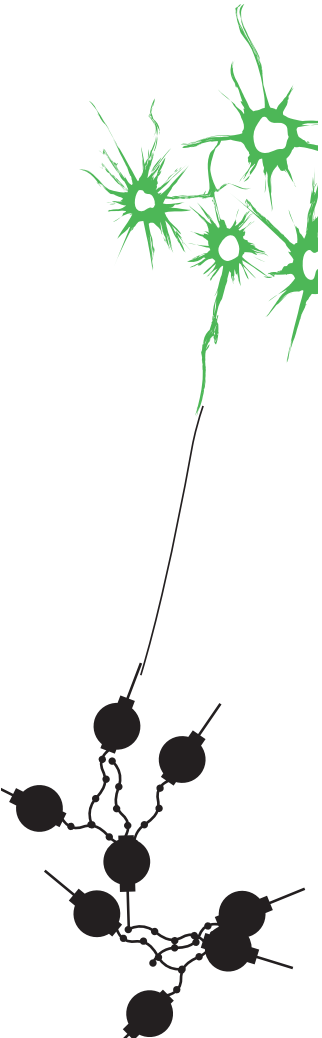

RAM ROBOTICS AND MECHATRONICS

REAL-TIME INTERRUPT-DRIVEN CONCURRENCY (RTIC) FOR MODERN APPLICATION PROCESSORS

J.K. (Jurgis) Balciunas

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
dr. ir. G. van Oort
ing. M.H. Schwirtz
dr. ing. K.H. Chen

December, 2021

077RaM2021
Robotics and Mechatronics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Summary

The Real-Time Interrupt-driven Concurrency (RTIC) framework (Lindgren et al., 2021) showed to be successful in developing guaranteed memory-safe and deadlock-free real-time applications on the Arm Cortex-M based microcontrollers. Based on this work, the project analyses the RTIC framework compatibility with modern application processors, emphasizing co-existence with the Linux kernel for integrating real-time and non-real-time applications on a single system. Various Cortex-A bare-metal and Linux kernel based RTIC implementation approaches are analyzed and the chosen Linux user-space thread-based RTIC implementation is realised. Benchmarking results showed that non-deterministic latencies introduced by the Cortex-A architecture are insignificant when compared to the scheduling overhead of the real-time patched Linux kernel. Nevertheless, the developed RTIC implementation is demonstrated by successfully implementing a real-time two-axis gimbal control application, featuring a 10 kHz cascaded motor current PID controller and a seamless integration with a non-real-time ROS software. Further improvements could be made by analyzing the Linux kernel scheduler bottlenecks or by exploring the alternative bare-metal approach.

Contents

1 Introduction

1.1 Context

The advent of robots is no longer a science fiction. They have been an integral part of industrial automation for multiple decades, but now, with advancements in technology and declining prices they have also found a way into everyday life: from robot vacuum cleaners to autonomous vehicles. It is clear that we will see more and more robots replace our everyday tasks.

With robot applications rapidly advancing, the control software is also getting increasingly complex. Early robots would only control a few motorized axes, which was purely within domain of real-time control systems. However, nowadays robots include a wide range of non real-time software ranging from Artificial Intelligence (AI) to Graphical User Interface (GUI). Combining such diverse software on a single system is difficult, because of different requirements.

Non real-time software usually has very large dependency trees and is tightly coupled to the Operating System's (OS) Application Programming Interface (API). This makes them almost impossible to run on Real-Time Operating System (RTOS), which are very lightweight and cannot support all the API features. On the other hand, General Purpose Operating System (GPOS) such as Linux are not designed for real-time systems and cannot guarantee execution deadlines by design.

This gap between real-time and non real-time software forces engineers to use multiple interconnected systems to serve different domains. However, not only this increases costs due to additional hardware, but also the additional communications layer greatly increases software complexity.

There are multiple projects attempting to bridge this gap. Most notable of them are the PREEMPT_RT Linux kernel patches (The Linux Foundation, 2015) and the Xenomai real-time layer for Linux (Gerum, 2001). These projects focus on low-level integration with Linux and their API consists of primitive building blocks, similar to the traditional RTOSs.

Writing real-time applications on these low-level APIs allows achieving exceptional performance, but at the same time, it is very difficult to ensure correctness. Without proper verification techniques, such applications often have race conditions, deadlocks and memory safety issues. There are, however, alternatives. The Real-Time Interrupt-driven Concurrency (RTIC) framework (Lindgren et al., 2019) is designed for writing guaranteed memory-safe and deadlock free real-time applications. It not only ensures safety at compile time without any runtime cost, but also features a novel and very efficient scheduling technique. Unfortunately, the RTIC framework is currently only implemented for the Cortex-M based microcontrollers, but its concept of tasks and resources does not restrict it to any particular platform. As such, it might be a solution for writing robust real-time applications on Linux, without any of the low-level primitives.

Thus, the purpose of this work is to investigate how the RTIC framework could work on a Linux system, implement it and test its performance by controlling a robot. Moreover, because one of the driving forces behind robot adoption are declining prices, it is desirable to develop cost-effective solutions. For this reason, Raspberry Pi 4 (RPi4) Single Board Computer (SBC) was selected as a testing platform for the RTIC implementation in Linux. Not only it is a cheap SBC, but also one of the most popular choices for embedded applications, with readily available documentation, libraries and examples.

1.2 Goals

The goals of the project are:

- Investigate how RTIC could be implemented in Linux.
- Implement RTIC by the chosen approach.
- Benchmark how well it works for real-time applications.
- Demonstrate implementation by controlling a robot.

1.3 Outline

Chapter 2 gives the definition of real-time computing, explains inner workings of the RTIC framework and principles of the Rust programming language, presents the RPi4 architecture and its compatibility with RTIC.

Chapter 3 analyses various RTIC implementations: bare-metal, real-time Linux, Dovetail interface.

Chapter 4 covers the bare-metal application writing on RPi4 and resource sharing with Linux.

Chapter 5 is about implementing RTIC framework on the Linux user space threads. Various issues with the Linux kernel API are discussed, which caused difficulties implementing RTIC task and resource models.

Chapter 6 contains benchmarks. Firstly, General-Purpose Input/Output (GPIO) interrupts are used to measure Linux interrupt handling overhead as compared to bare-metal. Then RTIC implementation on Linux is evaluated by task switching and resource locking performance.

Chapter 7 demonstrates the Linux RTIC implementation by controlling a robot with a real-time controller.

Chapter 8 concludes the report by summarising all findings and giving recommendations.

2 Background

2.1 Real-Time Computing

The term "real-time computing" is used to define hardware and software systems, which must guarantee response within specified time constraints (deadlines). Such systems are often mission or safety critical, meaning that any unexpected processing delay can lead to catastrophic failure. Note that real-time is often understood as high-performance, but that is not correct, because real-time system is all about response latency. While high-performance system can have higher processing speed, its latency could be worse than that of a real-time system.

Real-time systems and their deadlines are usually classified by the consequences of missing a deadline (Buttazzo, 2011):

- Hard: A real-time task is said to be *hard* if producing the results after its deadline may cause catastrophic consequences for the system under control.
- Firm: A real-time task is said to be *firm* if producing the results after its deadline is useless for the system, but does not cause any damage.
- Soft: A real-time task is said to be *soft* if producing the results after its deadline has still some utility for the system, although causing a performance degradation.

Almost all real-time systems are implemented on microcontrollers, Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs). They usually do not have any nondeterministic execution delays and task Worst-Case Execution Time (WCET) can be calculated statically in the number of clock cycles. Hence, with proper scheduling algorithms it can be mathematically proven that no deadlines will be missed (Buttazzo, 2011).

2.1.1 Real-time on Modern Application Processors

Modern application processors that are used in smartphones, desktops and SBCs are much more complex than microcontrollers. They are designed to squeeze every bit of performance to meet the demands of non real-time applications and use multitude of optimisations to achieve that. Some of these optimisations can cause unpredictable processing delays, such as memory caching, DRAM refreshing, branch prediction, etc.

Such systems are very hard to model mathematically and it is almost impossible to statically calculate the WCET. When large OS is added on top, the only feasible way to ensure real-time constraints is to do statistical analysis: real-time systems are run for many iterations with various system stress tests and response latencies are recorded. Then a histogram is built to check latency distribution and the statistical WCET can be found. This is not hard real-time by definition, but with large enough sample size, a decent statistical confidence can be achieved.

2.2 Real-Time Interrupt-driven Concurrency

Most RTOSs provide thread-based abstractions for writing real-time applications, under which incorrect resource management usually leads to memory race conditions and deadlocks. Moreover, hardware interrupts are external to the thread model and special handing of resources is required, which adds even more complexity. RTIC (Lindgren et al., 2019) framework was designed to reduce this complexity and allow writing statically memory-safe and deadlock free applications, with an emphasis on efficiency.

RTIC was originally known as Real-Time For the Masses (RTFM) (Lindgren et al., 2015) and was developed at the Luleå University of Technology. It consisted of a C runtime, but required a separate Domain-Specific Language (DSL) to define tasks and resources. The DSL would then analyse task and resource relationships to generate the C code. This was inconvenient. Even-

tually, Aparicio (2017) came up with an implementation in the Rust language, which proved to be really successful. Not only Rust language brought memory safety, but also Rust's procedural macro system allowed to completely avoid using a DSL. Since then, all of the development has shifted from the original C implementation to the Rust one.

2.2.1 RTIC Task Model

The main unit of concurrency in RTIC are tasks. They can be event triggered (i.e. hardware interrupt) or spawned by the application on demand. Tasks always run to completion and do not have infinite loops unlike thread based applications. All tasks are prioritized and can be preempted by a higher priority task.

Tasks in RTIC can communicate in the following ways:

- Message passing when spawning another task.
- Access to task-local resources without locking.
- Access to shared resources with locking.

Contrary to common practices, RTIC tasks are actually executed inside hardware interrupt handlers. This results in an extremely efficient scheduling, because most of the work is done by the hardware; software scheduling overhead is minimal. To employ such scheduling, the hardware interrupt controller must support prioritised nested interrupts.

This naturally leads to two types of tasks: hardware and software. Hardware tasks are dedicated to servicing some peripheral and are bound to its interrupt. Software tasks can be arbitrary, but they are still scheduled by the same technique. They are assigned an unused interrupt vector and spawning is done by simply pending the interrupt. To reduce the amount of needed hardware interrupts, software tasks of the same priority are grouped into a single interrupt and are executed by popping from a First In First Out (FIFO) queue. Software tasks can also be scheduled to be executed at a specific time, which is handled by a hardware timer.

2.2.2 RTIC Resource Model

In multitasking systems, shared memory resources must be protected from concurrent access in order to avoid race conditions. The resource model, which protects shared resources, is a very important part of the RTIC framework, because in addition to memory safety, it has to prevent unbounded priority inversion and ensure deadlock-free execution. The two popular protocols, which satisfy all of these requirements are the Priority Ceiling Protocol (PCP) and the Stack Resource Policy (SRP).

Priority Ceiling Protocol

The PCP was introduced by Sha et al. (1990) to bound the priority inversion problem and to prevent the formation of deadlocks.

This protocol works similarly to the Priority Inheritance (PI) protocol, which increases task priority in a critical section when a higher priority task is blocked by it. However, to avoid deadlocks, it does not allow a task to enter a critical section if there are locked resources, which could block it. This is achieved by assigning priority ceiling to each resource, which is equal to the maximum priority of all tasks that use it. Then, a task is only allowed to lock a resource when its priority is higher than all of the priority ceilings of resources currently locked by the other tasks.

An example of PCP is shown in Figure 2.1. There are 3 tasks τ_1, τ_2, τ_3, in the order of decreasing priority. There are also two resources s_1 and s_2. Their priority ceilings $C(s_1)$ and $C(s_2)$ are equal to priority P_1 because they are both used by the highest priority task τ_1. Task τ_3 begins execution first, locks the resource s_1 and at the time t_1 it gets preempted by a higher priority

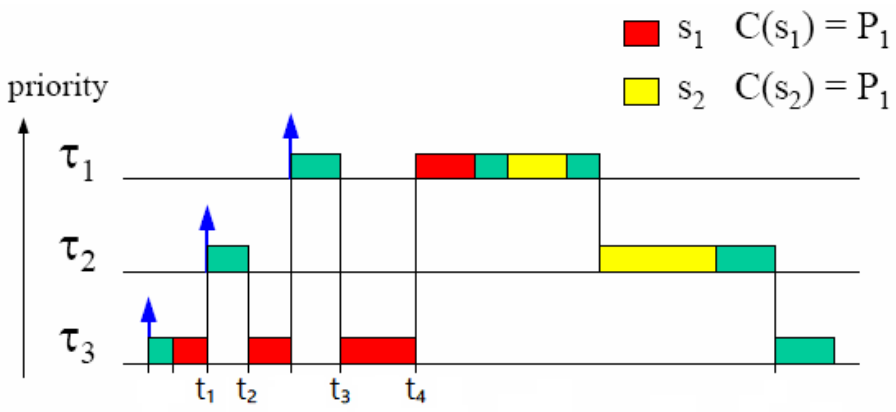

Figure 2.1: Example of Priority Ceiling Protocol (adapted from Ras and Cheng (2009))

task τ_2. At t_2, task τ_2 tries to lock a different resource s_2, but it is blocked by the PCP, because its priority P_2 is not higher than the ceiling of the currently locked resource s_1. This simultaneously increases the dynamic priority of τ_3 to P_2, because it is blocking τ_2. At t_3, the highest priority task τ_1 attempts to lock resource s_1, but is directly blocked because resource is already locked. Task τ_3 dynamic priority is now increased to P_1 (maximum priority). At t_4, task τ_3 finally unlocks resource s_1, its dynamic priority drops back to P_3 and task τ_1 is unblocked.

If PCP was not used, the task τ_3 would not inherit dynamic priority of blocked task and the highest priority task τ_1 would be waiting on τ_2 even when it is not directly blocked by it, but because the lower priority task τ_3 holding the resource s_1 could not proceed. The task τ_1 would be experiencing priority inversion.

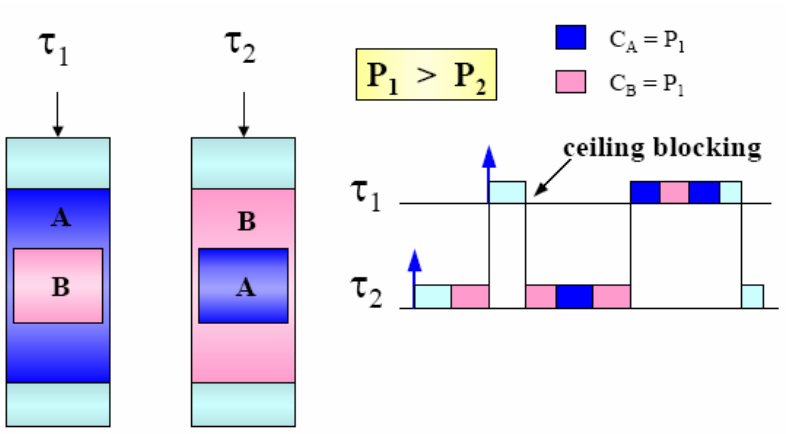

Figure 2.2: Priority Ceiling Protocol deadlock avoidance (Ras and Cheng, 2009)

Even though the priority inversion problem could also be solved by a simpler PI protocol, the additional complexity of PCP allows it to prevent deadlocks. This property is demonstrated in Figure 2.2. Two tasks τ_1 and τ_2 lock resources A and B in different order. Without PCP, it could so happen that task τ_2 is preempted while holding B lock, but not yet A. The higher priority task τ_1 would then lock resource A and could not continue any further, because resource B is locked by τ_2, resulting in a deadlock. However, when PCP is used, the higher priority task τ_1 is prevented from locking resource A, even though it is free. This is because task τ_1 priority P_1 is not higher than the currently locked resource B priority ceiling C_A.

Stack Resource Policy

The SRP was introduced by Baker (1990) as a refinement of the PCP. It reduces the amount of context switches and allows sharing of a single run-time stack.

SRP is similar to PCP, because it has the same concept of resource priority ceilings. The main difference is when tasks are blocked. In PCP tasks are blocked at the moment they attempt to lock a resource, while in SRP tasks are blocked earlier, when they try to preempt. This is achieved by raising task priority to the ceiling immediately when a resource is locked, without waiting for blocking to occur. Only tasks with priority higher than the current system ceiling can preempt.

Because of the early blocking, tasks in SRP have strict ordering and can use a single runtime stack. If a task is allowed to preempt, it is guaranteed that no lower priority task can block it until the task finishes. This implies that preempted task is only resumed when all higher priority tasks have finished and the runtime stack is at the same state when the task was preempted.

The RTIC developers chose to use SRP, because of the following two reasons:

- Sharing a single run-time stack is a huge advantage for small resource-constrained Cortex-M microcontrollers, because it saves memory.
- Implementing system priority ceiling in Cortex-M microcontrollers is as simple as masking all lower priority interrupts with a single register write. This is much simpler and faster than the PCP algorithm.

The main drawback of both PCP and SRP is that a static analysis of the program is needed to assign resource ceilings. Incorrect setting of the ceilings can also lead to race conditions, which puts a lot of burden on the developer. Hence, these protocols usually require additional tooling to ensure correctness.

However, in RTIC no additional tooling is needed. Resource ceiling analysis is done by the Rust procedural macros at compile time and ceilings are guaranteed to always be correct. This makes resource policy completely transparent to the programmer.

2.2.3 Framework Organisation

The RTIC framework consists of multiple crates (that is how Rust packages are called) and is designed with portability in mind. For example the Cortex-M RTIC implementation consists of 3 main parts:

- `rtic-syntax`: Parses and analyses application code, calculates resource ceilings. It is platform independent.
- `cortex-m-rtic-macros`: Generates platform-specific code from `rtic-syntax` output.
- `cortex-m-rtic`: Contains shared runtime code, used by both application and the expanded macro code.

The interaction between different parts is shown in Figure 2.3, which shows the compilation pipeline. User application code depends on the top level `cortex-m-rtic` crate and any other application-specific dependencies. This application code is then passed directly into Rust compiler (`rustc`), which notices the use of RTIC macro and expands it by firstly parsing with `rtic-syntax` crate and then generating code with `cortex-m-rtic-macros` crate. The now macro-free code can be compiled into application binary.

One of the drawbacks of RTIC is that it is officially only supported on the Cortex-M microcontrollers. The two `cortex-m-rtic` and `cortex-m-rtic-macros` crates are tightly coupled to the Nested Vector Interrupt Controller (NVIC) of Cortex-M architecture and cannot be easily ported. The third crate `rtic-syntax` is platform independent and can be used

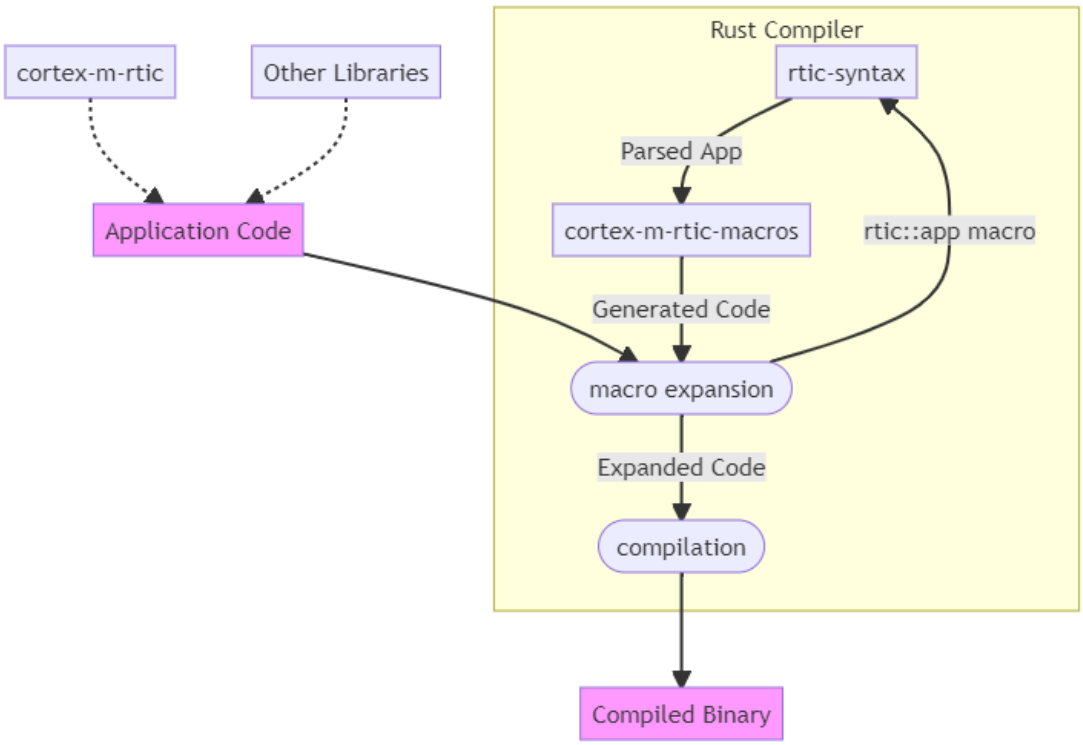

Figure 2.3: RTIC Application Compilation Steps

to implement other architectures. Multiple implementations for different architectures already exist, but they are not supported or maintained by the RTIC developer group.

One implementation of particular importance is the `linux-rtfm` (Aparicio, 2019). It implements RTIC in a user space Linux process. However, it is no longer maintained and is based on an old `rtic-syntax` version. Moreover, the implementation heavily relies on the x86 assembly code and is not compatible with the ARM architecture of RPi4. More details on this implementation are given in Section 5.1.

To better understand how RTIC actually works, see an application example in Appendix A.

2.3 Rust Language

Rust is a high-level general-purpose programming language that emphasizes performance and, especially, memory and concurrency safety. It started as a personal project by Graydon Hoare in 2006 and was soon sponsored by the Mozilla Corporation as a means to improve web browser security. Google LLC (2020) estimated that 70% of serious security bugs in their browser were memory safety problems. The Rust language was refined by the Mozilla developers while writing experimental web browser called Servo and the Rust compiler itself. The language went through many different iterations and changes, but has recently been stabilised. Rust has been voted the “most loved programming language” in the Stack Overflow Developer Survey every year since 2016 (Stack Overflow, 2021).

2.3.1 Rust Advantages

The first and foremost feature of Rust is memory safety. It hinges on the ownership concept and requires all variables to be initialized. When used, variables are moved by default (transfer of ownership) and can be used only once except when explicitly cloned. This concept is expanded with borrowing (referencing a variable) and the borrow checker, which allows using variables without transferring ownership. Rust ensures that at any time there may only exist either a

single mutable reference or multiple immutable references. The borrow checker verifies this by tracking lifetimes of variables and references. This also ensures that references never outlive their referenced variable. It is important to mention that everything is done at compile time and Rust's memory safety does not have any runtime costs.

This memory model is too strict to express some programming concepts, so Rust introduces the `unsafe` keyword, which lets developers explicitly violate memory safety rules. One example where `unsafe` is used, is implementing concurrency primitives such as mutex. By default it would not be possible to mutably access data inside mutex, because it is immutably shared between multiple threads. However, mutex implementation can use atomic variables (or OS API) to ensure exclusivity at runtime and then give out a mutable reference by an `unsafe` operation. Such code is usually limited to very small sections so that it is easy to prove correctness. It is very uncommon to use `unsafe` in application code and it is generally only used in lower level libraries.

Rust also emphasizes performance and closely follows "only pay for what you use" concept. This means that language minimises any hidden behavior and if there is a performance penalty, the operation must be explicit. Some examples of this is explicit cloning of variables with `clone` function or using `Box` type when heap allocation is desired. It is also common that Rust code actually runs faster than C/C++, because strict memory aliasing rules allows compiler to better optimise the code.

Apart from language features, there is also a package manager called `Cargo`. Packages in Rust are called crates and use the unified cargo build system. The Rust compiler and `Cargo` are the only tools needed to build even the most complex applications. All crates are published in the public crates registry `crates.io` (The Rust Foundation, 2021), which currently has over 70,000 crates.

Rust also has built-in documentation capabilities similar to Doxygen (van Heesch, 1997). Documentation is written next to code with triple slashes, which separates it from code comments that use double slashes. Documentation can be then generated with the same `Cargo` tool, which combines static code analysis with inline documentation to generate full API reference. The generated documentation can be viewed in HTML format. Because all published crates are required to follow this documentation format, you can always check documentation directly in `crates.io` registry. It is also possible to generate a single project documentation, which includes documentations of all used dependencies.

Finally, Rust was designed with interoperability in mind. It has a very similar memory model and call convention comparable to C, so that only type definitions are needed for Foreign Function Interface (FFI) and there is no additional overhead. However, it must be noted that calling C code from Rust is always `unsafe`, because C code itself is inherently not memory safe. To aid with integration, Rust also has crates such as `bindgen`, which automatically generates bindings from header files, and `cc` to compile C code during Rust build process.

2.3.2 Rust Shortcomings

Rust compiler does not generate machine code itself. Instead, it generates a standard `LLVM` Intermediate Representation (IR) code and then uses the well known `LLVM` compiler (Lattner and Adve, 2004) to emit platform-specific machine code. This lets Rust support many architectures without maintaining any platform-specific assembly code, but also comes as a drawback, because some platforms are not supported by the `LLVM` and use alternative compilers such as `GCC`. There is a `gccrs` project, which aims to implement `GCC` front-end for Rust, but it is in a very early stage.

Because Rust is a fairly new language, its crate ecosystem is still fairly small. Many of the shortcomings are plugged by binding crates for popular C libraries, which is often inconvenient due

to custom build systems and dependencies these external libraries use. However, the number of Rust-native libraries is constantly growing and even replacements for such complicated libraries as OpenSSL appear.

2.3.3 Rust for Embedded Systems

Most embedded systems heavily rely on correctness and memory safety is often a culprit of many hard to track down bugs. Rust's memory safety and focus on performance makes it a very good candidate for writing embedded software. Portable code and package manager also boosts productivity.

On caveat with embedded applications is that Rust's standard library is not available. Fortunately, Rust has split standard library into two parts: `core`, which contains functions without syscalls (memory operations, string formatting, etc), and `std`, which depends on syscalls (file system, networking, threading, etc). The later is not available when crate is marked with a special `no_std` attribute, meaning that the crate is suitable for embedded applications. Rust compiler enforces that all embedded application dependencies are `no_std`.

The Rust embedded ecosystem is growing quite rapidly. The most common Cortex-M microcontroller architecture is widely supported and safe Hardware Abstraction Layer (HAL) crates are available for popular chips such as ST STM32, Microchip SAM, PIC32 and Nordic Semiconductor NRF series. There is also a working group, which standardises peripheral API so that driver crates are portable and usable across different platforms.

2.4 Raspberry Pi 4 Architecture

As mentioned in the introduction, Raspberry Pi 4 SBC was selected as a testing platform for the project, because of its popularity in embedded applications and an affordable price. In order to implement RTIC as efficiently as possible, RPi4 architecture has to be understood.

RPi4 is based on a Broadcom BCM2711 System on a Chip (SoC), which has four Cortex-A72 cores. The Cortex-A architecture makes writing real-time software on it more difficult as explained in Section 2.1.1. Rust support for bare-metal Cortex-A applications is also limited.

2.4.1 ARM Generic Interrupt Controller

Because core concept of RTIC is exploiting hardware interrupt controller for efficient scheduling, it is important to understand if RPi4's controller is suitable for this. Earlier Raspberry Pi models used proprietary Broadcom interrupt controllers, which were limited and very poorly documented. RPi4 also has this proprietary controller, but in addition to it a standard ARM Generic Interrupt Controller (GIC) v2 controller was included.

The Cortex-A architecture has a complicated interrupt handling scheme as shown in Figure 2.4. All device peripheral interrupts are attached to the global GIC distributor. It handles interrupt prioritization and core affinity. When an interrupt arrives, the distributor forwards it to one of the CPU interfaces, which in turn raises the `nIRQ` exception line on Cortex-A72 core to preempt execution. CPU receives only a single signal from its interface, indicating that an interrupt is pending. To learn about interrupt details (i.e. interrupt id), CPU must query GIC CPU interface registers. Unlike in Cortex-M architecture, no context switching is done by the hardware and all registers must be saved/restored by the software. This might not have a large impact on latency, because Cortex-A clock frequency and cycles-per-instruction are much better than that of Cortex-M. However, the exact number depends on many factors: whether the Floating Point Unit (FPU) registers are also saved or if the virtual memory tables need to be swapped. Some of the timings are discussed in the benchmarking Chapter 6.

One of the RTIC requirements for hardware scheduling is nested interrupt support. This can be achieved by telling GIC that interrupt handling has started (but not yet complete) and re-

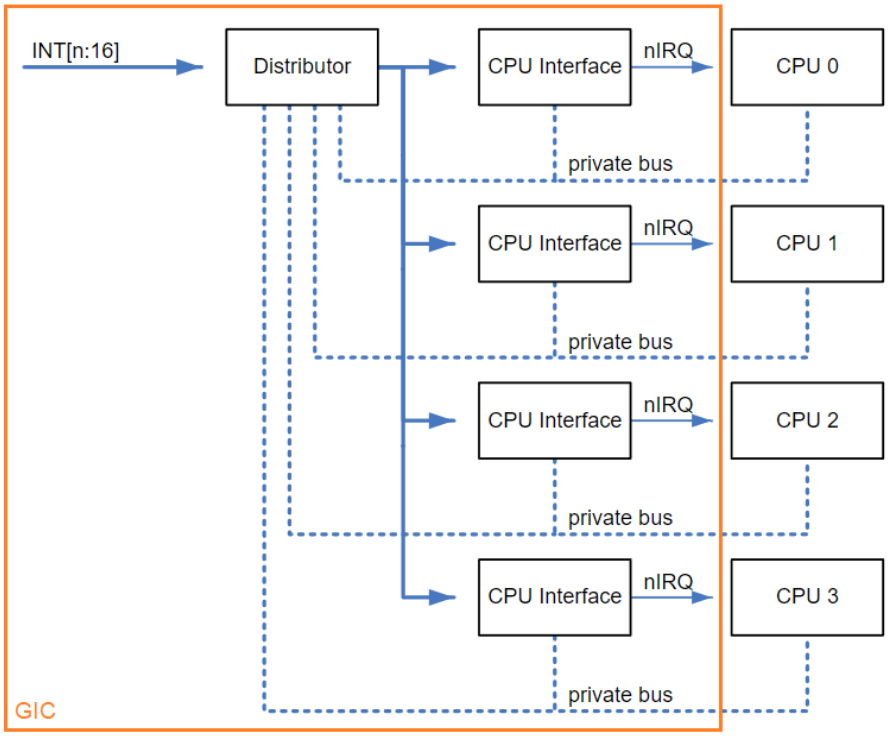

Figure 2.4: ARM Generic Interrupt Controller (adapted from Arm Limited (2007))

enabling exceptions before entering Interrupt Service Routine (ISR). This allows GIC to preempt a lower priority ISR, by issuing a new external interrupt signal. The second RTIC requirement of interrupt prioritization is handled by the GIC. Hence, it should be possible to implement bare-metal RTIC on RPi4.

3 Analysis

RTIC's scheduling was designed to run directly on the hardware interrupt controller, which achieves the best performance possible. However, when the Linux kernel has to run on the same system, hardware resource sharing becomes a problem. Some peripherals can be assigned to either side, but the hardware interrupt controller has to be shared between RTIC and Linux, and neither of them supports such functionality. Alternatively, a different scheduling approach can be used by sacrificing performance and using higher level abstractions instead of relying on hardware interrupts. This leads to a multitude of design decisions and this chapter attempts to analyse them and select the best implementation approach.

3.1 RTIC Requirements

At its core, RTIC's task model is based on prioritized preemption. Each task is assigned a static priority and can be preempted when a higher priority task is scheduled or becomes available. The original Cortex-M implementation achieves this by assigning tasks to interrupt handlers and utilizing the Nested Vector Interrupt Controller (NVIC) to schedule tasks. However, this is not the only way. For example, the Linux kernel scheduler supports preemptive multitasking, which could also be used to run RTIC tasks, albeit with an additional scheduler overhead.

Additional requirements come from the resource management. RTIC uses SRP, which works by masking all lower priority tasks. In the Cortex-M implementation this is done by utilizing the base priority mask register `BASEPRI` (Arm Limited, 2010), which allows masking interrupt priorities by a single register write and is really efficient. On the RPi4 this becomes a bit more complicated, because it does not have a global interrupt mask register. Interrupt masking is handled by the GIC CPU interfaces, which are local to each core. Also, masking is completely different when implementing tasks on top of the Linux scheduler, but this is covered later in Section 5.2.

3.2 Proposed Implementations

There are quite some ideas how RTIC could be implemented on the RPi4 and 5 of them are analysed in the following sections.

3.2.1 Bare-metal

The first and the most obvious way is to run RTIC bare-metal like the Cortex-M implementation. Because we also want to run non real-time software on Linux, one or more cores can be isolated to solely run RTIC while other cores would run the Linux kernel.

Bare-metal implementation has potential for the best performance, because RTIC tasks can run directly on hardware interrupts. Even though RPi4 does not support global interrupt masking, running on a single core is a viable alternative. Firstly, real-time applications usually rely on fast response times rather than heavy processing, which reduces the need for multiple cores. And most importantly, SRP was not designed for multiprocessor scheduling and results in poor schedulability. Although there is a wide range of different multiprocessor scheduling and resource access algorithms, redesigning core RTIC's principles is out of scope of this project.

However, the main issue with the bare-metal implementation is that the interrupt controller must be shared with the Linux kernel. This requires kernel modifications: either exposing kernel's interrupt controller API to the bare-metal RTIC or making the kernel use the bare-metal RTIC's interrupt controller shim. Both approaches are complicated, because in addition to memory sharing, some sort of synchronisation is needed between bare-metal and Linux cores.

Other resource sharing must also be taken care of, but it is less complicated. A fixed-size chunk of memory for the bare-metal process can be allocated from a Linux kernel and any peripherals used by the real-time application could just be disabled in Linux. Communication between the bare-metal application and the Linux could be handled by shared memory buffers.

3.2.2 Real-Time Linux Kernel Interrupts

Originally the Linux kernel was not suitable for real-time applications, because it had long unbounded latencies: many spinlocks and long non-preemptible critical sections. Other real-time features, such as high resolution timers were also missing. The project known as `PREEMPT_RT` was created to resolve these issues (The Linux Foundation, 2015). Initially, it started out as a series of patches, but with time, many of them were merged into the mainline Linux kernel. It is expected that soon the remaining patches will also be merged and Linux will support real-time scheduling out of the box.

With real-time Linux all unbounded latencies are removed from the kernel and it is possible to run RTIC as a Loadable Kernel Module (LKM). This gives direct access to the underlying hardware and the available Linux drivers simplify implementation. The kernel module loading mechanism is also much more convenient to use compared to the bare-metal approach. Such implementation could either run directly on hardware interrupts or use kernel threads, but both options have issues.

LKMs can bind interrupts dynamically at load time so running RTIC applications would be as simple as loading a kernel module. Unfortunately, the Linux kernel does not support hardware interrupt prioritisation. Even though interrupts in Linux have priorities, they refer to the thread priority when interrupts are executed in the threaded mode. In this mode, hardware interrupts only flag kernel threads to be run and immediately return. The actual interrupt handlers are then executed by the Linux scheduler in a thread. Hence, the only way to have hardware interrupt scheduling is to modify the Linux kernel to support interrupt prioritisation. However, kernel modifications are highly undesirable, because they either must have a very important use case to be included in the mainline kernel or else they become a maintenance burden to keep up to date with the latest kernel version.

3.2.3 Real-Time Linux Kernel Threads

Since native prioritized interrupts are the same as kernel threads, there is no need to keep all the interrupt complexity. As an alternative approach, tasks could just be run on kernel threads directly. Task spawning would then be done using kernel synchronisation primitives such as `workqueue` instead of firing interrupts.

3.2.4 Real-Time Linux User Space Threads

An even higher-level alternative to the kernel threads can be user space threads. In this case, task spawning would be done using user space synchronisation primitives, such as `futex`.

It must be noted that user space threads come with an additional performance penalty, because a context switch involves additional work (virtual memory remap, privilege level switch, etc). However, as observed in Chapter 6, this performance penalty is quite insignificant when compared to kernel threads as most of the scheduling overhead is not in the context switch, but in the kernel scheduler code itself. This makes user space threads a viable scheduling alternative.

Most importantly, running RTIC in the user space has many advantages. Firstly, process isolation prevents the application from crashing the entire system, which could happen with the kernel module. Secondly, while kernel module applications are forced to use `no_std` attribute, user space applications can utilise the full Rust standard library and a wide range of crates that depend on it. Developing and debugging user space applications is much easier.

3.2.5 Dovetail Interface

An alternative approach to the real-time Linux is a co-kernel design. Instead of making the Linux kernel fully preemptible, a small real-time microkernel runs between hardware and the Linux kernel. This allows real-time applications to run on the microkernel, unaffected by any blocking in the lower priority Linux kernel.

This approach is used by the two popular projects: RTAI (Mantegazza et al., 2000) and Xenomai (Gerum, 2001). Both of them are currently based on the Adaptive Domain Environment for Operating Systems (ADEOS) virtualisation layer (Yaghmour, 2001), but have different higher level services for real-time applications. However, even though the microkernel offers complete isolation from the Linux kernel, it comes with some drawbacks:

- The microkernel runs directly on the hardware and has a lot of platform-specific code, which needs to be maintained. Because of this, platform support is not as great as with the Linux kernel, which has much larger developer community.
- This approach also requires patching the Linux kernel interrupt handling, which means that the microkernel is usually a few kernel versions behind.
- It is not possible to use Linux device drivers, because of different microkernel API.

To solve at least some of these problems, Xenomai 4 will be shipped with a new Dovetail interface (The Xenomai Developers, 2020) as an alternative to the ADEOS layer. It attempts to be much smaller than ADEOS by reusing the Linux kernel code where possible. This reduces the amount of maintenance required and expands the range of supported platforms.

Because Dovetail taps into the Linux interrupt pipeline and provides a way to bind hardware interrupts outside of the Linux kernel, it would be ideal for implementing RTIC. However, both ADEOS and Dovetail are missing interrupt prioritisation. Hence, similar code modifications as with LKM interrupt approach would be needed.

3.3 Summary

Approach	Bare-Metal Core	LKM Interrupts	LKM Threads	User Space Threads	Dovetail Interface
Latency	5	3	2	1	4
Portability	1	3	4	5	2
Isolation	1	1	1	5	1
Driver availability	1	3	4	5	2
Ease of implementation	1	2	4	5	2
Total Score	9	12	15	21	11

Table 3.1: Comparison of the proposed RTIC implementations on the RPi4

All of the discussed approaches are summarised in Table 3.1. The following criteria have been considered:

- Latency – how good is the interrupt and scheduling latency.
- Portability – how reusable is the code on different chips/platforms.
- Isolation – how isolated is the application code from the rest of the system (crash resilience).
- Driver availability – can existing peripheral drivers be reused or do they have to be reimplemented.
- Ease of implementation – how much work is required to realise the implementation.

All categories are scored from 1 (worst) to 5 (best) and the sum of all scores is shown as a total score. The weight of each category can be very subjective and heavily depends on the target application requirements, but because we are analysing the general robotics case, all categories were treated as equal.

As we can see from the table, latency is at odds with all other categories. The bare-metal core approach offers the best latency, but it loses on all other categories. Since latency is a very important factor for real-time applications, this approach has been selected to investigate how good the raw interrupt latency could be, without actually implementing RTIC. This is useful for comparing overhead of other implementations. Bare-metal investigation is described in Chapter 4.

The user space thread approach was selected for implementing RTIC. Even though it has possibly the worst latency of all, the absolute values are hard to predict beforehand. Moreover, latency requirements depend on the application and if they are not too strict this approach makes up for it in all other categories. Writing real-time applications on user space RTIC should be easy, because of highly portable code, wide availability of drivers and an excellent process isolation. The process of implementing RTIC on user space threads is described in Chapter 5.

4 Investigating Bare-Metal Approach

This chapter analyses bare-metal RTIC compatibility with Linux, when running on isolated cores. The first section covers Linux compatibility and the second section discusses Rust bare-metal development on Cortex-A platform.

4.1 Bare-Metal and Linux

4.1.1 Booting Bare-Metal Application

Isolating cores in the Linux kernel is quite simple. The kernel supports command-line parameters, which are passed by the bootloader. For example, on the RPi4 kernel parameters can be specified in the `cmdline.txt` file on the boot SD card partition. The two relevant parameters are `isolcpus` and `maxcpus`. `isolcpus` prevents any tasks from running on the specified cores, but the Linux scheduler is still enabled and tasks can be manually migrated to it. On the other hand, `maxcpus` completely disables the cores and the scheduler is not even run. The exact mechanic how cores are suspended depends on the SoC.

On the RPi4, the bootloader launches the Linux kernel only on the first core. The remaining 3 cores are put into a sleep loop, waiting for a jump address in a hardware mailbox. The mailbox is just an 8 byte location in the memory, separate for each core. When the Linux kernel boots, secondary cores are woken up by writing kernel entry address into each of the mailboxes and all secondary cores jump to the kernel code. However, when `maxcpus` option is specified, unused cores are left in their sleep loops. This allows executing bare-metal code on the isolated core once Linux is booted. The summarised procedure looks like this:

- Bootloader starts the Linux kernel with `maxcpus=3`, because the RPi4 has 4 cores.
- Linux boots on cores 1-3 and the 4th core is left sleeping on the mailbox.
- Linux allocates a memory region for the bare-metal application and loads the application binary there.
- The application entry address (in the allocated memory) is written into the 4th core mailbox and it begins executing the bare-metal application.

This technique can be abstracted with the Linux `remoteproc` framework (Linux Kernel Developers, 2012), which provides a convenient API for executing applications on remote processors. It is designed for Asymmetric Multiprocessing (AMP) systems, which usually contain heterogeneous (different architecture) cores, but can also be used for isolated homogeneous cores. Xilinx, Inc. (2013) demonstrated the `remoteproc` framework by simultaneously running Linux and `FreeRTOS` on two Cortex-A9 cores. However, the `remoteproc` framework requires different drivers for each SoC and one for the RPi4's `BCM2711` SoC does exist yet.

An alternative approach would be to boot the bare-metal application first and then launch the Linux kernel on the remaining cores. However, booting Linux first is much more convenient:

- Memory can be allocated dynamically instead of setting up static memory regions in both the Linux kernel and the bare-metal application.
- The system starts with a functioning OS, which makes developing and updating bare-metal applications easier than flashing an SD card.

4.1.2 Sharing Resources

When running Linux and bare-metal applications on the same machine, system resources must be shared. As previously mentioned, memory sharing is as simple as allocating a static memory region in the Linux kernel and passing it to the bare-metal application. This memory is exclus-

ive and the bare-metal application can have its own allocator without calling into the Linux kernel.

The `remoteproc` framework has also basic resource management support. Applications loadable by `remoteproc` must be in an Executable and Linkable Format (ELF), which allows additional metadata to be attached. The resource table section in the ELF is used to define any resources used by the application. For example, `remoteproc` can automatically allocate a memory region for the application if it is defined in the table. This allows easily loading `remoteproc` applications without any additional setup from the Linux side.

One thing that is unfortunately missing from the `remoteproc` framework is device tree support. Device trees contain definitions about all the peripherals in a particular system: memory addresses, assigned drivers, configurations, interrupt mapping and more. If the `remoteproc` framework would support them, then the kernel could automatically disable peripherals locally and pass them for the remote application to use. However, this is not yet possible. The workaround is to manually disable specific peripherals in the kernel's device tree and then just define raw memory maps in the `remoteproc` application's resource table.

A different kind of issue are peripherals that have to be shared between the Linux kernel and the bare-metal application. In RTIC's case, it's the hardware interrupt controller, which is needed by both the Linux kernel and the RTIC scheduler to function. The exact mechanism how sharing could be implemented heavily depends on the hardware specifics. As described in Section 2.4.1, RPi4's GIC interrupt controller has two parts: the distributor and CPU interfaces. The latter are individual to each core and sharing is not a problem. However, the distributor is shared by all cores and its register state must be coordinated. Implementing such coordination would involve:

- Implementing an interrupt control protocol between the bare-metal application and the Linux kernel. This could utilise the `RPMsg` transport from the `remoteproc` framework for exchanging messages.
- Adding support for interrupt prioritisation in the Linux GIC driver.

Because such approach is very heavy in the implementation work and contains a lot of platform-specific code, it was deemed out of scope of this project. An alternative way of implementing RTIC on the Linux user space threads was chosen and is described in Chapter 5.

4.2 Bare-Metal Development

Instead of implementing RTIC on bare-metal, the research knowledge was used to implement a standalone bare-metal application to measure RPi4 GPIO interrupt latency. The measurements are used in Section 6.1 to compare how much overhead the Linux kernel adds.

The measurement application is very simple: it waits for a rising edge interrupt on the input GPIO pin and responds by sending an impulse on the output GPIO pin. The application could have been quickly hacked together in C language by just manipulating the registers, but effort was put to improve the Rust's Cortex-A and RPi4 ecosystem by writing HAL libraries. The intention was that this might be useful for both future standalone bare-metal applications and the future bare-metal RTIC implementation.

Bare-metal development on the RPi4 is quite some work and details of it are given in the Appendix B. A summary of what was done (except the `cortex-a` crate) is given as a dependency graph in Figure 4.1.

The following crates (libraries) were developed:

- `rpi-pac` – contains peripheral register definitions and memory addresses.

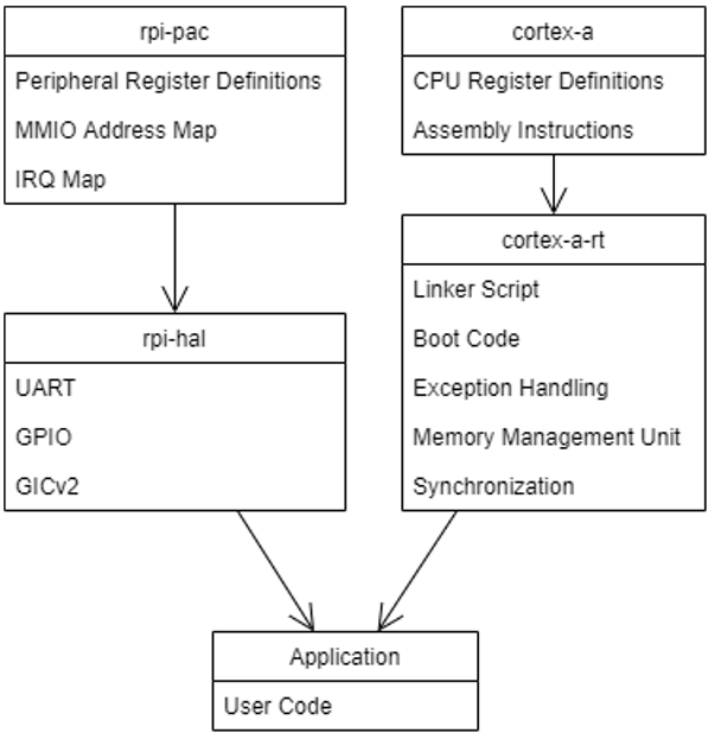

Figure 4.1: Bare-Metal Rust Application Dependency Graph for RPi4

- `rpi-hal` – a Hardware Abstraction Layer (HAL), built on top of `rpi-pac` definitions. For now, it implements GPIO, Universal Asynchronous Receiver-Transmitter (UART) and GIC peripherals.
- `cortex-a-rt` – contains startup code, needed to reach Rust's `main()` function: linker scripts, assembly entry code, Memory Management Unit (MMU) initialization, exceptions and privilege levels. Depends on an already existing crate `cortex-a`, which provides Cortex-A specific CPU register definitions.

Additionally, these utility crates were also developed:

- `cortex-a-quickstart` – an application template containing all of the dependencies needed to build a bare-metal application.
- `rpi-bootloader` – a bootloader that loads applications over UART using XMODEM protocol. A quicker alternative to SD card flashing.

With these crates it is now possible to write high-level Rust applications for the bare-metal RPi4. One such application was used to measure GPIO interrupt latency and the results of it are given in Section 6.1.

4.3 Summary

Implementing RTIC on an isolated Linux core is possible and the existing `remoteproc` framework makes loading applications convenient. However, resource sharing is a problem that still needs to be resolved. Exclusive resources can be allocated by some manual configuration, but shared ones (i.e. interrupt controller) require some kind of communication between the Linux kernel and the bare-metal application. Implementing it was skipped in favour of the user space thread approach described in Chapter 5.

Writing bare-metal applications for Cortex-A is considerably more complicated than for Cortex-M, because of all the additional hardware features: the MMU, privilege levels, multi-

core interrupt controller. Nevertheless, the groundwork for the bare-metal RPi4 development was laid by implementing essential `rpi-pac`, `rpi-hal` and `cortex-a-rt` crates. This foundation was used to build GPIO interrupt latency measurement application.

5 RTIC on Linux User Space Threads

This chapter describes the implementation of RTIC on user space Linux threads. Two preemption strategies are discussed in Section 5.1. Their compatibility with the RTIC's resource model is analysed in Section 5.2 and compatibility with the hardware peripherals is discussed in Section 5.3. Both approaches are compared in Section 5.4 and the final implementation is described in Section 5.5.

In order to use the Linux scheduler for RTIC tasks, the kernel must be built with real-time support, which is not default for the RPi4. The process on how to apply the `PREEMPT_RT` patch and build the kernel is described in Appendix C. All of the further development and testing was done on the kernel version `5.10.52-rt46-v8+`.

5.1 Task Model

RTIC is based on a prioritized preemption task model as described in Section 2.2.1. It has already been covered in the analysis (Chapter 3) that the Linux scheduler and its user space threads could be used for executing such model. However, threads are not the only way to achieve preemption in user space. The IEEE (2016) Portable Operating System Interface (POSIX) standard defines an additional preemption method, which is supported by the Linux kernel – asynchronous signals. This method is used by the mentioned `linux-rtfm` implementation (Aparicio, 2019).

5.1.1 POSIX Signals

POSIX signals are handled by the Linux kernel by interrupting (preempting) the execution of a program and jumping to an associated signal handler in the program. For example, the well known `Ctrl+C` key combination in the Linux terminal sends a `SIGINT` signal to the program, which can be handled to perform a graceful shutdown. In addition to the handful of standard signals, there are also user-defined signals `SIGUSRx` and a set of real-time signals, which can be freely used by the application. Real-time signals have the additional property of being ordered by their number, simulating priority levels. Signals can be triggered by the application itself with the `kill()` function, and can also be time triggered with a `timer_create()` function.

POSIX-signal-based RTIC implementation would work by replacing hardware interrupts in the original Cortex-M implementation by signal handlers. The overall behaviour would be almost identical: signal handlers have priorities just as hardware interrupts and spawning a task is sending a signal instead of pending an interrupt.

However, POSIX signal handlers have a very large limitation. Because signals can preempt process at any time (for example inside a synchronisation lock), signal handlers are limited to reentrant functions only. Hence, the POSIX standard defines a small subset of functions that are safe to use inside signal handlers. Unfortunately, this list forbids any sort of memory allocation/deallocation and synchronisation primitives. This problem is especially important for Rust applications, because they are guaranteed to be memory safe and signal handlers can violate memory safety if incorrect functions are used from inside a signal handler. Because there is no mechanism in Rust to disable parts of the standard library, the only way to preserve memory safety guarantees is to make RTIC application `no_std` type, which prevents the use of the entire standard library. A small subset of syscalls, which are safe to use inside signal handlers could be then reimplemented as part of RTIC API.

5.1.2 Threads

Thread-based RTIC implementation would work by spawning a thread for each task priority group, similar to how a single interrupt is assigned to handle a group of tasks in the Cortex-M implementation. However, because thread spawning is expensive, threads would run continuously and wait for work (spawned tasks) on a synchronisation primitive. Prioritisation would be achieved by setting threads to a real-time Linux scheduling policy `SCHED_FIFO`, which supports thread prioritisation.

Currently, the most common way of thread synchronisation in the user space Linux applications is a fast user space mutex, also known as futex (Gleixner and Kerrisk, 2015). It is "fast", because in an uncontended case the overhead is just one atomic compare-and-swap (CAS) instruction. Syscall is only invoked when another thread is blocking the futex and the current thread has to be put in a waiting list. It is used for the Rust's standard library mutex implementation and most of the queue libraries out there. Such futex-based FIFO queue is selected for scheduling software tasks: the executing thread is blocked on a queue pop operation and any tasks are spawned by pushing them to this queue, which wakes the executing thread.

Implementing timed scheduling in threads is a bit more complicated, because threads can only block on a single synchronisation operation at a time. For example, a thread cannot execute `FUTEX_WAIT` and `nanosleep()` syscalls at the same time. A workaround would be to spawn an additional timer thread, which would wait for a deadline and then would spawn a task by pushing to the spawn queue. This, however, is not an efficient way of doing it, because it involves multiple syscalls and context switches, greatly increasing latency.

An alternative solution was devised: utilising the futex timeout parameter to wait for both immediate tasks and a timer. The `FUTEX_WAIT_BITSET` operation is used for this, because it accepts absolute timestamps, which are suitable for real-time applications. A more in-depth explanation of this is given in Section 5.5.1.

5.2 Resource Model

The resource policy is another part of RTIC, which is tightly coupled to the task model implementation. The original RTIC uses SRP as described in Section 2.2.2. SRP's algorithm requires masking all tasks under a given priority, which might be hard to achieve depending on the task scheduling approach.

5.2.1 POSIX Signals

POSIX signals support masking with the `sigprocmask()` syscall, which is almost equivalent to the Cortex-M base priority mask register, used for SRP. `sigprocmask()` accepts a bit mask, which can be used to mask all signal handlers under a certain priority in a single syscall. However, the overhead of a syscall is multiple orders of magnitude larger than a single write instruction to the mask register in the Cortex-M case. This makes this approach really slow – syscalls have to be unconditionally invoked on each resource lock/unlock.

No other resource protection alternatives were found due to the signal handler API restrictions. Not even the futex based synchronisation, which is discussed in the next section, is signal-safe.

5.2.2 Threads

Implementing SRP in the thread-based approach is impossible without kernel modifications, because Linux does not natively support SRP, nor has any standard way of suspending threads from outside the thread itself. It is tempting to just rely on increasing the current thread priority, however, this method is not suitable for resource protection as there are absolutely no guarantees that the lower priority threads will not run: in an Symmetric Multiprocessing (SMP) system the lower priority threads would still be executed on different cores and even when restricted

to a single core, the syscalls in the higher priority thread could sleep, switching execution to the lower priority threads.

There are workarounds and threads could be theoretically suspended by using the same POSIX signals and sleeping in a signal handler until resumed. However, this approach is not viable as it would involve one syscall for each running thread, making it perform even worse than POSIX signal masking.

Even though implementing SRP is not feasible, alternative resource policies might be easier to implement and could have comparable guarantees. The main features of SRP are: bounded priority inversion, deadlock prevention and efficient stack usage (Buttazzo, 2011). The first two are essential to RTIC, but the efficient stack usage is not, because modern application processors usually have plenty of memory.

POSIX mutexes support the PCP policy with the `PTHREAD_PRIO_PROTECT` option and were considered as a definite replacement for the SRP. However, it turned out that they do not follow the original PCP definition (Sha et al., 1990). The original PCP protocol calls for an access test on all semaphores in the system (or group), but in the POSIX implementation this is not done and mutexes have no relation to each other, failing to prevent deadlocks. Moreover, the Linux kernel does not have native PCP support and the POSIX implementation is forced to change thread priority on each lock/unlock resulting in many redundant syscalls.

Another real-time resource protection API is the Linux PI futex. It is much more efficient than a POSIX mutex, because thread priority is only changed on priority inversion condition: high priority thread tries to lock a futex held by a lower priority thread. This makes PI futex zero-syscall in the best case and 1-2 syscalls in the worst case, depending if contention happens on a single lock/unlock or both. However, since it follows the original Priority Inheritance (PI) protocol, it does not prevent deadlocks either.

Giving up on deadlock prevention did not seem like an option as it is one of the core features of RTIC. Instead, a new approach was devised – implementing PCP policy on the Linux PI futex API. It uses PI futex to only raise priority when needed, but performs an additional PCP access test to prevent deadlocks. The in depth explanation of the algorithm is given in Section 5.5.2.

Policy	POSIX PCP mutex	PI futex	PCP on PI futex
Bounded Priority Inversion	+	+	+
Deadlock Prevention	-	-	+
Syscalls (best)	2	0	0
Syscalls (worst)	3-4	1-2	1-2

Table 5.1: Comparison of different resource access algorithms for Linux threads

Table 5.1 shows three different resource protection alternatives for Linux threads. They were compared in terms of 4 properties:

- Bounded Priority Inversion – specifies if the algorithm resolves unbounded blocking due to priority inversion.
- Deadlock Prevention – whether the algorithm guarantees that the program never deadlocks.
- Syscalls (best) – number of syscalls required for lock-unlock in the best case (contention free).
- Syscalls (worst) – number of syscalls required for lock-unlock in the worst case (contended mutex).

In summary, none of the available real-time resource protection APIs for threads are good enough to replace SRP in RTIC. A newly designed algorithm was the closest match in terms of features.

5.3 Peripheral Interrupts

In addition to software tasks, the hardware peripheral handling and their interrupts must be considered. The Linux kernel does not provide a way to bind hardware interrupts from the user space and they are handled by the peripheral drivers in the kernel. Peripheral events are usually exposed through the file descriptors and blocking APIs such as `read()` or `select()`. This allows the user space thread to sleep without constantly polling peripherals for new data.

In a POSIX signal approach this is problematic, because the entire application runs on a single thread and waiting for peripheral events would block all lower priority tasks (signals). Hence, external threads would have to be used for peripheral handling, which would immediately introduce all the problems related to the thread approach. Without the threads, POSIX signal approach is limited to the polled peripherals only.

On the other hand, the thread-based approach integrates seamlessly with the blocking peripheral API. The only required change is to exclude hardware tasks from the software task priority groups and run them in individual threads. Otherwise, sleeping hardware task would block the pending software tasks of the same priority.

5.4 Selecting Approach

Approach	Threads	POSIX signals
Task Spawning	+	+
Task Scheduling (timed)	+ (additional work required)	+
Resource Policy	PCP (additional work required)	SRP
OS API	full std library	reentrant functions only
Peripheral API	blocking + polled	polled only

Table 5.2: Comparison of thread and POSIX signal RTIC implementation approaches

A summary of both approaches is given in Table 5.2. The thread based approach requires more implementation work to get around the Linux scheduler API limitations. However, even though the POSIX-signal approach is easier to implement, the API limitations in signal handlers were ultimately deemed to be too restricting for the end-user RTIC applications. The thread-based implementation was selected and it is described in the next section.

5.5 Implementation Based on Threads

Even though the separate `rtic-syntax` crate makes implementing additional RTIC platforms easier, there is still a lot of code in the `cortex-m-rtic` crate, which is not entirely platform-specific. Hence, the original `cortex-m-rtic` code was taken as a base and the Cortex-M platform-specific code was replaced with Linux variants.

As discussed in previous sections, two libraries were developed and published as independent crates to aid the implementation:

- `futex-queue`[1]: a FIFO queue that implements efficient sending of both immediate and scheduled (timed) values. It's implementation is described in Section 5.5.1.
- `pcp-mutex`[2]: a PCP resource policy emulation on Linux PI futex. It's implementation is described in Section 5.5.2.

[1] https://crates.io/crates/futex-queue
[2] https://crates.io/crates/pcp-mutex

5.5.1 Efficient Task Scheduling on Threads

The `futex-queue` crate was developed to implement efficient scheduling of RTIC tasks. It is a Multiple Producer, Single Consumer (MPSC) queue with timer support. The sender API has two methods: `send(item)` and `send_scheduled(item, timestamp)`. Items sent with the latter are only made available to the receiver when the given timestamp expires. On the receiver side, there is only one `recv()` method which pops a single item from the queue or blocks the current thread until an item becomes available.

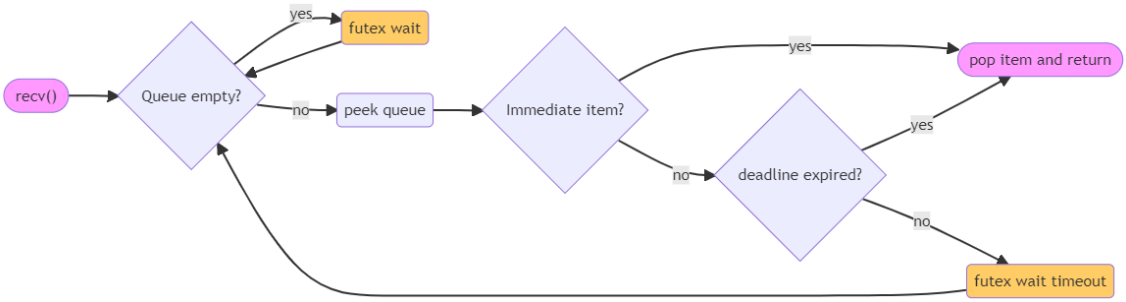

Figure 5.1: Flowchart of the `futex-queue` `recv()` function

The entire working principle can be summarised from the `recv()` function's flowchart, which is shown in Figure 5.1. Internally the `futex-queue` has a binary heap based priority queue, which prioritises immediate items first, then scheduled items by an increasing timestamp value. This forms a logic where the `recv()` function takes the first available action in the following order:

1. Return an immediate item
2. Return a scheduled item with an expired deadline
3. Wait on futex with a timeout of the earliest scheduled item
4. Wait on futex without timeout if the queue is empty

Both futex operations (marked yellow) use the same futex instance and are woken up when a new item is added with one of the `send` calls. One drawback of such implementation is that the receiver thread is woken up for each `send_scheduled()` call, regardless if there are ready items (expired deadlines). This wake up is needed to update the futex timeout value, in case the newly added item has an earlier deadline.

In the RTIC implementation there is one executing thread and an associated `futex-queue` for each priority group of software tasks. Tasks are then spawned by sending task id (Rust enum variant) into the queue. The executing thread receives this task id and executes the appropriate task function. This enables scheduling of both immediate and timed tasks with minimal amount of syscalls.

5.5.2 Original Priority Ceiling Protocol based on PI futex

The second developed library is the `pcp-mutex`, which implements PCP resource policy on top of the Linux PI futex API. It was developed, because none of the available Linux synchronisation APIs implemented proper SRP or PCP policies to prevent deadlocks.

The Linux PI futex was selected as a backend for the implementation, because it uses less syscalls than the alternatives. For example, the POSIX PCP mutex (even though it is not technically PCP as explained in Section 5.2.2) always uses 2 syscalls to raise/restore thread priority and 2 more optional syscalls if mutex is contended. This results in 2 syscalls in the best case and 3-4

syscalls in the worst case. Contrary to this, the PI futex only increases priority when the futex is contended and it is done in a single syscall. In other words, when a thread encounters a locked PI futex, it calls the `FUTEX_PI_WAIT` syscall, which tells the kernel to sleep the thread until the futex is unlocked and to increase the priority of the thread that is currently holding the lock.

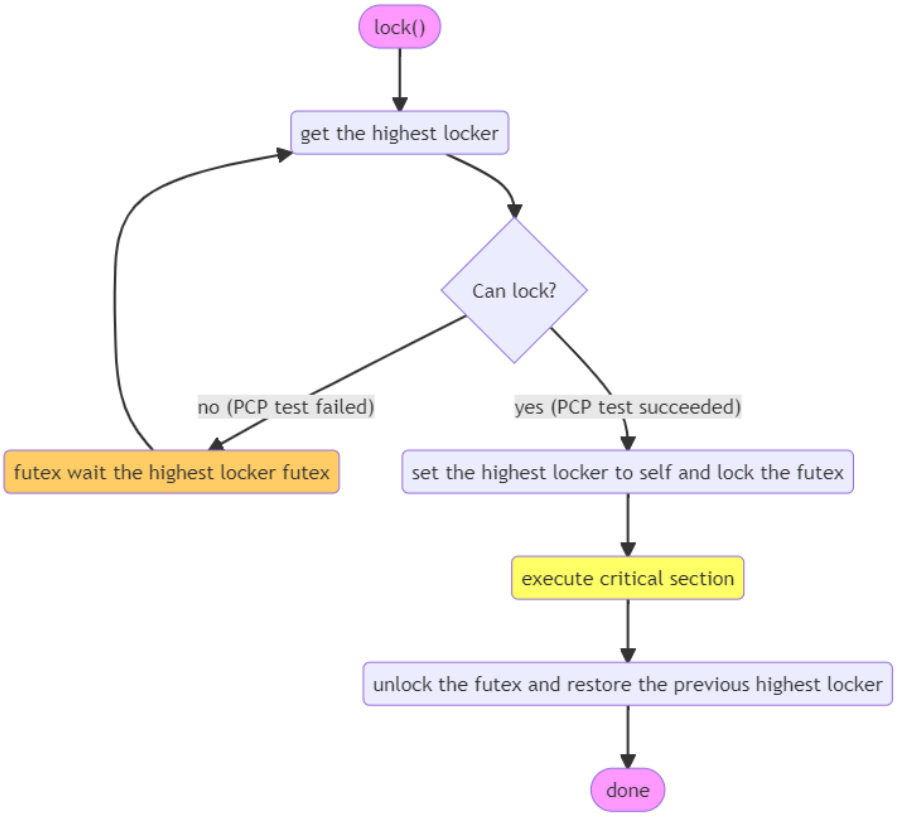

Figure 5.2: Flowchart of the `pcp-mutex lock()` function

The PI futex enables efficient handling of the priority inversion problem, but it is still lacking the deadlock prevention. To solve this problem, an additional PCP test is added before attempting to lock the futex. The simplified locking mechanism is shown as a flowchart in Figure 5.2. The PCP test checks if the current thread priority is above the highest locker ceiling (or if the current thread is the highest locker) and if it's not, the current thread is put to wait until the highest locker is unblocked. The highest locker is a single global state, which forms a stack-like data structure of increasing priority ceilings. Each resource has its own PI futex, but if the thread gets blocked by a priority ceiling, it waits on a futex of the other resource, which increased the ceiling.

The given flowchart shows the high-level behavior of the `pcp-mutex`, however, the actual implementation is quite a bit more complicated. Because it is shared by multiple threads, it must be free of race conditions. This is ensured by using an atomic pointer for the highest locker state and atomic compare-and-swap (CAS) instructions to ensure consistency. However, properly implementing atomic operations is hard – modern processors employ a variety of optimisations that usually result in out-of-order execution. This goes unnoticed in single threaded programs, because data dependencies are still respected, but can cause unpredictable behavior in concurrent programs. To protect critical areas, memory barriers must be carefully placed around atomic instructions. This caused problems in the initial implementation, but strategically crafted test cases and multiple code reviews allowed to track down the issues.

5.6 Summary

With the two `futex-queue` and `pcp-mutex` libraries at disposal, implementing the rest of the RTIC on Linux was fairly straightforward: spawning threads, hooking up run queues and wrapping resources in PCP futexes.

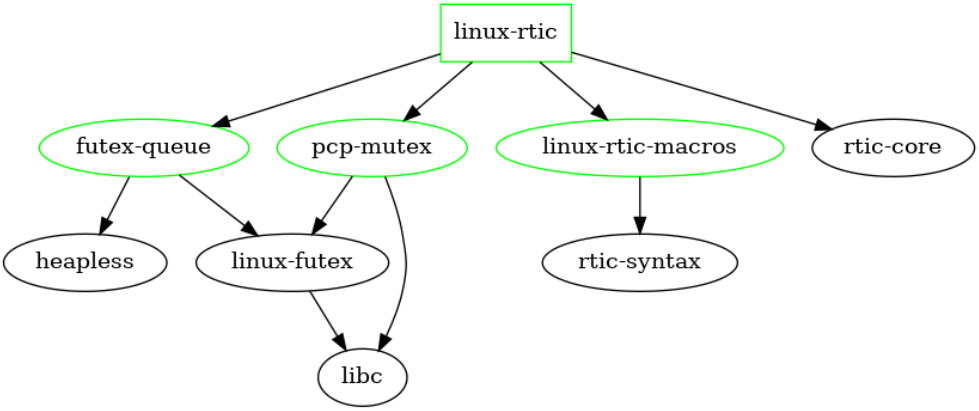

Figure 5.3: Dependency graph of `linux-rtic`

The final dependency graph of the Linux thread based RTIC implementation is shown in Figure 5.3. Some of the irrelevant leaf dependencies are not shown (rust syntax parsing, tracing libraries, etc.) for simplicity. The dependencies marked in green were developed as part of this project. The resulting `linux-rtic` crate can be found on the Rust crate registry[3]. It also includes 11 example applications demonstrating the usage of the API.

[3] https://crates.io/crates/linux-rtic

6 Benchmarks

This chapter is about performance benchmarks on the RPi4. Section 6.1 does not include RTIC and focuses on comparing raw interrupt latency between the bare-metal application and the Linux kernel. Section 6.2 is about analysing the performance of the developed RTIC implementation on user space Linux threads and discusses what could be done to improve it.

6.1 GPIO Interrupt Latency and Jitter

This test was devised to measure the Linux kernel overhead compared to the bare-metal application. It was decided to use the GPIO peripheral interrupts for measuring latency, because its timings are easy to measure and observe with an oscilloscope. The general idea is to send an impulse to the input GPIO pin on the RPi4, which is attached to a level-triggered interrupt. The software on the RPi4 reacts to this interrupt and sends a response impulse on the output GPIO pin. The time difference between leading edges of the two impulses is interpreted as the interrupt latency. In addition to the interrupt latency itself, the measurement also includes the time it takes to write the output GPIO pin, but it is considered negligible.

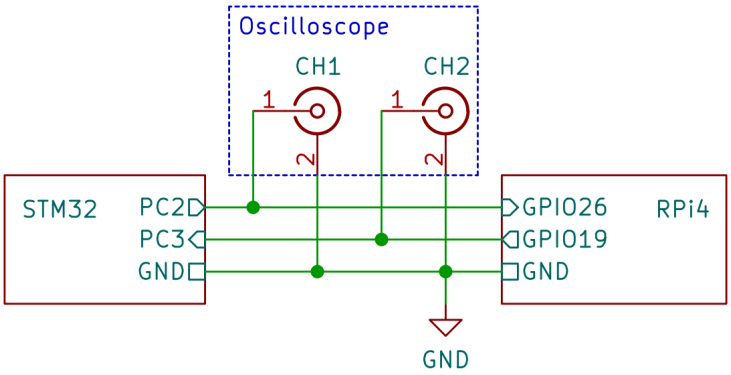

Figure 6.1: GPIO latency measurement schematic

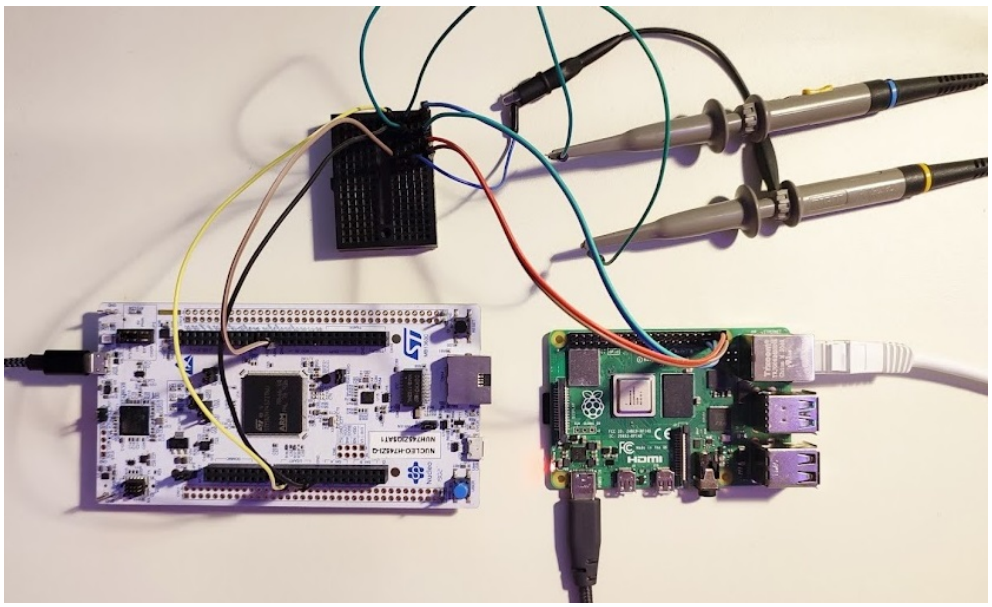

Figure 6.2: GPIO latency measurement hardware setup

The schematic of the test is shown in Figure 6.1 and the hardware setup is shown in Figure 6.2. In addition to the oscilloscope connections, the test also includes a `NUCELO-H745ZI` board. It is used not only to send out impulses, but also to record many measurements and build a histogram. This is needed, because there are many hardware and software factors that affect the latency and some jitter will inevitably be present. A large number of measurements allows to approximate the worst case latency. In the following tests 5 million samples were used as good trade-off between testing time and statistical accuracy. If strict timing guarantees are required, such tests are usually run for days or even weeks.

RPi4 Software Setup

The RPi4 testing software includes two different setups: bare-metal and Linux (multiple configurations).

The bare-metal software on the RPi4 is based on the developments of Chapter 4 and is written in Rust. It contains the bare-minimum code required to handle GPIO interrupts and the interrupt handler is less than a 100 instructions. This should give a close to the minimal latency that is possible by the hardware.

The Linux software is run on the real-time patched kernel in a few different configurations:

- Kernel module: implemented as a Loadable Kernel Module (LKM) and uses the Linux `request_irq()` function to bind the interrupt handler. The GPIO peripheral is handled over the standard Linux GPIO API.
- Kernel module (no thread): same as the kernel module, but the `request_irq()` function is called with `IRQF_NO_THREAD` flag, which tells the kernel to handle interrupt in the hardware interrupt context instead of an interrupt thread.
- User space: based on the Linux user space `gpiod` API. Interrupt is handled by waiting on a `gpiod` event file descriptor.

It is also important to ensure that dynamic frequency scaling (CPU governor) is disabled in the Linux kernel, otherwise the downscaling can negatively affect latency measurements. This was done using the `cpufreq-set` utility by setting CPU governor to the performance mode.

Results

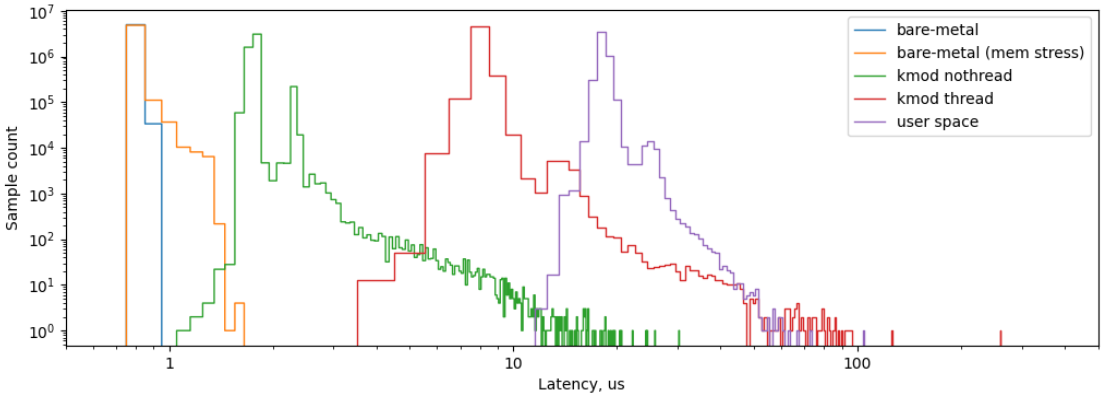

Figure 6.3: GPIO latency histogram in logarithmic scale (no system load)

The results of the latency measurements are shown in Figure 6.3. The bare-metal has the lowest latency averaging at about 0.8 µs. Considering that RPi4 cores run at 1.5 GHz, this represents about 1200 instructions, which seems a lot. It is probable that most of this time is spent in the internal Cortex-A exception handling circuitry or in the GPIO or GIC peripherals. This timing

should not be affected by the cache misses, because the bare-metal application sleeps in the main loop and only processes GPIO interrupts.

To make the measurements fair, a cache grind code was added to the bare-metal application. The code allocates a 4 MB size array in the memory so that it does not fit in the 1 MB L2 cache of the Cortex-A72. Then it proceeds to read and write random values across the array. This ensures that the interrupt handler sometimes gets evicted from the caches. And as we can see from the results, this indeed increases bare-metal interrupt latency from a worst case of 0.9 µs to almost 2 µs. This might not seem much, but if the interrupt handling code had many functions spread out in the memory, each interrupt could cause multiple cache misses and further increase the latency.

The Linux kernel has a fairly decent average interrupt latency in the no-thread mode of about 2 µs, which is comparable to that of the bare-metal. However, the worst case latency gets pretty bad – up to 30 µs. This is likely due to the critical sections in the Linux kernel, which disable interrupts. The `PREEMPT_RT` patch team worked hard to remove most of such sections, but they could still be present in the RPi4 drivers, which are not reviewed by the Linux kernel maintainers and are published in a Raspberry Pi's fork of the Linux kernel.

Finally, the two thread-based configurations perform the worst, as expected. The kernel thread approach has an average latency of about 9 µs, while the user space approach has an average latency of about 20 µs. What is interesting from these two measurements, is that the user space context switch is not that more expensive than the kernel thread context switch. The average user space latency is twice that of the kernel, but that is because two context switches are performed instead of one. In the kernel thread test the context switch only happens between the interrupt handler and the interrupt thread, while in the user space test the context switch happens from the interrupt handler to the peripheral driver thread and only then to the user space thread. This shows that a single context switch time is roughly the same in the kernel space and in the user space, which is surprising.

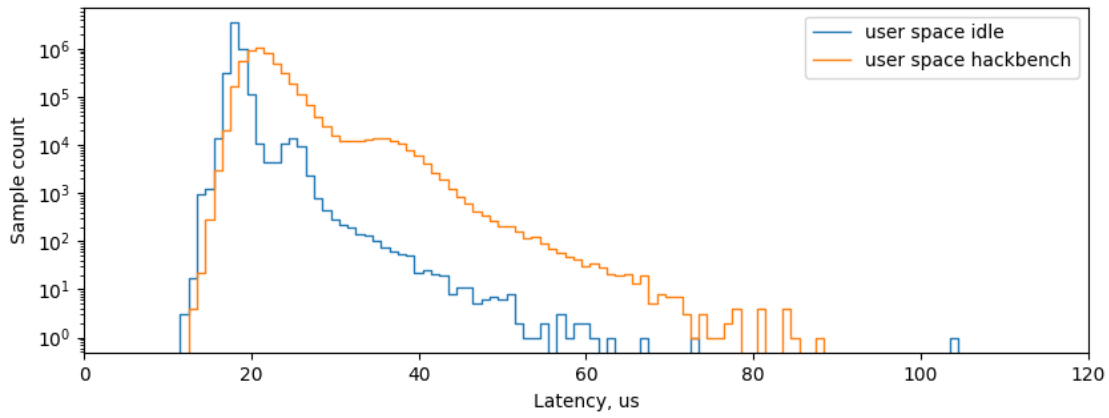

Figure 6.4: User space GPIO latency histogram with different system load

Another notable benchmark is shown in Figure 6.4, which demonstrates the user space latency difference between an idle and a loaded system. The system was loaded with a `hackbench` tool from the Real-Time Linux Test Suite (Gleixner et al., 2007), which is specifically designed to stress test the Linux scheduler by spawning many threads and exchanging messages between them. It can be seen that on the loaded system latency is slightly increased but not by much. There is, however, a strange one sample fluke above 100 µs in the idle case, but it is unlikely to be related to the system being idle. It is probably a long lock in a rare kernel code path, which would be triggered more often if tested for longer periods of time.

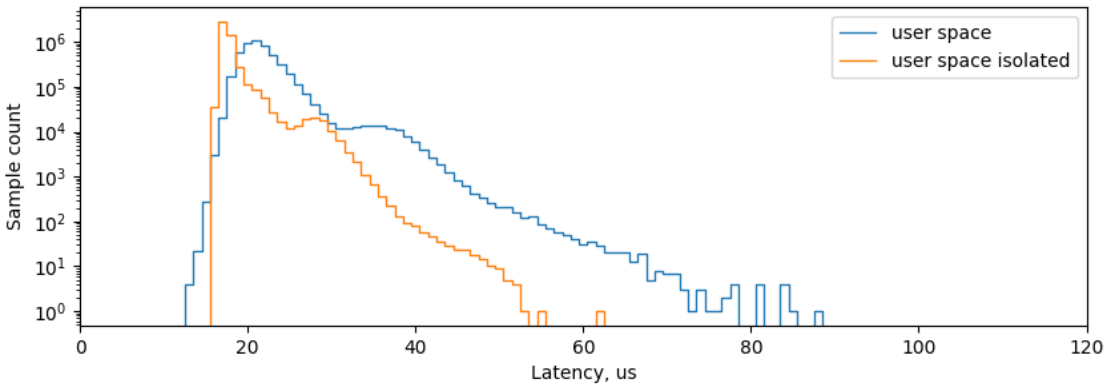

Figure 6.5: User space GPIO latency histogram with isolated CPU

The final GPIO latency test was done to see how isolating cores would affect the user space latency. Kernel was booted using `isolcpus=3` parameter to isolate the 4th core and the test application was launched with a `taskset -c 3 <cmd>` command to assign it to this core. System was loaded with the `hackbench` tool as in the previous test. As we can see from the graph in Figure 6.5, latency was reduced and is similar to the idle case of the previous test. It is, however, interesting that the minimum latency became slightly worse. While it is hard to speculate, it could be that some active code paths in the Linux kernel lead to a faster interrupt handling than from a sleeping core.

6.2 Linux RTIC Performance

This section is dedicated to benchmarking the RTIC implementation based on user space Linux threads, which was developed in Chapter 5.

6.2.1 Task Switching Performance

The first test was devised to measure the task switching performance and the efficiency of the underlying `futex-queue` crate, which is at the basis of scheduling. The test was written as an RTIC application to measure the real-world performance.

The test application is shown in Listing 1. It contains two tasks that are exchanging variable `x` in a ping-pong manner, increasing it by 1 each time. This is done 10 million times, after which the measured execution time is printed. Experimental results show that on the RPi4 it executes in 3.94 s, which equals to 394 ns per task switch or about 600 instructions at 1.5 GHz.

However, the benchmark was done using equal priority for both tasks, which means that they are executed on the same thread due to the RTIC's grouping of software tasks. When two different priorities are used, each task is executed on a different priority thread and a kernel context switch is required. When measured, this case resulted in the execution time of 98.9 s, which equals to 9.89 µs per task switch. This is about 25 times slower than the single threaded case. However, an interesting thing happens when the test application is limited to a single core – execution time drops to 64.8 s or 6.48 µs per task switch. This can be explained if we delve deeper into the `futex-queue`-based scheduling mechanism. In the multiple core case, each executor thread alternates between sleeping and running states, because after each execution its run queue is emptied and it has to wait on a futex. In the single core case, though, the lower priority thread never goes to a sleeping state, because as soon as it notifies the higher priority thread of a new task, it gets preempted. It returns from the preemption when the other task has finished executing, but at this point its run queue already has a task queued and the process repeats. This is, however, not universal and with a larger set of tasks multiple cores would out-

```rust
#[rtic::app]
mod app {
    #[local]
    struct Local {
        start: Instant,
    }

    #[init]
    fn init(_: init::Context) -> (Shared, Local, init::Monotonics) {
        task1::spawn(0).unwrap();
        (Shared {}, Local { start: Instant::now() }, init::Monotonics())
    }

    #[task(priority = 1)]
    fn task1(_cx: task1::Context, x: i32) {
        task2::spawn(x + 1).unwrap();
    }

    #[task(priority = 1, local = [start])]
    fn task2(cx: task2::Context, x: i32) {
        if x < 10_000_000 {
            task1::spawn(x + 1).unwrap();
        } else {
            println!("Time: {:?}", cx.local.start.elapsed());
        }
    }
}
```

Listing 1: Task Switching Benchmark Code

perform a single core. Nevertheless, it shows how unpredictable timings in operating systems can be and that everything requires testing.

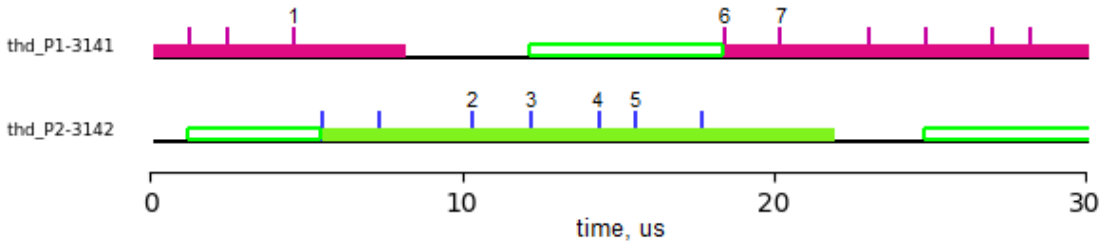

Figure 6.6: Kernelshark analysis of the RTIC task switching

To understand the cause of such a large thread scheduling overhead, the Linux kernel tracing facilities were used. Tracing of system calls and scheduler events was done using the `trace-cmd` command, which is an interface for the Linux `ftrace` subsystem. The captured traces were then opened with the `kernelshark` tool (Rostedt, 2010) for analysis.

Figure 6.6 shows a trace of the previously discussed RTIC benchmarking application, where the two tasks have different priorities. The timeline contains two CPUs, which were involved and the two executor threads of RTIC. The events of interest are labeled 1-7 and are the following:

1. Thread P1 calls `FUTEX_WAIT` and goes to sleep.
2. Thread P2 calls `FUTEX_WAKE` to wake up thread P1.
3. Linux scheduler event `sched_waking`.
4. Linux scheduler event `sched_wakeup`.

5. Thread P2 returns from `FUTEX_WAKE` call.
6. Linux scheduler event `sched_switch`.
7. Thread P1 wakes up from the `FUTEX_WAIT` call.

From the events above, we can see that it takes around 10 microseconds to wake up a thread from the start of `FUTEX_WAKE` call (2) until the thread exits `FUTEX_WAIT` call (7). It is unclear what causes the latency, because the scheduler events `sched_*` are not documented and understanding the Linux scheduler code is not a trivial task. However, it is clear that the scheduling latency comes from the kernel itself and is not caused by inefficiencies in the RTIC implementation.

Investigating the Linux kernel scheduling overhead could be a topic for a future work. But it seems that this problem is already acknowledged by other developers and various patches are appearing. For example, Oskolkov (2020) submitted a patch introducing the `FUTEX_SWAP` operation, which allows user space applications to specify a thread to switch to, skipping the Linux scheduler. This patch later grew into a larger User-Managed Concurrency Groups (UMCG) framework (Oskolkov, 2021), which is currently in the Linux mailing list discussions. These patches could potentially improve RTIC's task switching performance.

6.2.2 Resource Locking Performance

The second test for the RTIC implementation is devised to benchmark resource locking performance and the backing `pcp-mutex` library. It is also split into two parts: contention-free and contented.

The contention-free test measures the resource locking and unlocking time when there is no resource contention. It was implemented as a single RTIC task that locks/unlocks resource 1 million times. Test results showed that locking the resource a single time took 309 ns, while unlocking took 171 ns. The longer locking time is expected, because locking operation involves a PCP check as described in Section 5.5.2. Overall, the timings seem normal.

```
#[task(priority = 2, shared = [a], local = [release_time])]
fn task1(mut cx: task1::Context, x: u32) {
    let start = cx.shared.a.lock(|a| {
        *a += 1;

        if x < NUM_SAMPLES {
            task2::spawn(x + 1).unwrap();
        } else {
            println!("Release time: {:?}", *cx.local.release_time / NUM_SAMPLES);
        }

        // Sleep so that task2 is able to run and block on the shared resource
        std::thread::sleep(Duration::from_micros(20));
        Instant::now()
    });
    *cx.local.release_time += start.elapsed();
}

#[task(priority = 1, shared = [a])]
fn task2(mut cx: task2::Context, x: u32) {
    cx.shared.a.lock(|a| {
        *a += 1;
    });
    task1::spawn(x).unwrap();
}
```

Listing 2: Contended Resource Unlocking Benchmark Code

The contented test measures the resource unlocking time when some other task is waiting for it. The code for the test is given in Listing 2 (irrelevant code excluded). The test works by firstly locking the shared resource in `task1` and then spawning `task2`. Because `task1` has a higher priority, it is momentarily put to sleep so that `task2` would get a chance to execute and block on the shared resource. This completes the setup for the test. Now, `task1` unlocks the shared resource and measures how much time it takes. The test is repeated for 50,000 samples (slow test). Experimental results show that unlocking a contended resource takes 5.43 µs. This is considerably slower than the contention-free test, but this is expected. When not contended, the `pcp-mutex` implementation is free of syscalls and only an atomic CAS instruction is needed to unlock. However, when mutex is contended (there are waiters), it must be unlocked via `FUTEX_WAKE` syscall, which introduces the given 5.43 µs delay.

6.3 Summary

From the GPIO interrupt latency tests it is clear that the Linux kernel has considerable scheduling overhead for both the kernel and the user threads. The non-threaded interrupts in the Linux kernel seem to have a comparable average latency to the bare-metal, but the kernel introduces long sporadic delays.

The RTIC application benchmarks confirm the Linux scheduler overhead. However, if the number of priority transitions in the application can be kept low, the scheduling becomes quite efficient.

7 Demonstrator

This chapter demonstrates the developed `linux-rtic` implementation for controlling a robot. A motorized two axis camera called JIWY (shown in figure 7.1) was selected as a testing platform. To challenge the implementation, a 10 kHz cascaded Proportional–Integral–Derivative (PID) motor controller was selected. It contains two PID control loops: 10 kHz current control loop and a 1 kHz position control loop.

Figure 7.1: The JIWY camera gimbal (Broenink and van Oort, 2020)

7.1 Setup

Due to the corona restrictions, testing on a real JIWY robot was not possible. Instead, a 20sim JIWY model (Broenink and van Oort, 2020) was used for the simulation. The model diagram is shown in Figure 7.2. The two boxes marked in purple and green show the cascaded PID controllers, which are run on the RPi4. The cyan box is the physical robot model, which is simulated.

To make the simulation as close to the real robot as possible, it was run on a separate microcontroller, which communicated over a full-duplex Serial Peripheral Interface (SPI) bus to exchange control and feedback signals. Because the real plant simulation requires intensive floating point calculations, a `NULCEO-H745ZI` development board with an `STM32H745ZIT6` microcontroller was selected, featuring a 480 MHz Cortex-M7 core with an FPU.

The simulator software on the STM32 was implemented on the original `cortex-m-rtic` framework by acting as an SPI slave device. The real plant 20sim model was exported as a C code and included in the RTIC application as a software task. The simulation frequency had

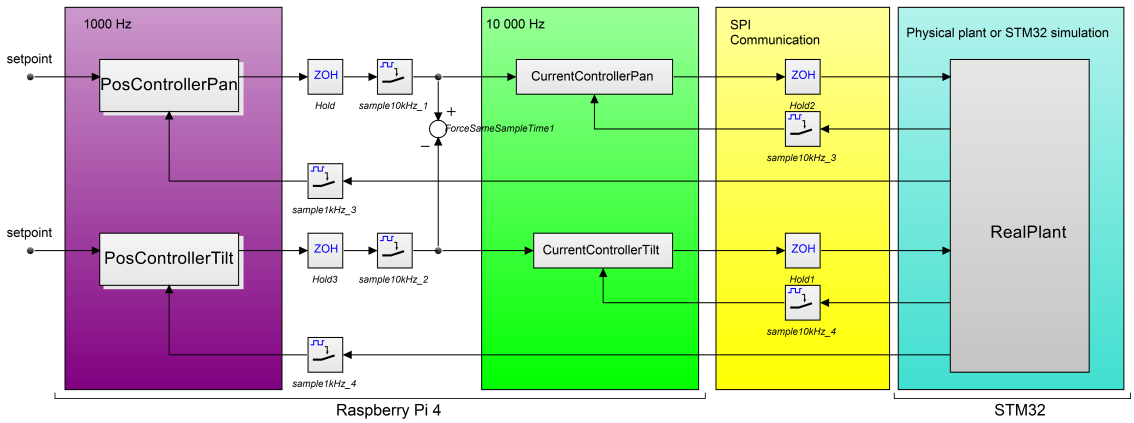

Figure 7.2: The 20sim JIWY model (Broenink and van Oort, 2020)

to be increased from 10 kHz to 40 kHz, because the simple Euler integration method diverged with aggressive PID parameters.

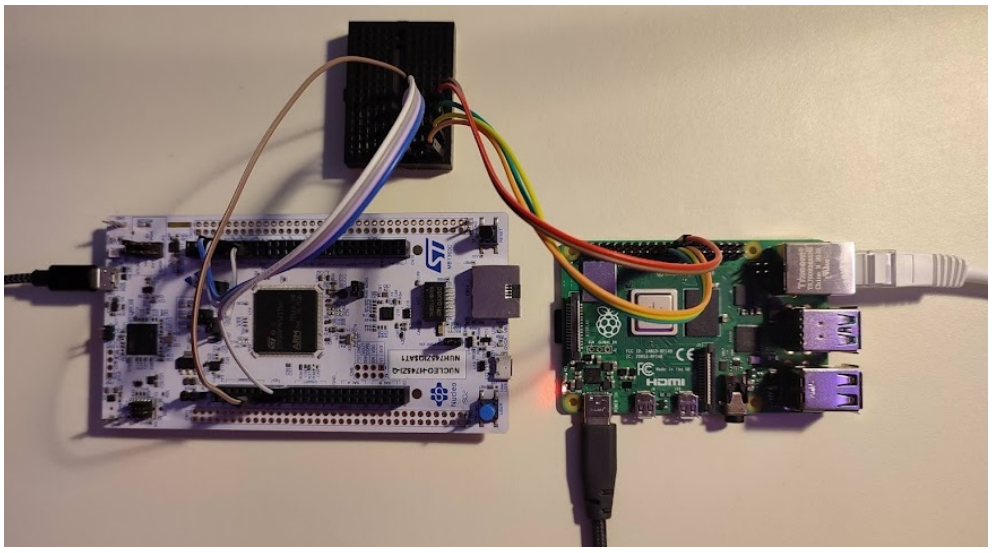

Figure 7.3: The simulator (NULCEO-H745ZI) and the JIWY controller (RPi4)

The whole hardware setup is shown in Figure 7.3.

7.2 Implementing Real-Time Control Software in RTIC

The same 20sim model was used to tune and export the current and position controllers for the RPi4. This time, however, C++ code export option in the 20sim had to be used as the generated C code contained global variables, which interfered when 4 controllers were used together. Special care had to be taken when interfacing with the unsafe C++ code, especially with regards to the pointers. Because all variables in Rust can be freely moved (if they are not referenced) they change memory locations and taking a raw pointer to such variable is dangerous. So any pointers used by the C++ code (objects, input/output arrays) had to be allocated in the heap with the Rust's `Box` type to keep the memory addresses constant.

The high level RTIC task diagram of the JIWY controller is given in Figure 7.4. Everything is centered around the `sample` task, which performs the SPI exchange: sends the new steering voltages from the shared resource and receives the position and current feedback from the simulator. It then selectively spawns controller tasks: the position controllers are run every 10th iteration (1 kHz loop), while the current controllers are run on every iteration (10 kHz loop).

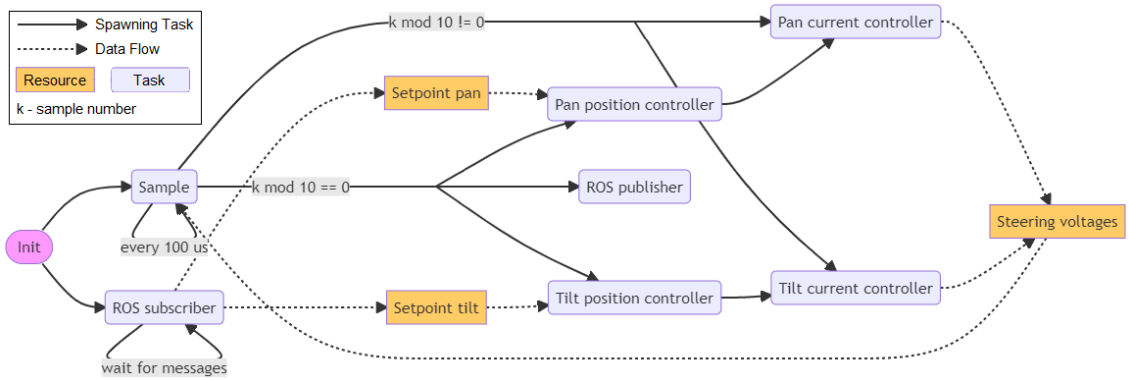

Figure 7.4: The JIWY controller task diagram

.

Note that when position controllers are run, the current controllers are chained to run after them rather than being spawned by the `sample` task. The final computed steering voltages are put into a shared resource and are used by the next `sample` task invocation, which is scheduled every 100 µs (10 kHz).

7.3 Integrating with non real-time software

One of the project goals was also an integration of real-time and non real-time software on a single system. Because the user space Linux implementation was chosen, this becomes very easy. All of the Rust standard API and crates are available to use in RTIC.

To control the demonstrator, Robot Operating System (ROS) was selected as it is a highly popular control software in robotics. The version 1 of ROS was used, because it was the best supported by the rust-native client library `rosrust` (Ademovic, 2016).

The `rosrust` library runs a ROS node on a separate thread and uses the two publisher and subscriber tasks to communicate with the real-time controllers (shown in Figure 7.4). ROS tasks are set to lower priority so they would not interfere with latency of real-time tasks. The publisher task periodically sends the current position on the `angles` topic and the subscriber task waits on the `setpoint` topic for messages to update the setpoints. Both topics use the standard `Vector3` message, with `x` representing the pan and `y` representing the tilt (`z` is ignored).

7.4 Results

When tested, the current controller seemed to oscillate, but it was fixed by slightly reducing the controller gain. It was likely caused by some random real-world jitter, which was not present in the 20sim simulation, because just a few percent reduction in gain fixed the problem.

There were also a few issues with the 10 kHz loop frequency, which was too much to be scheduled in time and caused some samples to be missed at random. Isolating the 4th core with the `isolcpus` kernel parameter partially cured the issue, but occasionally samples would still be lost. This was tracked down with the Linux tracing tools (also used in Section 6.1). Problem was with the Linux SPI driver, which spawns a kernel thread to manage the peripheral, but the thread's scheduling policy was not real-time. The priority was updated with `chrt -f -p 50 $(pidof spi0)` command and the missed samples were resolved.

To measure how well the controller behaves a plot was created by logging the controller state in a Comma-Separated Values (CSV) format with the following ROS utility command: `rostopic echo -p /angles`. Once recording, a command to change the setpoints and generate a step response was sent with the `rostopic pub` utility. The plot depicting the

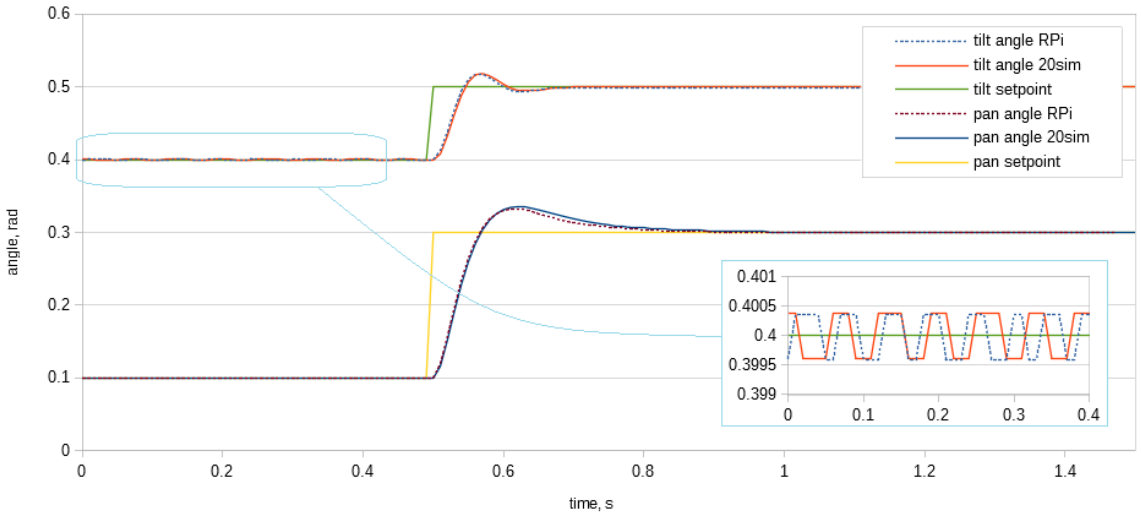

Figure 7.5: The JIWY controller step response on the RPi4

recorded step response is shown in Figure 7.5. It also contains a reference step response from 20sim that was obtained using the same step size and integration method (Euler). It can be seen that step responses are almost identical, suggesting that the controller is working correctly.

However, minor differences between the two responses can be seen and their cause is hard to predict. It could be due to the latency jitter, which is not present in the 20sim simulation. This jitter would cause samples to arrive inconsistently with the real plant simulation. Normally, the 10 kHz controller would run every 4 real plant simulation steps (40 kHz simulation). However, it has been shown in Chapter 6 that the Linux scheduler could have frequent latency spikes of up to about 50 µs, which would cause the controller to run every 2-6 samples instead of a constant 4. This irregular sample interval could cause the observed discrepancy.

Moreover, both the 20sim and the RPi4 angle plots have some oscillations, which are zoomed in on the lower-right corner of Figure 7.5. Their square wave nature suggests that this is due to the digital quantisation error. The oscillation peak-to-peak amplitude is about 767 µrad, which equals to the 13 bit rotary encoder resolution of 766.6 µrad per count, confirming the suspicion.

Unfortunately, the current controller response was not captured, because the RTIC application is running right on the edge of the RPi4 capabilities, utilizing about 90% of the assigned core. The low-priority ROS client did not have enough CPU time left to publish motor current messages at 10 kHz. However, this being a cascaded controller, we can assume from the correct position control that the current controller is working as designed.

8 Conclusions and Recommendations

8.1 Conclusions

In Chapter 3, various ways how the RTIC framework could be realized on the RPi4 were discussed. All of them included some trade offs between performance, implementation difficulty and usability.

The bare-metal approach was selected to measure the lower bound of RPi4's interrupt latency, which showed results of 1-2 µs. Even though the performance was really good, this approach is very hard to implement alongside the Linux kernel due to the system resource sharing problems. It is also very heavy in the driver implementation work to be usable for writing applications. Considering all the drawbacks of the bare-metal, it was skipped in favor of the user space Linux approach.

Implementing the RTIC framework on the Linux scheduler threads proven to be challenging, because RTIC's task and resource models were hard to map to the Linux kernel API. However, any deficiencies in the API were worked out by the two developed Rust crates: `futex-queue` and `pcp-mutex`, which allowed the `linux-rtic` to be successfully implemented.

The benchmarks in Chapter 6 showed that the Linux scheduler has a significant overhead, which negatively affected the performance of the developed implementation. On the other hand, the lesser performance is greatly compensated from the usability perspective. Writing real-time applications on the user space Linux RTIC is easy and they are highly portable. The `linux-rtic` framework should be able to run on any platform that supports real-time Linux.

The above points are reinforced with the demonstrator shown in Chapter 7, which uses the `linux-rtic` to control a two-axis robot. It quantified the actual performance and proved that running two motor current PID controllers at 10 kHz sampling frequency is possible on the RPi4 using the thread based RTIC.

In summary, for moderate performance requirements, the developed `linux-rtic` framework provides a convenient way for writing robust real-time applications on Linux platforms. It saves on money and development time by integrating real-time and non real-time software on a single system, as well as ensuring deadlock free and memory-safe program execution.

8.2 Recommendations

This work presented a lot of alternatives and trade offs, which could have been selected differently depending of the target application requirements. However, given the things learnt, the most viable future developments are as follows:

- If high bare-metal performance is desired, continuing the bare-metal implementation is a good choice. It could initially be done without having Linux on the same system, because most of the parts for this are almost complete. The Linux support could be added later as an alternative interrupt controller driver over a communication interface to the kernel.
- The developed `linux-rtic` performance could be improved by working on the Linux kernel. One potential scheduling solution is the mentioned UMCG framework (Oskolkov, 2021). It might also be worthwhile investigating the kernel traces and finding what causes the poor scheduling performance.
- The other less discussed approaches (kernel module, Dovetail interface) could also be investigated, but they were mainly skipped due to the maintenance burden, which is associated to any kernel modifications that are not included in the mainline kernel repository.

A Example RTIC Application

The best way to familiarize with RTIC framework is by reading through the RTIC online book (Lindgren et al., 2019). However, as a shorter introduction an example application is given here, which is shown in listing 3. Note that some irrelevant code parts such as imports were stripped down.

```
#[rtic::app(device = lm3s6965, dispatchers = [SSI0, SSI1])]
mod app {
    #[shared]
    struct Shared {
        counter: u32,
    }

    #[local]
    struct Local {}

    #[init]
    fn init(cx: init::Context) -> (Shared, Local, init::Monotonics) {
        foo::spawn_after(1.seconds()).unwrap();

        (Shared { counter: 0 }, Local {}, init::Monotonics())
    }

    #[task(priority = 10, shared = [counter], local = [counter: u32 = 0])]
    fn foo(cx: foo::Context) {
        println!("counter: {}", *cx.local.counter);
        // Access to local resource is lock-free
        *cx.local.counter += 1;

        // Access to shared resource requires locking
        cx.shared.counter.lock(|counter| {
            // Set shared counter to equal local counter
            *counter = *cx.local.counter;
        });

        // Spawn task instantly (with argument)
        bar::spawn(*cx.local.counter).unwrap();

        // Periodic every 1 second
        foo::spawn_after(1.seconds()).unwrap();
    }

    #[task(priority = 15, shared = [counter])]
    fn bar(cx: bar::Context, counter: u32) {
        // Print value passed as an argument
        println!("counter arg: {}", counter);

        // Print value from shared resource
        cx.shared.counter.lock(|counter| {
            println!("counter shared: {}", counter);
        });
    }
}
```

Listing 3: Example RTIC Application Code

The RTIC application begins with an `#[rtic::app(...)]` macro, which tells the compiler that the whole `app` module must be preprocessed by the macro code (`rtic-syntax` and `cortex-m-rtic-macros` crates). The macro also has additional properties, such as target

device and a list of hardware interrupts (dispatchers) for the software tasks. The code inside `app` module completely conforms to the Rust syntax, so Integrated Development Environments (IDEs) have no problem parsing the code.

Inside the `app` module there are two structures that define shared resources (that can be accessed from different tasks) and local resources (that are local to a single task). These resources can be initialized in `init` function, which is called once application starts. Local resources can also be directly defined in task headers as seen on line 18.

Finally, all tasks are defined with `#[task(...)]` attribute, which lets you specify task priority and all the resources that the task uses. These resources are then available under task context (`foo::Context` and `bar::Context`), any resources not defined in the task attribute are completely inaccessible to it. This way, the code generation step knows what resources are used by the tasks and priority ceilings can be calculated automatically.

B Appendix 1: Raspberry Pi Bare-Metal Development

This appendix is an overview of bare-metal application development on the RPi4. Section B.1 contains general information about the RPi4's hardware and its boot sequence. Section B.2 describes the tools used to aid the bare-metal development. Section B.3 explains how the RPi4 bare-metal crates were developed according to the Rust's embedded ecosystem standards.

B.1 Raspberry Pi 4 Hardware Boot Sequence

In addition to the four Cortex-A72 cores, the RPi4 also contains a proprietary VideoCore VI (VC6) processor, which shares the same Synchronous Dynamic Random-Access Memory (SDRAM) space. The VC6 processor communicates with the main Cortex-A cores and is used for offloading tasks such as video encoding/decoding and 3D graphics. Even though the official firmware of VC6 is closed-source, a large chunk of it was reverse engineered and published as an open-source alternative (Badea et al., 2017).

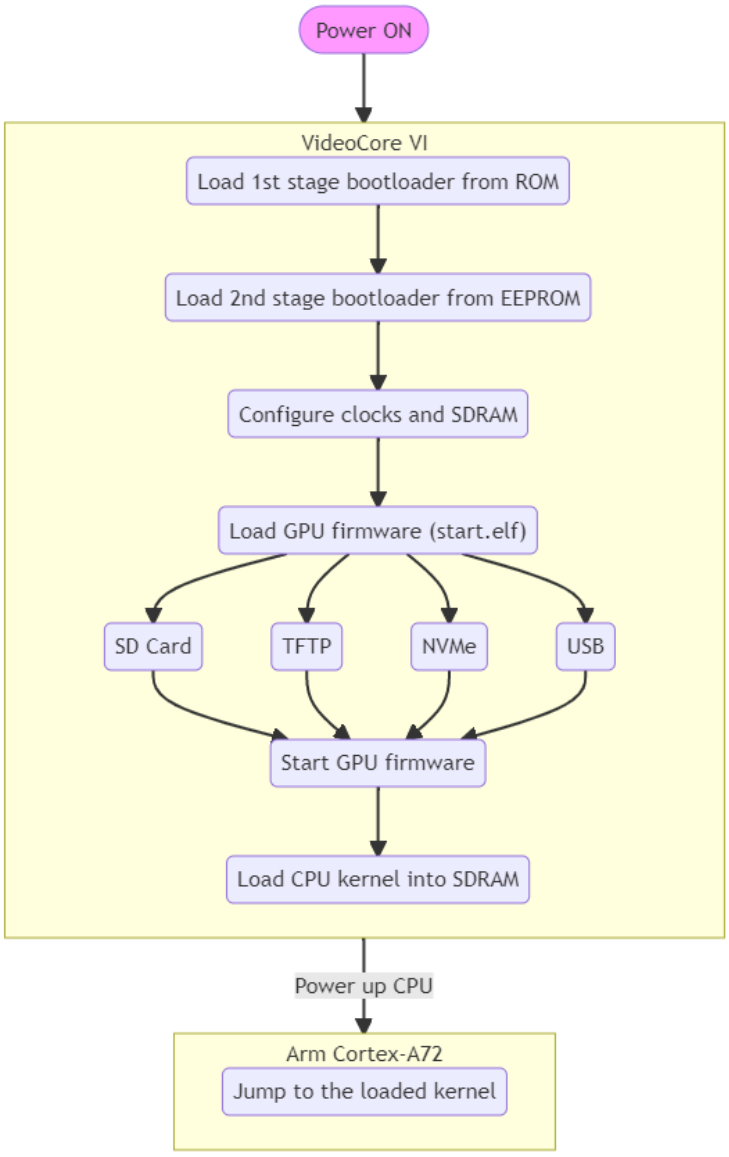

Figure B.1: RPi4's boot sequence

What is special about this VC6 processor, is that it handles the whole boot process, which is shown in figure B.1. The boot sequence, summarised from the Raspberry Pi (Trading) Ltd (2021a) documentation, proceeds as following:

1. When the RPi4 is powered on, the Cortex-A cores are disabled and the VC6 processor starts executing from the embedded Read-Only Memory (ROM), which contains the 1st stage bootloader.

2. The 1st stage bootloader then loads the 2nd stage bootloader into the L2 cache. On older Raspberry Pi models, the 2nd stage bootloader would be loaded from an SD card (`bootloader.bin` file), but the RPi4 has an additional Electrically Erasable Programmable Read-Only Memory (EEPROM), which is used to store the 2nd stage bootloader. This allows RPi4 to be booted without an SD card if network, NVMe or USB storage is used.

3. The 2nd stage bootloader performs various SoC initialisation tasks: setting up clock Phase-Locked Loops (PLLs), initialising SDRAM.

4. The VC6 firmware is now loaded into the SDRAM from the configured source: SD card, Trivial File Transfer Protocol (TFTP) server, NVMe disk or USB storage. The source is configured in the EEPROM configuration file. The firmware is started.

5. Finally, the VC6 firmware loads Cortex-A binary (kernel image) from the same configured source and starts the primary core. Other 3 cores are put into sleep loops as described in section 4.1.1.

For a bare-metal application to have a complete control over the SoC, the VC6 firmware would have to be replaced. However, developing firmware for the reverse-engineered hardware is difficult. Moreover, the VC6 is using a custom Instruction Set Architecture (ISA), which is not supported by the LLVM compiler and consequently by Rust. Instead, the original binary VC6 firmware is used to bootstrap the bare-metal application directly on the Cortex-A cores.

B.2 Bare-Metal Development Tools

It is important to have proper development tools, especially when writing bare-metal applications, where program behavior is hard to observe. This section describes a few tools that greatly aided in development: UART bootloader and JTAG debugger.

B.2.1 UART Bootloader

The standard way of loading application from the SD card is very tedious – the card has to be swapped each time the code is recompiled. To improve the development cycle speed, a UART bootloader was used. It is written once to the SD card and any applications are then uploaded via UART using XMODEM protocol (Christensen, 1984).

The Rust Raspberry Pi OS Tutorials (Richter, 2018) was used as a starting point for the bare-metal development. It is a tutorial series on writing a monolithic OS kernel for the Raspberry Pi 3 and 4 in Rust. Unfortunately, it being a monolithic kernel, meant that the code was hardly reusable for stand-alone bare-metal applications. Nevertheless, it contained a UART bootloader, which was adapted to the XMODEM protocol so that standard terminal applications could be used for uploading applications. Once proper HAL libraries were written (more about those in section B.3.2), the hacked bootloader was rewritten as a stand-alone application.

B.2.2 JTAG Debugging

The most common way of debugging embedded applications is a UART console. However, it is not always the best tool, especially when experiencing weird hardware errors or the console

itself stops working. In such cases a JTAG debugger is used, which allows inspecting memory, processor registers, setting breakpoints and more.

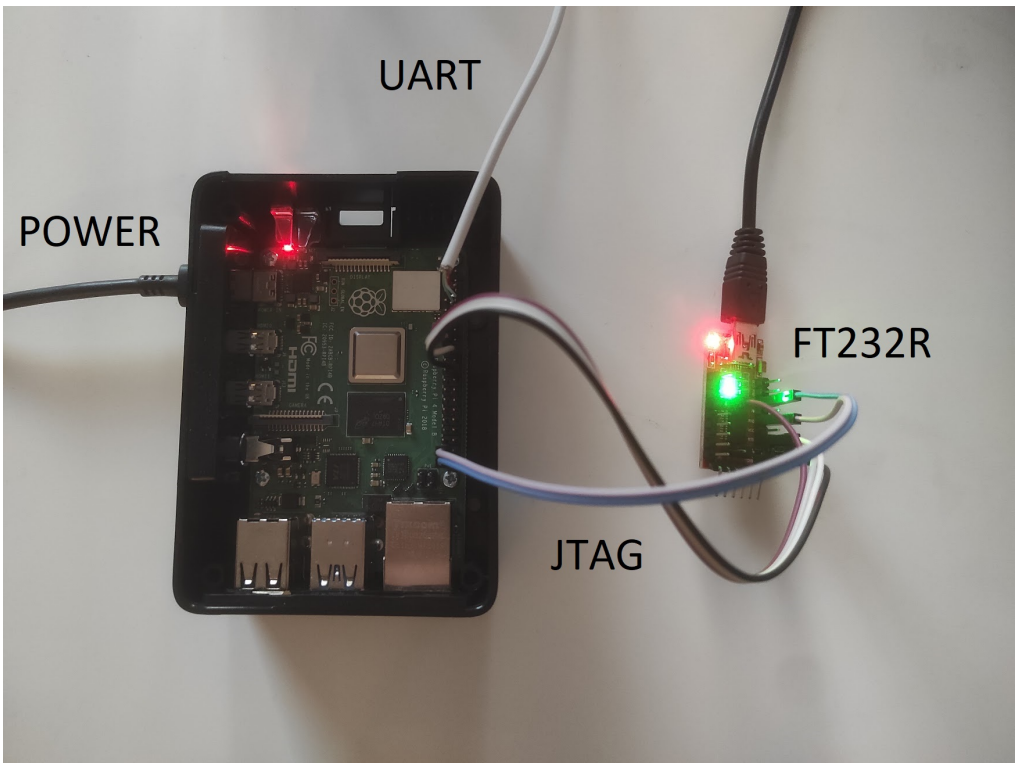

Figure B.2: FT232R JTAG debugger attached to the RPi4

JTAG debuggers are usually very expensive, however, it was discovered that the popular FT232R UART adapter can also be used as a JTAG adapter with the open source OpenOCD tool (Rath, 2005). The OpenOCD tool launches a GDB server for the well-known GDB debugger (Stallman, 1986). This setup was used successfully to debug cases of wrong addresses in jump instructions or memory alignment issues.

B.3 Writing Bare-Metal Code in Rust

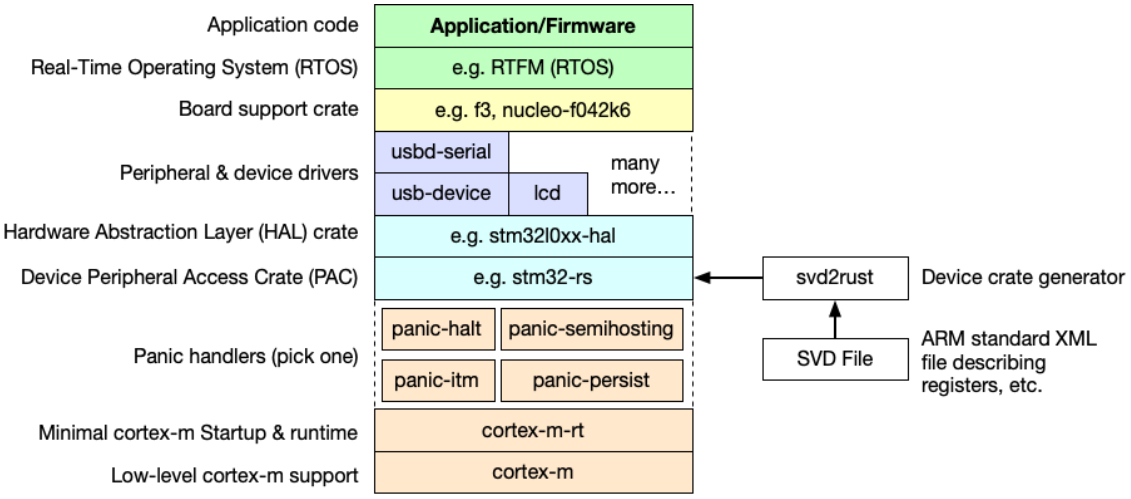

Figure B.3: Cortex-M embedded Rust ecosystem (Bishop, 2019)

The embedded Rust has an established ecosystem for the Cortex-M devices, which is shown in figure B.3. This multi-layer library structure was proven really successful as different domains were isolated from each other ensuring maximum portability. For example, the `cortex-m` and `lcd` crates can be used on both STM32 and NRF devices by just swapping the middle Peripheral Access Crate (PAC) and HAL crates.

Unfortunately, most of the crates are missing for the Cortex-A processors. The only crate that was available is the `cortex-a`, which provides register definitions for the Cortex-A architecture and was used by the mentioned Rust Raspberry Pi OS Tutorials. By sticking to the successful Cortex-M ecosystem model, the following crates had to be written:

- `cortex-a-rt` – low-level startup code, needed to reach Rust's `main()` function.
- `rpi-pac` – contains peripheral register definitions and memory addresses.
- `rpi-hal` – a Hardware Abstraction Layer (HAL), built on top of `rpi-pac` definitions.

B.3.1 Low-level Boot Code

The `cortex-a-rt` crate is responsible for performing a low-level initialization and calling the `main()` function. In addition to the initialization code it also contains a linker script, which defines the applications memory layout. Because linker script is specific to the target device, the equivalent `cortex-m-rt` crate uses a configuration file to specify RAM and FLASH regions of the device. This simple configuration may not be suitable for all Cortex-A devices, because of the widely different boot procedures. However, it is hard to predict requirements of other devices so the `cortex-a-rt` crates was written specifically for the RPi4 for now.

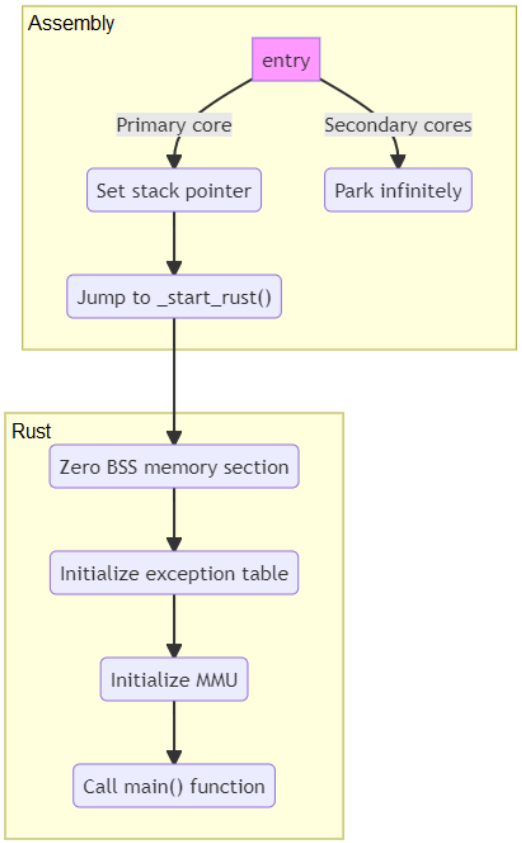

Figure B.4: `cortex-a-rt` initialisation sequence

The flowchart of the `cortex-a-rt` initialization code is shown in figure B.4. When Cortex-A cores are woken up by the VC6 firmware, they jump directly into the assembly entry code.

The sole purpose of this code is to setup the stack pointer, which is required to execute Rust. Additionally, the assembly code ensures that the 3 secondary cores are parked so that the single stack space is not accidentally corrupted. Support for multiple cores could be added later.

With the stack pointer configured, the rest of initialization is done by the Rust: zeroing the BSS memory section, which contains the zero-initialized global variables, initialising the exception table and MMU. With all these steps done, the user application's `main()` function is finally called.

The application is cross-compiled with the Rust's cargo tool by specifying 64-bit Arm target: `cargo build --target aarch64-unknown-none-softfloat`. The resulting binary is in ELF format, which contains headers, and in order to be loaded onto the RPi4 it must be stripped down to raw binary with `aarch64-none-elf-objcopy -O binary input.elf output.bin`.

B.3.2 Peripherals

The two developed crates `rpi-pac` and `rpi-hal` implement high-level API to the RPi4 peripherals. They could also easily be ported to the older Raspberry Pi 3 by supplying a different memory map in `rpi-pac`.

rpi-pac

The `rpi-pac` crate contains register definitions and the memory map of the peripherals. The PAC crates of the Cortex-M ecosystem are mostly automatically generated from the System View Description (SVD) files provided by the manufacturers, but unfortunately the Raspberry Pi does not provide them. Hence, the `rpi-pac` crate was manually written from the BCM2711 Arm Peripheral Documentation (Raspberry Pi (Trading) Ltd, 2020).

The PAC crates must also ensure Rust's memory safety rules and prevent memory aliasing. This is done by having a single accessor function `Peripherals::take()`, which returns peripherals only once. This is usually done in the application's `main()` function and the individual peripheral ownership is then given out to different software modules. However, because RPi4 has multiple cores there are also banked peripherals, which are local to each core (i.e. GIC CPU interfaces). They were split into a separate group `BankedPeripherals`, which are returned once per-core. It was implemented by an atomic bitmask variable, where each bit represents if a specific core has taken the peripherals.

rpi-hal

The `rpi-hal` crate depends on the definitions from the `rpi-pac` crate and implements the high-level API to the RPi4 peripherals. The GPIO, UART and GIC peripherals were implemented.

It also implements traits from the `embedded-hal` crate, which is a set of standardised peripheral APIs. This allows using device drivers (i.e. I/O expansion or LCD chips) on any HAL library that implements the standard traits. With this, all of the existing `embedded-hal` compatible device drivers can also be used on the bare-metal RPi4.

B.4 Summary

With the 3 developed crates (`cortex-a-rt`, `rpi-pac` and `rpi-hal`), writing bare-metal applications on the RPi4 is much easier. An example project called `cortex-a-quickstart` was written to demonstrate usage of the crates. The simple bare-metal hello-world program is just a few lines of code.

C PREEMPT_RT Patch on Raspberry Pi 4

The Raspberry Pi (Trading) Ltd (2021b) describes how to build a kernel for the RPi4, however, some of the documentation is outdated. Moreover, real-time patched versions of the kernel are no longer provided and it has to be patched manually.

All of the following steps are done directly on the RPi4 using the standard Raspberry Linux distribution so that no cross compiling is needed. It must be ensured that the RPi4 is running a 64-bit kernel, otherwise the resulting kernel will also be built for 32-bit.

As of time of writing, the latest Raspberry Pi kernel version is `5.10.47`, which is used in this guide. However, all of the steps should apply to newer kernel versions as well. It must be ensured that the Linux kernel version matches the real-time patch version, though.

1. Install build tools on the RPi4:

   ```
   sudo apt install git bc bison flex libssl-dev make
    ↪  libncurses-dev
   ```

2. Get the Raspberry Pi Linux kernel:

   ```
   git clone --depth=1 https://github.com/raspberrypi/linux
   ```

3. Get the real-time patch (check the cloned kernel version):

   ```
   wget http://cdn.kernel.org/pub/linux/kernel/projects/rt/5.10
    ↪  /patch-5.10.47-rt46.patch.gz
   ```

4. Prepare the kernel for the RPi4:

   ```
   cd linux
   KERNEL=kernel8
   make bcm2711_defconfig
   ```

5. Apply the real-time patch:

   ```
   gzip -cd ../patch-5.10.47-rt46.patch.gz | patch -p1
    ↪  --verbose
   ```

6. Enable real-time features:
   - Open kernel config with `make menuconfig`.
   - Enable `CONFIG_PREEMPT_RT_FULL` in Kernel Features -> Preemption Model -> Fully Preemptible Kernel (RT). This converts all spinlocks in the Linux kernel to sleeping mutexes.
   - Enable `HIGH_RES_TIMERS` in General setup -> Timers subsystem -> High Resolution Timer Support.
   - Set kernel frequency to 1000 Hz in Kernel Features -> Timer frequency.
   - Save and exit.

7. Build the kernel:

   ```
   make -j4 Image modules dtbs
   sudo make modules_install
   ```

8. Install all the device tree files and the new kernel with the `-rt` suffix to keep the old one:

```
sudo cp arch/arm64/boot/dts/broadcom/*.dtb /boot/
sudo cp arch/arm64/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm64/boot/dts/overlays/README /boot/overlays/
sudo cp arch/arm64/boot/Image /boot/$KERNEL-rt.img
```

9. Change the default kernel in `/boot/config.txt`:

```
kernel=kernel8-rt.img
```

10. Reboot the RPi4 and check if `uname -a` shows `PREEMPT RT`.

Bibliography

Ademovic, A. (2016), Pure Rust implementation of a ROS client library.
https://crates.io/crates/rosrust

Aparicio, J. (2017), Fearless concurrency in your microcontroller, [Online; accessed October 15, 2021].
https://blog.japaric.io/fearless-concurrency/

Aparicio, J. (2019), Real Time for The Masses on Linux, [Online; accessed June 10, 2021].
https://github.com/japaric/linux-rtfm

Arm Limited (2007), Application Note 176, [Online; accessed May 4, 2021].
https://developer.arm.com/documentation/dai0176/c

Arm Limited (2010), Cortex-M3 Devices Generic User Guide, [Online; accessed November 4, 2021].
https://developer.arm.com/documentation/dui0552/a

Badea, A., A. Rosenzweig and K. Brooks (2017), Open source VPU side bootloader for Raspberry Pi, [Online; accessed July 8, 2021].
https://github.com/librerpi/rpi-open-firmware

Baker, T. (1990), A stack-based resource allocation policy for realtime processes, in *Proceedings 11th Real-Time Systems Symposium*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 191,192,193,194,195,196,197,198,199,200, doi:10.1109/REAL.1990.128747.
https://doi.ieeecomputersociety.org/10.1109/REAL.1990.128747

Bishop, C. J. (2019), Embedded Rust ecosystem, [Online; accessed November 5, 2021].
https://craigjb.com/2019/12/31/stm32l0-rust/

Broenink, J. and G. van Oort (2020), Real-time software development 2020 — Exercise set 3, University of Twente Course Number: 2020-191211090-1B.

Buttazzo, G. C. (2011), *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*, Springer.

Christensen, W. (1984), XMODEM: A Standard Is Born, *PC Mag*, **vol. 3**, p. 451–452.

Gerum, P. (2001), Xenomai Project Announcement, [Online; accessed November 5, 2021].
https://www.mail-archive.com/rtl@fsmlabs.com/msg01156.html

Gleixner, T. and M. Kerrisk (2015), FUTEX - Linux Programmer's Manual, [Online; accessed June 15, 2021].
https://man7.org/linux/man-pages/man2/futex.2.html

Gleixner, T., C. Williams and J. Kacur (2007), Real-Time Linux Test Suite, [Online; accessed November 5, 2021].
https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git/

Google LLC (2020), Chromium Memory Safety, [Online; accessed November 4, 2021].
https://www.chromium.org/Home/chromium-security/memory-safety

van Heesch, D. (1997), Doxygen Documentation Generator.
https://www.doxygen.nl/

IEEE (2016), IEEE Standard for Information Technology—Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7, *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)*, pp. 1–3957, doi:10.1109/IEEESTD.2016.7582338.

Lattner, C. and V. Adve (2004), LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, in *CGO*, San Jose, CA, USA, pp. 75–88.

Lindgren, P., J. Aparicio et al. (2019), Real-Time Interrupt-driven Concurrency (RTIC). https://rtic.rs/

Lindgren, P., M. Lindner, A. Lindner, D. Pereira and L. M. Pinho (2015), RTFM-core: Language and implementation, in *2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*, pp. 990–995, doi:10.1109/ICIEA.2015.7334252.

Linux Kernel Developers (2012), Linux Remote Processor Framework Documentation. https://www.kernel.org/doc/Documentation/remoteproc.txt

Mantegazza, P., L. Dozio and S. Papacharalambous (2000), RTAI: Real time application interface, *Linux Journal*, **vol. 2000**, p. 10.

Oskolkov, P. (2020), Linux patch: introduce FUTEX_SWAP operation, [Online; accessed November 5, 2021]. https://lore.kernel.org/lkml/20200722234538.166697-2-posk@posk.io/

Oskolkov, P. (2021), Linux patch: add UMCG syscall stubs and CONFIG_UMCG, [Online; accessed November 5, 2021]. https://lore.kernel.org/linux-api/YKgVR5dM9RTZmCjh@gmail.com/T/

Ras, J. and A. M. Cheng (2009), An Evaluation of the Dynamic and Static Multiprocessor Priority Ceiling Protocol and the Multiprocessor Stack Resource Policy in an SMP System, in *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 13–22, doi:10.1109/RTAS.2009.10.

Raspberry Pi (Trading) Ltd (2020), BCM2711 ARM Peripherals, [Online; accessed July 8, 2021]. https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf

Raspberry Pi (Trading) Ltd (2021a), Raspberry Pi Documentation – Raspberry Pi Hardware, [Online; accessed July 8, 2021]. https://www.raspberrypi.com/documentation/computers/raspberry-pi.html

Raspberry Pi (Trading) Ltd (2021b), Raspberry Pi Documentation – The Linux Kernel, [Online; accessed July 8, 2021]. https://www.raspberrypi.com/documentation/computers/linux_kernel.html

Rath, D. (2005), *Open On-Chip Debugger*, Master's thesis, University of Applied Sciences Augsburg. https://openocd.org/files/thesis.pdf

Richter, A. (2018), Operating System development tutorials in Rust on the Raspberry Pi, [Online; accessed November 5, 2021]. https://github.com/rust-embedded/rust-raspberrypi-OS-tutorials

Rostedt, S. (2010), kernelshark - graphical reader for trace-cmd output, [Online; accessed October 25, 2021]. https://kernelshark.org/

Sha, L., R. Rajkumar and J. Lehoczky (1990), Priority inheritance protocols: an approach to real-time synchronization, *IEEE Transactions on Computers*, **vol. 39**, pp. 1175–1185, doi:10.1109/12.57058.

Stack Overflow (2021), Stack Overflow Annual Developer Survey, [Online; accessed November 4, 2021]. https://insights.stackoverflow.com/survey/

Stallman, R. (1986), GDB: The GNU Project Debugger, [Online; accessed November 5, 2021]. https://www.gnu.org/software/gdb/

The Linux Foundation (2015), The Real-Time Linux Collaborative Project Wiki, [Online; accessed June 23, 2021].
https://wiki.linuxfoundation.org/realtime/start

The Rust Foundation (2021), The Rust community's crate registry, [Online; accessed November 4, 2021].
https://crates.io/

The Xenomai Developers (2020), Dovetail Interface Documentation, [Online; accessed November 5, 2021].
https://evlproject.org/dovetail/

Xilinx, Inc. (2013), Zynq All Programmable SoC Linux-FreeRTOS AMP Guide, [Online; accessed November 5, 2021].
https://www.xilinx.com/support/documentation/sw_manuals/petalinux2014_2/ug978-petalinux-zynq-amp.pdf

Yaghmour, K. (2001), Adaptive Domain Environment for Operating Systems, [Online; accessed November 4, 2021].
http://www.opersys.com/ftp/pub/Adeos/adeos.pdf