntrl_reg3, gpio_pup_pdn_cntrl57, gpset1, set57, gpclr1,
        clr57, Unknown),
194 ]);
```

### A.1.13 Rust native HAL lib.rs

```
1 #![no_std]
2
3 pub use bcm2711_lpa as pac;
4
5 pub mod gpio;
6
7 mod critical_section;
8 mod entry;
```